

Broadview  
www.broadview.com.cn 博文视点



蒋金楠 著

电子工业出版社  
CHINA-PUB.COM  
http://www.chinapub.com

## 前言

第一次邂逅 WCF 是在微软举办的一场关于 Windows Vista 技术推广的培训上,时间大概是2005年10月份,当时我对 WCF 可谓一见钟情。如果读者也像我一样,之前习惯了采用.NET Remoting、XML Web Service、WSE、MSMQ 来架构分布式应用的话,应该不难想象我第一次接触 WCF 时心中的那份震撼。WCF 是 Windows 平台下所有分布式技术集大成者,它将这一系列独立的分布式技术整合,提供一个统一的应用编程接口,这本身就是一项创举。这些被整合的分布式技术不仅仅包含前面提到的这些,还包括 DCOM、Enterprise Service 等。WCF 并非单纯地将它们进行简单的累加,而是从底而上进行了革新性的重新设计,使 WCF 成为了一个可定制、可扩展的通信框架。

WCF, 全称 Windows Communication Foundation, 从命名上我们就不难看出微软对 WCF 所寄予的厚望,他们要将 WCF 打造成为 Windows 平台下唯一的通信基础框架。从 WCF 这些年的发展使用,以及未来趋势的预测来看,WCF 没有辱没这样的使命。这几年持续灼热的 SOA 尚未有降温的迹象,云计算又开始沸腾了。不论是 SOA,还是云计算,都须要解决一个核心的问题,那就是通信(Communication),而 WCF 解决的就是通信问题。相信大家不难想象 WCF 在整个 Windows 产品体系中将会具有怎样的一个“江湖地位”。

基于对 WCF 的浓厚兴趣,我这些年来一直没有放弃对 WCF 的研究,加上在众多项目中的实践,我逐渐对 WCF 有了一些积累。在写这本书之前,我阅读了现今已经出版的绝大部分 WCF 著作,并订阅了很多 WCF 专家的博客,一遍又一遍地翻看 MSDN,我想可能很少有人会像我一样一次又一次地从头到尾阅读 WCF MSDN。在获取知识的过程中,我发现了这样的现象:通过上述这些途径获取的资料绝大部分都仅限于 WCF 编程层面的介绍。若试图了解整个 WCF 一些底层的实现机制,是很难找到相关参考资料的。在这些年的学习和项目实践中,不论是对于 WCF 的编程模型、底层实现,还是设计思想,我感觉都有了一定的积累和沉淀。所以希望尽我所能,写一部全面剖析 WCF 的书籍,与读者分享。

在工作中,有读者可能会这样想:我仅仅是一个普通的 WCF 编程人员或项目实施人员,只须要知道 WCF 程序如何编写和配置就可以了,何须花那么多时间和精力去了解所谓的实现原理和设计模式呢。我个人认为这样的想法是片面的,正如一个对.NET Framework 不了解的人不可能写出高质量的.NET 程序一样,一个对 WCF 实现机制完全不了解的人也难以写出高质量的 WCF 服务。至于设计,如果不理解面向服务(SO: Service Orientation)的设计思想,还是按照传统的面向组件(CO: Component Orientation)来设计 WCF 服务,那将是对 WCF 最大的误用。

WCF 不但提供了强大的通信功能,而且还是一个极具可扩展性的通信框架。WCF 的通信实现是一个相对复杂的流程,它在整个通信处理流程中为每一个步骤提供了扩展点。用户可以通过实现相关的接口,或者继承相应的基类,自定义这些扩展的组件。最终通过配置或其

他方式（比如应用自定义特性）将这些定制的组件应用到 WCF 的整个处理流程中，从而改变 WCF 的通信行为，让 WCF 按照你希望的方式工作。当然，自由灵活地对 WCF 进行扩展要建立在你对 WCF 的底层实现充分了解的基础之上。

由于 WCF 涉及的内容实在太多，《WCF 技术剖析》不得不拆分为多卷来写。卷1首先出版，随后会进行卷2的写作。

## **本书的特点**

如果说《WCF 技术剖析》具有哪些市面上其他 WCF 图书不具备的特点的话，我觉得可以通过以下三个“注重”来概括。

### **注重原理**

“知其然，且知其所以然”，对一项技术实现原理的把握能够帮助你更加有效地利用这项技术。WCF 建立在 .NET 平台下，提供基于托管代码的应用编程接口（API），在编程层面不会太复杂，也很容易入手。但是，如果希望通过 WCF 构建一个具有高性能、高可维护性、高可扩展性的分布式应用，就要求架构师、设计师和开发者对 WCF 的实现机制，以及面向服务的设计原理具有正确、全面而深入的理解。当然，介绍 WCF 编程的内容，任何一本关于 WCF 的图书都是必需的，《WCF 技术剖析》也不例外，不过，它和那些完全介绍 WCF 编程的书不同的是，本书中此类内容仅占全部内容的一半左右。

### **注重细节**

“细节决定成败”，如果将 2/8 原则应用到编程领域，它将体现在：花 20% 的时间编写出 80% 程序主体，而 80% 的时间用于编写剩下 20% 核心程序并纠错和解决 Bug。这些 Bug 的产生往往由于对细节不够重视所导致。《WCF 技术剖析》会介绍一些常被人们忽视而导致产品 Bug 的细节，这些细节多源自于笔者这些年来实践的积累和总结。

### **注重实践**

“实践出真知”，不断地在具体应用中进行实践是学习 WCF 最有效的手段。实践是检验真理的唯一标准，通过将所学的 WCF 的知识应用到一个真正的应用之中，才能确保我们掌握的知识的正确性。此外，实践不但可以巩固我们的所学，还会让我们意识到不足。《WCF 技术剖析》在每一章节都会提供一系列的案例演示，通过一个个具体案例应用去实践 WCF。

## **本书为谁而作**

本书的内容不仅适合那些尚未接触过 WCF，希望尽快入门并进行深入研究的开发人员，同样适合那些对 WCF 具有一定了解的开发设计人员和架构师。相信不同层次的读者都能从本书中找到自己希望了解的部分。阅读本书的读者须要对 .NET，包括对 C# 和 .NET Framework 具有一定的了解。如果读者具备了 DCOM、Enterprise Library Service、.NET Remoting、Web

Service、MSMQ 及 SOA 相关的基础，对阅读此书尽快掌握 WCF 将大有裨益。

## **本书的结构**

本书以分卷的形式出版，第1卷率先完稿，后续部分也已列入作者的写作计划。《WCF 技术剖析（卷1）》涵盖 WCF 最基本的框架，相关的内容已经赋予了读者构建一个基本 WCF 应用的能力。本卷共分10章，各章的内容如下。

### **第1章 WCF 简介（WCF Overview）**

本章简单讲述了 WCF 产生的历史背景，以及 WCF 在微软产品线中所处的地位。本章的最后将提供一个功能简单，但结构完整的 WCF 事例应用程序。该事例应用程序涵盖了构建一个基本 WCF 应用所需的所有步骤，其中包括服务契约（Service Contract）的定义、服务的实现、服务的寄宿（Service Hosting）、元数据（Metadata）的发布和导入、服务代理的创建和服务调用等。在一步一步对案例应用进行演示的过程中，还穿插介绍了 WCF 的一些基本概念和原理，比如终结点（Endpoint）、地址（Address）、绑定（Binding）、契约（Contract）、元数据（Metadata）、服务寄宿（Service Hosting）等。

### **第2章 终结点地址与 WCF 寻址（Endpoint Address and WCF Addressing）**

本章着重介绍终结点三要素之一的“地址（Address）”和相关的寻址（Addressing）机制。在本章中，我们会谈到不同网络协议地址之间的差异，以及如何在服务寄宿和服务调用的时候通过代码或配置的方式设定终结点的地址。本章涉及的内容还包括通过地址报头（Address Header）的形式为消息添加寻址信息，以及端口共享在 WCF 中的应用。本章的最后将深入介绍 WCF 下寻址的实现机制。

### **第3章 绑定和信道栈（Binding and Channel Stack）**

本章着重介绍终结点的第二个元素：“绑定（Binding）”，并以绑定作为切入点，对 WCF 整个信道层进行深入而详细的讲解。本章会介绍 WCF 信道层所涉及的所有相关组件，包括信道（Channel）、信道管理器（Channel Manager）、信道监听器（Channel Listener）、信道工厂（Channel Factory）、绑定元素（Binding Element），以及绑定上下文（Binding Context）等。在本章的最后还会对常见的系统绑定进行全面的剖析和比较，并且指导读者创建自定义的绑定。

### **第4章 服务契约（Service Contract）**

终结点的服务契约元素的介绍放在本书的第4章。为了让读者深入理解契约的本质，在本章的一开始，我们将从“抽象与接口”、“服务描述”及“消息交换模式”全方位、多角度透视 WCF 中的服务契约。紧接着，将详细介绍如何通过 ServiceContractAttribute 和 OperationContractAttribute 这两个自定义特性来定义服务契约。在本章的最后，将会深

入探讨操作契约和消息交换模式之间的关系，以及如何定义适合多线程场景中的服务契约。

## 第5章 序列化与数据契约 (Serialization and Data Contract)

本章着重介绍 WCF 对“数据”的处理，包括数据的定义（数据契约）和数据的序列化及反序列化。本章将从序列化在一个分布式应用中所起的重要作用谈起，然后详细介绍数据契约的定义，以及数据契约序列化器（DataContract Serializer）进行序列化和反序列化的实现原理和规律。本章涉及的内容还包括：如何为数据契约序列化器设定已知类型（KnownType），以及已知类型在序列化和反序列化过程中所起的重要作用；如何定义基于泛型数据契约和集合数据契约；等价数据契约在 WCF 消息交换中的意义。在本章的最后，我们将介绍在整个 WCF 消息分发、处理流程中，是如何实现数据的序列化和反序列化的。

## 第6章 消息、消息契约与消息编码 (Message, Message Contract and Message Encoding)

本章的所有内容都是围绕着消息（Message）展开的，首先我们会通过 SOAP1.2 规范介绍一个 SOAP 消息的基本结构，并由此引出消息在 WCF 的表示：System.ServiceModel.Channels.Message 类型的介绍。在介绍 Message 类型的时候，将重点介绍消息处理中消息对象表现出来的状态机（State Machine）。接下来会介绍消息契约（Message Contract）的定义，以及消息契约的应用场景的选择。消息编码（Message Encoding）是本章的重点，我们会对 WCF 采用的3种典型的编码方式进行全面分析和比较，在本节的最后部分还会深入介绍消息编码分别在 WCF 服务端与客户端框架中的实现原理。

## 第7章 服务寄宿 (Service Hosting)

服务的寄宿是部署服务必需的步骤，为了让读者了解服务寄宿（Service Hosting）的本质，我们会从服务描述（Service Description）谈起。接下来再深入探讨在服务寄宿的每一个步骤中，WCF 内容为我们作了哪些“鲜为人知”的操作。本章的后半部分，我们会讨论4种常见的服务寄宿方式，包括自我寄宿（Self-Hosting）、IIS 寄宿（IIS Hosting）、WAS 寄宿（WAS Hosting）和 Windows Service 寄宿（Windows Service Hosting）。在介绍基于 IIS 服务寄宿的过程中，我们会对不同版本的 IIS（IIS5.x, IIS 6.0, IIS 7.0）的工作机制进行全面的分析和比较，并就 IIS 和 ASP.NET 管道之间的通信进行详细介绍。此外，还会深入介绍两种不同模式下，即 ASP.NET 并行（ASP.NET Side by Side）模式和 ASP.NET 兼容（ASP.NET Compatible）模式，实现 WCF 服务寄宿的实现原理和表现行为。

## 第8章 客户端 (Client)

本章主要介绍在客户端如何创建服务代理进行服务调用，以及 WCF 客户端框架内部如何完成一次正常的服务调用。WCF 具有两种典型的服务调用方式：通过添加服务引用或通过相应的工具导入元数据并声称客户端代理类型（继承自 ClientBase<T>）和相关配置；借助

ChannelFactory<T>直接创建服务代理对象。通过本章的介绍，你将对这两种服务调用方式具有一个全面的认识，本章还将深入剖析 ClientBase<T>和 ChannelFactory<T>这两个重要的类型。在本章的最后部分会为你提供基于会话服务调用的最佳实践。

## 第9章 实例管理与会话 (Instancing and Session)

实例化 (Instancing) 和会话 (Session) 是 WCF 中两个重要的概念，前者旨在实现服务实例对象的激活，后者则实现对客户端调用状态的保持。本章将详细介绍 WCF 3种典型的实例化模式所表现的行为、实现的原理及各自适合的场景。在介绍会话的部分会深入剖析会话如何保持客户端多次服务调用的状态，以及会话、实例化模式、绑定和信道之间的关系。

## 第10章 WCF 实例研究 (WCF in Practice)

《WCF 深入剖析 (卷1)》将为你提供一个 WCF 版本的 PetShop，通过一个具体实在的例子，指导读者如何利用 WCF 构建一个完整的分布式应用。在该案例应用中，不仅仅会为你提供单纯 WCF 相关的应用，还引入了一系列设计模式，包括：模块化与层次化设计、面向方面编程 (AOP, Aspect Oriented Programming)、控制倒置 (IoC, Inverse of Control) 与依赖注入 (DI, Dependency Injection)、MVP (Mode-View-Presenter) 模式等。我们还会根据 PetShop，介绍如何通过 WCF 扩展实现与微软开源开发框架企业库集成，比如通过与 Unity 的继承将 DI 容器引入 WCF 应用服务实例的创建；通过与 EHAB (Exception Handling Application Block) 继承实现可配置的异常处理；通过 Unity 与 PIAB (Policy Injection Application Block) 将实现基于 AOP 的编程等。

### 关于《WCF 技术剖析 (卷2)》

《WCF 技术剖析 (卷2)》将按照与卷1相似的方式对 WCF 一些相对“高级”的主题进行介绍，大概的内容包括：异常处理 (Exception Handling)、元数据 (Metadata)、并发与流量限制 (Concurrency and Throttling)、事务 (Transaction)、安全 (Security)、WCF 分发体系 (WCF Dispatching System)、WCF 扩展 (WCF Extension) 等。

### 关于作者

蒋金楠，网名 Artech，现就职于某知名软件公司担任高级软件顾问 (Senior Software Consultant)。微软解决方案架构 (Solutions Architecture) 与互联系统 (Connected System) 双料 MVP (最有价值专家)，具有5年以上软件开发设计与架构经验。对 .NET Framework、C#、ASP.NET、SQL Server、设计模式、软件架构及主流的开源框架具有深入的研究。属国内较早接触 WCF 的人之一，同时对 .NET Remoting、MSMQ 通信技术有深入的理解。2007年2月起在个人博客 (<http://www.cnblogs.com/artech>) 上发表数十篇深入介绍 WCF 的文章，成为国内 WCF 技术最早的推广者之一。

### 致谢

本书得以出版，最应该感谢的人是博文视点的资深编辑周筠女士和杨绣国(Lisa)女士，她们的专业能力和认真的工作态度给我留下深刻的印象，感谢 Lisa 在我因工作原因不能按时交稿时给予的理解。两个月后，Lisa 即将初为人母，在这里表示衷心的祝福。同时，要感谢博客园的创始人杜勇为我们创建了一个这么好的技术学习和交流的平台，而且本书的出版也依赖于杜勇本人的推荐。此外，还要感谢郭金链、葛子昂、黄昕、李会军、孟永刚、王翔、曲春雨、王森（台湾）、张逸、张玉彬等人在百忙之中为本书审稿，并提出宝贵的建议，本人收益良多。同时曲春雨、李会军和王翔三位老师还为本书写了推荐序，感谢之情无以言表。最后要感谢我博客文章的所有读者，是你们让我具有了创作的勇气。

当然，必须感谢我的父母，他们赐予我一顆不算愚笨的脑袋，并从小培养我独立思考的习惯。感谢我未来的老婆徐妍妍，当我才思枯竭的时候同她谈心总能让我灵感四溢。在本书创作期间，发生了一些事情给她们全家带来了难以愈合的伤口，希望时间能够尽快治愈她们心灵的创伤，祝她的家人永远健康幸福。

## 本书的支持

本书涉及的很多 WCF 底层实现的内容，大多不能通过官方的渠道获取。它们来自本人对 WCF 源代码的分析、通过应用程序的证明，以及这些年来使用 WCF 经验的总结。由于能力有限，对于本书涉及的内容，难免存在原理或表达上的偏差。如果读者在阅读本书过程中，发现任何问题可以直接向我本人反馈。如果你遇到任何 WCF 相关的问题，也可以和我一起交流。

## 目录

第1章 WCF 简介 (WCF OVERVIEW)	1
1.1 SOA 的基本概念和设计思想	2
1.2 WCF 是对现有 WINDOWS 平台下分布式通信技术的整合	4
1.3 构建一个简单的 WCF 应用	6
1.3.1 步骤一 构建整个解决方案	7
1.3.2 步骤二 创建服务契约	7
1.3.3 步骤三 创建服务	8
1.3.4 步骤四 通过自我寄宿的方式寄宿服务	9
1.3.5 步骤五 创建客户端调用服务	12
1.3.6 步骤六 通过 IIS 寄宿服务	16
第2章 终结点地址与 WCF 寻址 (ENDPOINT ADDRESS AND WCF ADDRESSING)	19
2.1 ENDPOINTADDRESS	20
2.1.1 URI	22
2.1.2 如何指定地址	24
2.1.3 如何指定 AddressHeader	36

2.2	端口共享 (PORT SHARING)	42
2.2.1	端口共享在 WCF 中的意义何在?	43
2.2.2	基于 HTTP HTTPS 的端口共享	44
2.2.3	基于 TCP 的端口共享	45
2.3	WCF 寻址 (ADDRESSING) 详解	47
2.3.1	服务的角色	47
2.3.2	逻辑地址和物理地址	48
2.3.3	ListenUri 和 ListenUriMode	50
2.3.4	消息筛选	53
2.3.5	案例演示: 通过 tcpTracer 进行消息的路由	57
第3章	绑定与信道栈 (BINDING AND CHANNEL STACK)	67
3.1	绑定简介	68
3.1.1	信道 (Channel) 与信道栈 (Channel Stack)	68
3.1.2	绑定与信道栈 (Binding and Channel Stack)	69
3.1.3	案例演示: 如何直接通过绑定进行消息通信	70
3.1.4	WCF 的绑定模型	74
3.2	绑定编程	75
3.2.1	服务寄宿对绑定的指定	75
3.2.2	服务调用对绑定的指定	79
3.3	信道与信道栈	84
3.3.1	CommunicationObject 与 DefaultCommunicationTimeouts	84
3.3.2	IChannel 和 ChannelBase	87
3.3.3	消息交换模式与信道形状 (Channel Shape)	88
3.3.4	案例演示: 如何自定义信道	96
3.4	信道管理器 (CHANNEL MANAGER)	101
3.4.1	信道监听器 (Channel Listener)	101
3.4.2	信道工厂 (Channel Factory)	106
3.5	绑定与绑定元素 (BINDING AND BINDING ELEMENT)	110
3.5.1	绑定元素 (Binding Element)	110
3.5.2	绑定揭秘	113
3.6	系统绑定与自定义绑定 (SYSTEM DEFINED BINDING & CUSTOM BINDING)	122
3.6.1	系统绑定	122
3.6.2	自定义绑定	131
第4章	服务契约 (SERVICE CONTRACT)	137
4.1	服务契约透视	138



- 4.1.1 抽象、接口与服务契约 138
- 4.1.2 元数据与服务契约 139
- 4.1.3 WSDL、XSD 与服务契约 139
- 4.1.4 消息交换与服务契约 140
- 4.1.5 WCF 是 CLR 类型与厂商无关服务描述的适配器 141
- 4.2 服务契约编程接口 142
  - 4.2.1 ServiceContractAttribute 与 OperationContractAttribute 142
  - 4.2.2 为终结点指定契约 154
  - 4.2.3 服务契约的继承 158
  - 4.2.4 操作重载与操作选择 164
- 4.3 消息交换模式 (MEP) 与服务操作 167
  - 4.3.1 请求-回复模式下的服务契约与操作 167
  - 4.3.2 单向 (One-way) 模式下的服务契约与操作 171
  - 4.3.3 双工模式下的服务契约与操作 172
- 4.4 多线程与异步操作 181
  - 4.4.1 异步信道调用 182
  - 4.4.2 异步服务实现 186
- 第5章 序列化与数据契约 (SERIALIZATION AND DATA CONTRACT) 191
  - 5.1 漫谈序列化 192
    - 5.1.1 封送 (Marshaling) 与序列化 192
    - 5.1.2 持久化 (Persisting) 与序列化 193
    - 5.1.3 数据结构与序列化 193
    - 5.1.4 XML 序列化器 194
  - 5.2 数据契约与数据契约序列化器 198
    - 5.2.1 数据契约的本质 199
    - 5.2.2 数据契约的定义与数据契约序列化器 200
  - 5.3 已知类型 (KNOWN TYPE) 211
    - 5.3.1 未知类型导致序列化失败 211
    - 5.3.2 DataContractSerializer 的已知类型集合 213
    - 5.3.3 基于接口的序列化 214
    - 5.3.4 KnownTypeAttribute 与 ServiceKnownTypeAttribute 215
  - 5.4 泛型数据契约与集合数据契约 217
    - 5.4.1 泛型数据契约 217
    - 5.4.2 数据契约对数组与集合的支持 221
    - 5.4.3 IDictionary<TKey, TValue>与 Hashtable 231

5.5	等效数据契约与数据契约版本控制	236
5.5.1	数据契约的等效性	236
5.5.2	数据成员的添加与删除	237
5.5.3	数据契约代理 (Surrogate)	244
5.6	序列化 WCF 框架中的实现	249
5.6.1	MessageFormatter	249
5.6.2	MessageFormatter 在 WCF 框架中的应用	250
第6章	消息、消息契约与消息编码 (MESSAGE, MESSAGE CONTRACT AND MESSAGE ENCODING)	253
6.1	SOAP 与 WS-ADDRESSING	254
6.1.1	SOAP (基于 SOAP 1.2标准)	254
6.1.2	WS-Addressing (基于 WS-Addressing 1.0)	260
6.2	消息 (MESSAGE)	262
6.2.1	消息版本 (Message Version)	262
6.2.2	如何创建消息	264
6.2.3	消息的基本操作和消息状态	272
6.2.4	消息报头集合	276
6.3	消息契约 (MESSAGE CONTRACT)	284
6.3.1	消息契约的定义	284
6.3.2	案例演示: 基于消息契约的方法调用是如何格式化成消息的?	289
6.4	消息编码 (MESSAGE ENCODING)	294
6.4.1	序列化 (反序列化) 和编码 (解码)	294
6.4.2	XmlDictionary、XmlDictionaryWriter 和 XmlDictionaryReader	295
6.5	消息编码在 WCF 框架中的实现	302
6.5.1	消息编码器 (MessageEncoder)	303
6.5.2	案例演示: 通过 MessageCoder 对消息进行编码	304
6.5.3	WCF 体系下的编码机制实现	306
第7章	服务寄宿 (SERVICE HOSTING)	309
7.1	服务描述 (SERVICE DESCRIPTION)	310
7.1.1	ServiceDescription 与 ServiceBehavior	310
7.1.2	ServiceEndpoint 与 EndpointBehavior	314
7.1.3	ContractDescription 和 ContractBehavior	315
7.1.4	OperationDescription 和 OperationBehavior	316
7.2	服务寄宿详解	317
7.2.1	创建 ServiceHost	317

7.2.2	开启 ServiceHost	318
7.3	WCF 服务的自我寄宿 (SELF-HOSTING)	325
7.3.1	案例演示：如何通过 Windows 应用进行服务寄宿	325
7.3.2	自定义 ServiceHost	330
7.4	通过 IIS 进行服务寄宿	333
7.4.1	案例演示：如何通过 IIS 进行服务寄宿	334
7.4.2	IIS 管道与 ASP.NET 架构	337
7.4.3	IIS 服务寄宿实现详解	350
7.4.4	案例演示：利用 ASP.NET 兼容模式创建支持会话 (Session) 的 WCF 服务	356
7.5	通过 WINDOWS SERVICE 进行服务寄宿	360
7.5.1	案例演示：如何通过创建 Windows Service 寄宿 WCF 服务	361
第8章	客户端 (CLIENT)	365
8.1	WCF 客户端框架简述	366
8.1.1	从透明代理 (Transparent Proxy) 和真实代理 (Real Proxy) 说起	366
8.1.2	通过自定义 RealProxy 实现方法调用的劫持 (Interception)	368
8.1.3	案例演示：通过自定义 RealProxy 实现日志功能	369
8.1.4	WCF 客户端是如何进行服务调用的	372
8.1.5	案例演示：创建一个托管应用模拟最简单的 WCF 框架	373
8.2	CHANNELFACTORY<T>和 DUPLEXCHANNELFACTORY<T>详解	383
8.2.1	创建 ChannelFactory<T>和 DuplexChannelFactory<T>	383
8.2.2	开启 ChannelFactory<T>和 DuplexChannelFactory<T>	387
8.2.3	创建服务代理	392
8.2.4	通过服务代理进行服务调用	393
8.3	CLIENTBASE<T>揭秘	398
8.3.1	ClientBase<TChannel>简介	398
8.3.2	ChannelFactory<T>的缓存机制	402
8.4	基于会话信道的客户端	405
8.4.1	服务契约的关闭与并发会话的限制	405
8.4.2	会话信道与异常处理	408
8.4.3	案例演示：通过 AOP 的方式解决会话信道的关闭与中断	410
8.4.4	额外的思考：性能与并发的权衡	415
第9章	实例管理与会话 (INSTANCING AND SESSION)	417
9.1	实例上下文 (INSTANCECONTEXT) 与实例上下文模式 (INSTANCECONTEXTMODE)	418

9.1.1	实例上下文 (Instance Context)	418
9.1.2	实例上下文模式 (InstanceContext Mode)	420
9.1.3	实例服务行为	422
9.2	单调 (PER-CALL) 实例上下文模式	422
9.2.1	单调模式下的服务实例上下文提供机制	423
9.2.2	案例演示: 单调模式下服务实例的生命周期	423
9.2.3	服务实例上下文的释放	425
9.2.4	单调模式与可扩展性	427
9.3	单例 (SINGLE) 实例上下文模式	428
9.3.1	案例演示: 演示服务实例的单一性	429
9.3.2	单例模式下服务实例上下文提供机制	430
9.3.3	单例服务与可扩展性	433
9.4	会话 (SESSION) 与会话 (PER-SESSION) 实例上下文模式	435
9.4.1	WCF 会话简介	435
9.4.2	WCF 会话编程模型	437
9.4.3	会话 (Per-Session) 实例上下文模式	442
9.5	会话模式、绑定与实例上下文模式	445
9.5.1	单调服务决定于单调实例上下文模式	445
9.5.2	单例服务决定于单例实例上下文模式	448
9.5.3	会话服务决定于会话信道 (栈) 和会话实例上下文模式	448
9.6	WCF 服务实例上下文提供机制	450
9.6.1	服务实例上下文的提供者 (InstanceContextProvider)	451
9.6.2	服务实例的提供者 (InstanceProvider)	454
9.6.3	服务实例的释放	455
第10章	WCF 实例研究 (WCF IN PRACTICE)	457
10.1	实例应用功能与结构概述	458
10.1.1	PetShop 功能简介	458
10.1.2	PetShop 的物理结构	460
10.1.3	PetShop 的模块划分	460
10.1.4	PetShop 模块的层次划分	466
10.2	PETSHOP 设计原理	482
10.2.1	如何实现用户验证	482
10.2.2	上下文的共享及跨域传递	488
10.2.3	异常处理	494
10.2.4	依赖注入在 PetShop 中的应用	499

10.2.5 AOP 在 PetShop 中的应用 504

10.2.6 MVP 模式在 PetShop 中的应用 507

参考文献 513

索引 515

## 推荐序一

Windows Communication Foundation (WCF) 是用来在不同应用间进行互通信的一个编程框架,它是.NET Framework 中偏重于通信的重要组成部分。原代码名为 Indigo 的这个编程框架,在其 Beta2 版本时正式更名为 WCF,于2006年12月作为.NET Framework 3.0的4套 API 之一发布。

从 Native Win32到 Managed 的世界,应用进程间的通信机制一直是开发者最为关注的方面,在.NET Framework 2.0 (2005年11月发布)及以前的 CLR 版本中,微软提供了若干套相互分开的 APIs,来支持应用间的互通信:有相对基础的 Managed Socket 机制;有为二进制优化而设计的.NET Remoting (同时支持 TCP/HTTP/Pipeline);有可支持事务的通信机制 Distributed Transactions;还有为使互操作能力最大化而设计的 Soap-based 通信机制 XML Web Services;也有可以与老式遗留系统 (COM/COM+/MTS 等) 异步通信的机制 Message Queues。由于这些各种各样的通信机制设计方法不同,而且彼此间也有重叠性,对于开发者来说,不同的选择须要学习掌握不同的程序设计模型,非常不方便。另外 SOA (Service-Oriented Architecture) 也开始盛行,随着技术的推进,微软重新审视了.NET 中的这些通信机制,设计实现了一个统一化的通信编程开发模型,这就是 WCF,它为.NET 平台上的数据通信提供了最基本最富有弹性和一致性的基础设施。

WCF 是按照 SOA 的架构原则设计的分布式计算基础环境,在其上开发人员可以实现各种 Services 提供给客户端消费者来调用。WCF 提供了各种各样的基础结构来最大程度地为开发人员提供方便性和灵活性,快速有效地构建 Web Service 应用。

WCF 的魅力除了来自于它为开发人员所提供的模型和基础外,还来自于它自身的设计和实现。记得,第一次了解 WinFx 中 Indigo 的使命时,我就对它充满了期待,同时也有几分悬疑;当2002年得知 Don Box 加入微软负责 Indigo 的架构时,悬疑彻底被转化,成为更多的期待,我更加关注它每个版本的成长,到2006年 WCF 随.NET 3.0发布时,呈现在我们面前的 WCF 堪称经典。其中恰到好处的抽象、分层和模块关系,简妙高效的运行时模型与框架机理,以及各部分丰满实在的内在实现,还有它相当彻底的可扩展性设计,都是我们学习架构设计的绝佳范本。可以说,WCF 是微软 SOA 思维的第一轮近乎完美的演绎。自 WCF 发布以来,它迅速成为微软技术体系下企业级应用开发的首选平台,时间证明了它在微软 SOA 战略中的基石地位。

国内 WCF 的应用正稳步成长中,其中一些 WCF 技术先行者对广大社区的带动和推进作用

是不可忽略的。本书的作者蒋金楠 (Artech) 就是其中走在前面的一位。他在博客园中关于 WCF 的博文成为许多 WCF 开发人员的第一手学习资料。

这里还要说一个小故事。去年, 和 TerryLee 还有武汉博文的编辑朋友一起小聚, 其中谈到技术写作, 我向两位谈了自己关于“WCF 三部曲 (应用实践篇、技术内幕分析篇和扩展篇)”的写作计划, 编辑朋友告诉我已经有作者在 WCF 上先期开始了, 我马上就猜到了是 Artech。随后通过 TerryLee 联络得知, Artech 的书稿已经完成近半了。在拿到了 Artech 新书的目录稿后, 基于我对 Artech 博文写作的判断, 我大胆做了决定, 暂时搁置自己 WCF 首篇的写作计划, 转而期待 Artech 新书的完稿。

直到最近看了 Artech 的整部书稿, 通篇读后的感觉, 书如己出, 不亚于当年得知 Don Box 去了微软负责 Indigo 架构的消息时的感觉, 事实证明, 我当时搁置写作计划的决定是正确的。

知识全面、论述准确、逻辑严密是本书的写作特点, 这是一本各个层次开发人员都可以从中受益的书: 对于 WCF 的初、中级开发人员, 这本书可以帮助你获得 WCF 全方位的知识, 全面系统地梳理关于 WCF 的知识结构, 提升动手实施能力; 对于 WCF 的高级开发人员, 这本书既可以有效弥补你 WCF 相关知识结构中的盲点, 又可以让你在自己熟悉的知识点上领略作者的看法和理解。

当然, 读完本书, 也不是完全没有疑惑, 我最大的疑惑就是含有“卷1”字样的书名, 难道这又是一个暗示? 暗示我又要考虑搁置我关于 WCF 三部曲后续两篇的写作计划? 抑或是可能的合作机会? 我当然希望是后者, 不过无论是哪一个, 显然都是很值得我期待的!

资深架构师 曲春雨

2009年6月于北京

## 推荐序二

随着核心 Web 服务标准 (SOAP 和 WSDL) 逐渐被广泛采纳和应用, 高度异构的软件系统之间的互操作性取得了前所未有的进步, 同时它也在安全性、事务性、可靠性方面提出了新的要求, 以至于后来又推出了大量的 Web 服务补充标准。在 .NET 平台下做过分布式开发的朋友, 想必对以下技术都不会陌生: ASP.NET 服务、Web 服务增强、.NET Remoting、MSMQ 等, 这些技术各自独立, 编程模型差别较大, 无法用一种统一的编程模型进行分布式应用程序开发。

在2003年时, 微软启动了一个代码名为 Indigo 的项目, 微软试图实现一个宏伟的计划, 用一套统一的 API 完成上述各模型的功能, 同时支持良好的扩展性, 为了出现新的 Web 服务标准、协议时, 无须再开发另外一套模型, 这听起来有些不可思议, 但微软却做到了, 2005

年 Indigo 发展为一个稳定的版本，同时更名为 Windows Communication Foundation（简称 WCF），并且作为 .NET Framework 3.0 的一部分发布，.NET Framework 3.5 中，WCF 得到了进一步增强，在开发 REST 服务方面也提供了支持。可以说 WCF 是一个统一的、可配置、可扩展的分布式应用程序开发框架，使用它可以非常轻松高效地构建分布式应用程序。

目前 WCF 技术已经得到了广泛的应用，但国内在这方面的资料却非常少，据我所知，迄今为止还没有一本 WCF 的原创中文书籍，蒋金楠（Artech）的《WCF 技术剖析（卷1）》是国内第一本。非常荣幸，我能在第一时间阅读本书书稿，书中内容涵盖了 WCF 技术的方方面面，有知识点讲解，也有案例演示，从使用的角度阐释了 WCF。

Artech 在 WCF 方面的造诣和写作功底不容置疑，这一点从他在个人博客上发表的 WCF 技术文章就可以看得出来，他的“WCF 之旅”和“WCF 后续之旅”两个系列深受广大读者好评。在读完本书后，只想对 Artech 说，上市后多送我几本吧，我要把它送给身边的朋友。

软件架构师 李会军

2009年6月于北京

### 推荐序三

15年前，当我还在用 C++ 和汇编语言在 Win 3.1 平台上写桌面程序的时候，我考虑的内容可以说很具体，也很全面。说具体是因为我要关心存储、提取、进程间互斥、界面展现等众多技术实现内容；说全面是因为我大概要关心这些应用几乎全部的开发内容。

现在，时代变了。计算已经从桌面搬移到我们视野之外，计算可能发生在太平洋海底某个探测传感器上、太空中北斗卫星阵上、家里的洗衣机里、Windows Azure 的 SQL Services……而我们除了专注于自己的开发工作外，也要多多“借助他山之石”，根据需要把其他计算系统的资源“粘”到我们自己的成果上，并对外提供必要的服务和支持。

以前，为完成这些我们不得不面对众多技术，而多种“方言”往往会造成工作成果无法重用。WCF 的出现使得全球的 .NET 开发者可以用一种，而且是唯一的一种“语言”很好地把各种计算系统“黏合”在一起，同时成果也可以被操着各种“方言”的用户分享，无论他信仰 COM+、.NET、Java、REST 还是其他。

感谢蒋先生的辛勤工作，《WCF 技术剖析》为我们解读了 WCF 这个被国内外同行普遍看好的技术。不过，由于 WCF 框架自身的延展性和灵活性，系统学习 WCF 本身也是一个较为艰苦的过程，蒋先生撰写的 WCF 专著的第1卷相信会成为您学习 WCF 入门并实际完成大部分项目的良好基础。

祝您顺利地用 WCF 跃上更为广阔的分布式计算平台。

博客园资深技术专家 王翔

2009年5月于北京

## 第5章 序列化与数据契约

(Serialization and Data Contract)

大部分的系统都是以数据为中心的 (Data Central)，功能的实现表现在对相关数据的正确处理方面。而数据本身，是有效信息的载体，在不同的环境具有不同的表示。一个分布式的互联系统关注于数据的交换，而数据正常交换的根本前提是参与数据交换的双方对于数据结构的一致性理解。这就为数据的表现提出了要求，为了保证处于不同平台、不同厂商的应用能够正常地进行数据交换，交换的数据必须采用一种大家都能够理解的展现方式。在这方面，XML 无疑是最好的选择。所以 WCF 下的序列化 (Serialization) 解决的就是如何将数据从对象的表现形式转变成 XML 表现形式，以确保数据的正常交换。

WCF 主要通过数据契约 (Data Contract) 来定义通过服务操作进行交换的数据，在本章我们将着重介绍基于数据契约的序列化。

### 5.1 漫谈序列化

对于一个托管应用程序，在运行时，数据是通过基于某种类型的对象承载的。存在于内存中的托管对象隶属于它所在的应用程序域 (Application Domain)，是不能直接跨域传递的，更不用说是基于跨机器、甚至是跨网络的传递了。要实现这种跨应用程序域、跨机器甚至是跨网络的数据传递，序列化是必须的。此外，持久化也是序列化应用的一个典型例子，对象只有被序列化成文本或二进制才能被保存于物理介质中。由于序列化解决的是相同的数据在不同表现形式之间进行转化的问题，所以序列化依赖于数据具体的结构描述。对于最为典型的基于 XML 的序列化，可以通过专门的 XML 序列化器对其进行有效的序列化。

#### 5.1.1 封送 (Marshaling) 与序列化

在原有的基于进程的程序隔离机制基础上，.NET 采用了更为细粒度的隔离级别，即基于应用程序域 (Application Domain) 的隔离。一个托管的程序运行于独自的应用程序域中，一个进程中可以同时运行一个或多个应用程序域。处于相同进程中的各个应用程序域是相互独立的，从而确保一个程序的崩溃不至于对另一个应用程序或整个进程造成影响。

应用程序域的隔离表现在基于内存的隔离上。在默认情况下，CLR 加载程序集 (Assembly) 到当前的应用程序域中，被加载的程序集被当前的应用程序域独占使用，并不能被其他应用程序域共享。而对于在一个应用程序域中创建的托管对象，由于其内存空间被当前应用程序域独占使用，是不能直接被另一个应用程序域访问的。

注：关于 Assembly 的加载，可以采用两种不同的方式，一种以独占的形式加载到相应的 Application Domain 中；另一种则以共享的形式加载到 SharedDomain，供所有的 AppDomain 共享，比如 MScorLib.dll 就是以这样的方式加载的。



要解决这种跨应用程序域对象访问的问题，须要采用一种特别的机制：封送（Marshaling）。具体来讲，有两个不同的封送机制，一种是按引用封送（Marshal By Reference），另一种则是按值封送（Marshal By Value）。

按引用封送的机制是在调用端的应用程序域上创建代理对象，通过代理间接地实现跨应用程序域的对象调用。代理包含对于目标对象的引用，在跨域调用的时候，调用的请求通过代理对象传递到目标对象所在的应用程序域，并调用目标对象。.NET Remoting 完全就是采用的这样的实现机制。

而在按值封送下，则是将对象进行序列化，将序列化的 XML 或二进制流传递到另一个应用程序域中，并通过反序列化（Deserialization）重建该对象。通过反序列化重建的对象和原来的对象没有任何关系，如果调用该对象，则方法的执行在当前应用程序域中进行。

### 5.1.2 持久化（Persisting）与序列化

如果按照存储介质和生命周期的长短划分，所有的数据都以两种形式存在，其中一种是保存于内存中的运行时对象，另一种则是存储于持久化物理介质中的文件，比如数据库文件等。数据的持久化关注于相同的数据在不同形态数据之间的转化，解决的是如何将内存对象持久化存储，以及从物理介质中加载数据并创建内存对象。

数据的持久化是序列化的又一个典型的应用，对象只有在序列化之后才能进行持久化存储，从持久化存储介质加载的数据通过反序列化转变成运行时对象。

### 5.1.3 数据结构与序列化

数据是有效信息的载体，相同的信息可以通过不同形态的数据表达。数据对信息有效表达的前提是数据本身必须是结构化的，也就是说数据元素需要通过一定的结构进行组织。对于数据的使用者来说，只有在明确了数据结构的前提下，才能确定数据的每一个元素所表达的含义。对于一个无序的、杂乱无章的数据，就好像无法破译的外星人密码，我们无法获知数据所承载的信息，数据本身也就失去了其原有的价值。

序列化关注于如何将运行时对象转换成另一种能够被跨域传递或持久化存储的形态，比如基于纯文本的 XML 或二进制码等，所以序列化（或反序列化）的本质是解决数据在不同形态之间转换的问题。

而不同表现形式的数据，都具有相应的数据结构描述方式。对于.NET 程序运行时的托管对象，采用怎样的结构描述方式呢？答案是对象的类型。我们知道，程序集的元数据表（Metadata Table）描述了组成程序集的物理与逻辑结构，记录了定义在当前程序集中的所有类型和类型的成员。在托管的世界里，每一个托管对象都对应一个确定的类型，组成对象的所有数据成员的结构，都通过类型进行描述。所以，无论是序列化还是反序列化，都必须预先确定被序列化对象或反序列化生成对象的类型。

WCF 的序列化和反序列化解决的是托管对象与 XML 之间的转换，以实现与厂商无关、跨平台的数据交换。之所以基于 XML 的数据表示能够实现跨平台的数据交换，在于 XML 是一种被不同的厂商普遍接受和易于理解的数据表示。WCF 以 XML 的形式进行数据交换，一方传输的 XML 能够被另一方正确理解，同样需要交换双方就传递的 XML 结构达成一致的理解。具体采用怎样地实现描述 XML 的结构呢？XSD 无疑是最好的选择。在 WCF 中，服务契约借助 WSDL 的形式实现对服务的描述，同理，数据契约通过 XSD 实现对数据的描述。

#### 5.1.4 XML 序列化器（1）

在 WCF 之前，.NET Framework 就对基于 XML 的序列化提供了原生支持，比如在 .NET Remoting 中，通过 SoapFormatter 和 BinaryFormatter 分别实现了基于 SOAP 和二进制的序列化和反序列化。ASP.NET 则采用基于 XmlSerializer 的序列化方式。接下来，我将通过一个简单的例子介绍如何通过 XmlSerializer 进行序列化和反序列化，以及 XmlSerializer 在进行序列化和反序列化时采用怎样的规则。

XmlSerializer 默认的序列化规则

假设我们需要通过 XmlSerializer 对一个 Order 对象进行序列化,Order 类型定义如下。

```
1 namespace Artech.XmlSerializerDemos
2 {
3     public class Order
4     {
5         private double _totalPrice;
6
7         public Guid ID
8         { get; set; }
9
10        public DateTime Date
11        { get; set; }
12
13        public string Customer
14        { get; set; }
15
16        public string ShipAddress
17        {get;set;}
18
19        public Order()
20        { }
21
22        public Order(double totalPrice)
23        {
24            this._totalPrice = totalPrice;
25        }
```

```

26     }
27 }

```

为了方便，定义下面一个泛型方法专门进行基于 XmlSerializer 的序列化操作，并通过启动进程的方式打开保存序列化 XML 的文件。

```

28 static void Serialize<T>(T instance, string fileName)
29 {
30     using (XmlWriter writer = new XmlTextWriter
31         (fileName, Encoding.UTF8))
32     {
33         XmlSerializer serializer = new XmlSerializer(typeof(T));
34         serializer.Serialize(writer, instance);
35         Process.Start(fileName);
36     }

```

在下面的代码中，我们创建一个 Order 对象，并设定相应的属性，然后通过上面的方法将该对象序列化到一个 XML 物理文件中。下面列出了最终生成的 XML 的内容。

```

37 namespace Artech.XmlSerializerDemos
38 {
39     class Program
40     {
41         static void Main(string[] args)
42         {
43             Order order = new Order(8888)
44             {
45                 ID = Guid.NewGuid(),
46                 Date = DateTime.Today,
47                 Customer = "Foo",
48                 ShipAddress = "#328, Airport Rd,
49                 Industrial Park, Suzhou Jiangsu
50                 Province"
51             };
52             Serialize<Order>(order, @"E:\Order.xml");
53         }
54     }
55     <?xml version="1.0" encoding="utf-8"?>
56     <Order xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
57         xmlns:xsd="http://www.w3.org/2001/XMLSchema">
58         <ID>85954dfe-4f9d-422f-9fb0-4be4f63f9cc0</ID>
59         <Date>2008-12-02T00:00:00+08:00</Date>
60         <Customer>Foo</Customer>

```

```

61     <ShipAddress>#328, Airport Rd, Industrial Park, Suzhou Jiangsu
62         Province</ShipAddress>
63 </Order>

```

通过 Order 类型的定义和被序列化的 XML 结构的对比，我们可以看出 XmlSerializer 进行序列化在默认情况下采用的规则：

XML 根节点的名称为对象类型的名称，并且没有命名空间；

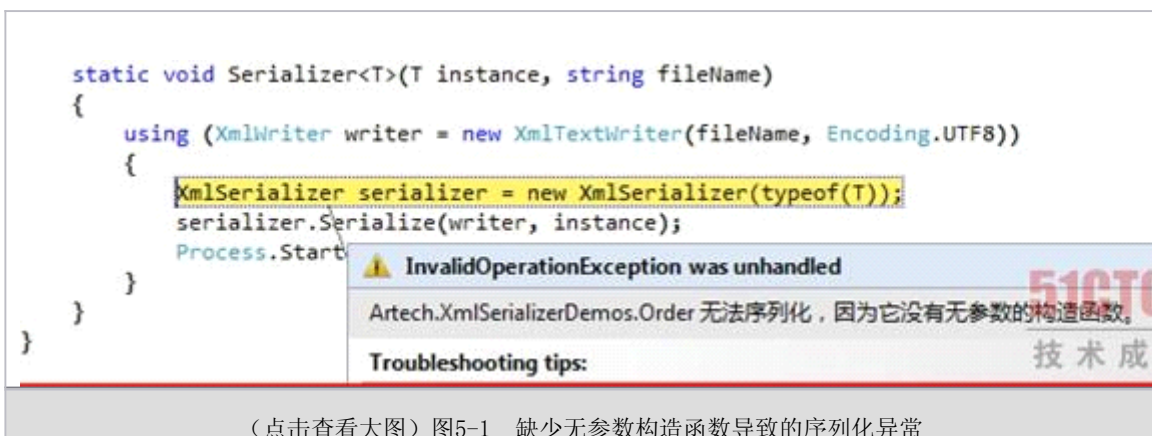
对象属性或字段成员以 XmlElement 的形式输出，名称和属性或字段名称一致，并且不具有命名空间；

只有 public 属性或字段成员才会参与序列化(private 的 \_totalPrice 字段的值并不会被序列化)；

XmlElement 的顺序与对应的属性或字段在类型中定义的顺序一致。

#### 5.1.4 XML 序列化器 (2)

基于 XmlSerializer 的序列化和反序列化需要把类型定义为默认无参的构造函数，这是因为 XmlSerializer 在进行反序列化的时候要通过调用它来创建对象。即使在序列化的时候不会调用此默认构造函数，仍然需要类型有此定义。如果我们将 Order 类型的无参构造函数去掉，运行上面的代码将会抛出如图5-1所示的 InvalidOperationException 异常。



此外，XmlSerializer 还具有一些额外的序列化规则。必须是公有 (Public)、可读、可写的属性才能通过序列输出到 XML 中。为了验证此序列化规则，我对 Order 类型进行了一些修正，将 ID 和 Date 属性分别改成了只读和只写的属性。

```

1 namespace Artech.XmlSerializerDemos
2 {
3     public class Order
4     {
5         private double _totalPrice;
6
7         private Guid _id;

```

```

8         public Guid ID
9         {
10             get { return _id; }
11         }
12
13         private DateTime _date;
14         public DateTime Date
15         {
16             set { _date = value; }
17         }
18
19         public string Customer
20         { get; set; }
21
22         public string ShipAddress
23         {get;set;}
24
25         public Order()
26         { }
27
28         public Order(double totalPrice, Guid id)
29         {
30             this._totalPrice = totalPrice;
31             this._id = id;
32         }
33     }
34 }

```

通过下面的代码调用 XmlSerializer 对创建的 Order 对象序列化后,最终的 XML 将会如下所示。我们会发现属性 ID (read-only) 和 Date (Write only) 的值并没有出现在最终的 XML 中。

```

35 Order order = new Order(8888, Guid.NewGuid())
36 {
37     Date = DateTime.Today,
38     Customer = "Foo",
39     ShipAddress = "#328, Airport Rd,
Industrial Park, Suzhou Jiangsu Province"
40 };
41 Serialize<Order>( order,@"E:\Order.xml");
42 <?xml version="1.0" encoding="utf-8"?>
43 <Order xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
44     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
45     <Customer>Foo</Customer>
46     <ShipAddress>#328, Airport Rd, Industrial Park, Suzhou Jiangsu

```

```
47         Province</ShipAddress>
48     </Order>
```

### 自定义 **XmlSerializer** 的序列化规则

上面列出的仅仅是 XmlSerializer 采用的默认序列化规则，在很多情况下，我们希望手工控制最终生成的 XML 结构（Schema）。在这种情况下，需要依赖一些特殊自定义的特性：XmlRootAttribute、XmlAttributeAttribute、XmlElementAttribute 等，它们均定义在 System.Xml.Serialization 命名空间下。针对上面列出的默认规则，合理地使用这些特性可以有效地控制最终 XML 的结构。

可以在类型上面应用 XmlRootAttribute 特性，通过 Name 和 Namespace 属性改变根节点的名称和命名空间；

可以在字段或属性上面应用 XmlAttributeAttribute 特性，使之以 XML 属性的形式而不是以 XML 元素的形式输出，同时通过 Name 和 Namespace 属性指定 XML 属性的名称和命名空间；

以 XML 元素形式输出的字段或属性，也可以通过 XmlElementAttribute 特性的 Name 和 Namespace 属性设置其元素名称和命名空间；

XML 元素的先后次序可以通过 XmlElementAttribute 的 Order 属性控制。

在下面的代码中，我们通过 XmlRootAttribute、XmlAttributeAttribute、XmlElementAttribute 对 Order 类型进行了重新定义。通过 XmlRootAttribute 将根节点的名称和命名空间设置成 "Ord" 和 "<http://www.artech.com>"；通过 XmlAttributeAttribute 将 ID 属性以 XML 属性的形式输出，并设置了属性名称和命名空间 ("OrderID"和"urn:artech")；通过 XmlElementAttribute 对 XML 元素重新排序：Customer、Date 和 ShipAddress。最终序列化后的 XML 反映了希望的输出格式。

```
49 namespace Artech.XmlSerializerDemos
50 {
51     [XmlRoot("Ord", Namespace="http://www.artech.com")]
52     public class Order
53     {
54         private double _totalPrice;
55
56         [XmlAttribute(AttributeName = "OrderID", Namespace =
"urn:artech")]
57         public Guid ID
58         { get; set; }
59
60         [XmlElement(ElementName = "OrderDate",Order = 2, Namespace=
"urn:artech")]
```

```

62         public DateTime Date
63         { get; set; }
64
65         [XmlElement(Order = 1)]
66         public string Customer
67         { get; set; }
68
69         [XmlElement(Order = 3)]
70         public string ShipAddress
71         { get; set; }
72
73         public Order()
74         { }
75
76         public Order(double totalPrice)
77         {
78             this._totalPrice = totalPrice;
79         }
80     }
81 }
82 <?xml version="1.0" encoding="utf-8"?>
83 <Ord xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
84     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
85     d1p1:OrderID="fa06d7a1-e514-4b9e-8ce3-ffe699f4d232"
86     xmlns:d1p1="urn:artech" xmlns="http://www.artech.com">
87     <Customer>Foo</Customer>
88     <d1p1:OrderDate>2008-12-03T00:00:00+08:00</d1p1:OrderDate>
89     <ShipAddress>#328, Airport Rd, Industrial Park, Suzhou Jiangsu
90         Province</ShipAddress>
91 </Ord>

```

## 5.2 数据契约与数据契约序列化器

一个正常的服务调用要求客户端和服务端对服务操作有一致的理解，WCF 通过服务契约对服务操作进行抽象，以一种与平台无关的，能够被不同的厂商理解的方式对服务进行描述。同理，客户端和服务端进行有效的数据交换，同样要求交换双方对交换数据的结构达成共识，WCF 通过数据契约来对交换的数据进行描述。与数据契约的定义相匹配，WCF 采用新的序列化器——数据契约序列化器（DataContractSerializer）进行基于数据契约的序列化与反序列化操作。

### 5.2.1 数据契约的本质

“契约”就是双方或多方就某个问题达成的一种共识。数据契约作为 WCF 4大契约（服务契约、数据契约、消息契约和错误契约）之一，是对数据结构的描述。站在不同的角度，我们可以对数据契约有不同的理解。

数据契约以一种与平台无关的形式对数据结构进行描述

抛开任何技术层面的东西，抛开 WCF 相关的技术实现，数据契约可以看成是面向服务（Service Orientation）的范畴。谈到 SOA，很多人自然而然地会联想到 Web Service、WCF，而事实上，SOA 本身是和技术无关的。Web Service 也好，WCF 也罢，都是实现 SOA 的一种技术手段而已。并不是说实现 SOA 必须通过 Web Service、WCF 这样的技术手段，尽管到目前为止它们是最好的选择。也不是说你采用了 Web Service 或 WCF，开发出来的东西就是基于 SOA 的。

和其他的软件构建模式一样，面向服务的一个重要的目标和特征是实现服务的重用性和松耦合。而面向服务的重用已经达到了前所未有的高度，并不仅仅局限于同一个应用不同模块之间的相互调用，以及同一个局域网内不同异构系统之间的互相访问，我们完全可以实现基于 Internet 的服务访问。现在很火的“云计算”甚至将所有的功能实现在相应的服务中，将服务放入“云端”，供所有的潜在消费者调用。

重用性与耦合性可以看成是同一个问题的不同表现，它们之间是相辅相成的。重用的程度往往决定于交互双方耦合的程度：如果交互双方的耦合度过高，依赖性过强，必然影响其重用性。松耦合设计的根本途径就是提取影响交互所必须的依赖，去除与交互无关的依赖。而契约所描述的就是保证两者正常交互所必须的依赖。

对于传统的面向组件设计，契约通过什么来体现呢？答案是类型，显而易见，如果须要调用定义在某个组件的某个方法，你需要在本地具有该方法涉及的所有类型的引用。而反映在部署上面，你需要在本地具有定义了这些类型的 Dll。

很显然，这种基于类型系统的契约已经无法适应面向服务的要求。首先类型是基于某种特定的技术平台的，.NET 平台和 J2EE 平台具有不同的类型定义。其次这种基于 Dll 的部署方式也是不能被接受的，你不可能将程序集部署到所有需要访问服务的机器上。所以面向服务对契约的表示提出了新的要求，首先对契约的描述必须是厂商中立、跨平台的，其次契约能够通过简单的方式进行发布和获取。基于这两点，使用 XML 的方式进行契约的描述无疑是最好的选择。

数据契约旨在描述数据的结构，数据交换只有在双方对数据结构具有相同的理解下才能正常进行。对于数据的接收方来讲，当它接收到数据时，只有借助于数据结构的描述，才能理解数据的每个元素所承载的信息。由于数据交换通过 XML 表示，XSD 用于定义 XML 的结构（Schema），所以 XSD 可以看作是数据契约在 SOA 下的表现形式。



## WCF 提供 CLR 类型和数据契约的适配

WCF 面对了两个世界，一个是 XML 的世界，另一个是托管类型的世界。数据和服务最终实现在一个个具体的托管类型中，而 WCF 采用基于 XML 的消息通信方式。所以 WCF 架构是连接两个世界的纽带。在托管的世界里，数据契约通过一个具体的类型表示，数据成员对应于类型中相应的字段或属性成员。WCF 最终需要将其转换成基于 XML 的表现形式，并通过元数据的形式发布出来，所以 WCF 在这里起到了一个适配的作用。

### 5.2.2 数据契约的定义与数据契约序列化器（1）

同服务契约类似，WCF 采用了基于特性（Attribute）的数据契约定义方式。基于数据契约的自定义特性主要包含以下两个：DataContractAttribute 和 DataMemberAttribute，接下来我们将讨论这两个重要的自定义特性。

#### DataContractAttribute 和 DataMemberAttribute

WCF 通过应用 DataContractAttribute 特性将其目标类型定义成一个数据契约，下面是 DataContractAttribute 的定义。从 AttributeUsage 的定义来看，DataContractAttribute 只能用于枚举、类和结构体，而不能用于接口；DataContractAttribute 是不可以被继承的，也就是说当一个类型继承了一个应用了 DataContractAttribute 特性类型，自身也只有显式地应用 DataContractAttribute 特性才能成为数据契约；一个类型上只能应用唯一的一个 DataContractAttribute 特性。

```
1  [AttributeUsage(AttributeTargets.Enum |
2  AttributeTargets.Struct |
3  AttributeTargets.Class, Inherited = false, AllowMultiple = false)]
4  public sealed class DataContractAttribute : Attribute
5  {
6      public bool IsReference { get; set; }
7      public string Name { get; set; }
8      public string Namespace { get; set; }
9  }
```

DataContractAttribute 仅仅包含3个属性成员。其中 Name 和 Namespace 表示数据契约的名称和命名空间；IsReference 表示在进行序列化的时候是否保持对象现有的引用结构。比如说，一个对象的两个属性同时引用一个对象，那么有两种序列化方式，一种是在序列化后的 XML 仍然保留这种引用结构，另一种是将两个属性的值序列化成两份独立的具有相同内容的 XML。

对于服务契约来说，我们将在一个接口或类上面应用的 ServiceContractAttribute 定义成服务契约后，并不意味着该接口或类中的每一个方法成员都是服务操作，而是通过

OperationContractAttribute 显式地将相应的方法定义成服务操作。与之类似，数据契约也采用这种显式声明的机制。对于应用了 DataContractAttribute 特性的类型，只有应用了 DataMemberAttribute 特性的字段或属性成员才能成为数据契约的数据成员。DataMemberAttribute 特性的定义如下所示。

```
9  [AttributeUsage(AttributeTargets.Field | AttributeTargets.Property,  
10     Inherited = false, AllowMultiple = false)]  
11  public sealed class DataMemberAttribute : Attribute  
12  {  
13      public DataMemberAttribute();  
14  
15      public bool  
16  EmitDefaultValue { get; set; }  
17      public bool  
18  IsRequired { get; set; }  
19      public string  
20  Name { get; set; }  
21      public int  
22  Order { get; set; }  
23  }
```

下面的列表列出了 DataMemberAttribute 的4个属性所表述的含义。

**Name:** 数据成员的名称，默认为字段或属性的名称。

**Order:** 相应的数据成员在最终序列化的 XML 中出现的位置，Order 值越小越靠前，默认值为 1。

**IsRequired:** 表明属性成员是否是必须的成员，默认值为 false，表明该成员是默认的。

**EmitDefaultValue:** 表明在数据成员的值等于默认值的情况下，是否还须要将其序列化到最终的 XML 中，默认值为 true，表示默认值会参与序列化。

**注：** 数据契约和数据成员只和是否应用了 DataContractAttribute 和 DataMemberAttribute 有关，与类型和成员的存取限制修饰符（public、internal、protected、private 等）无关。也就是说，应用了 DataMemberAttribute 的私有字段或属性成员也是数据契约的数据成员。

数据契约序列化器（DataContractSerializer）

在 WCF 中，数据契约的定义是为序列化和反序列化服务的。WCF 采用数据契约序列化器（DataContractSerializer）作为默认的序列化器。在上面一节，介绍了 XmlSerializer 以及基于 XmlSerializer 的序列化规则，现在以相同的方式来介绍

DataContractSerializer 和基于 DataContractSerializer 采用的序列化规则。先来看看 DataContractSerializer 的定义。

```
24 public sealed class DataContractSerializer :  
    XmlObjectSerializer  
25 {  
26     //其他成员  
27     public DataContractSerializer(Type type);  
28     //其他构造函数  
29  
30     public override object ReadObject(XmlReader reader);  
31     public override object ReadObject(XmlDictionaryReader  
    reader, bool  
32         verifyObjectName);  
33     public override object ReadObject(XmlReader reader, bool  
34         verifyObjectName);  
35     public override void WriteObject(XmlWriter writer,  
    object graph);  
36  
37     public IDataContractSurrogate DataContractSurrogate { get; }  
38     public bool IgnoreExtensionDataObject { get; }  
39     public ReadOnlyCollection<Type> KnownTypes { get; }  
40     public int MaxItemsInObjectGraph { get; }  
41     public bool PreserveObjectReferences { get; }  
42 }
```

DataContractSerializer 定义了一系列的重载的构造函数，我们可以调用它们构建相应的 DataContractSerializer 对象，通过制定相应的参数控制系列化器的序列化和反序列化行为。在后续的介绍中我们会通过这些相应的构造函数创建 DataContractSerializer 对象，在这里就不一一介绍了。DataContractSerializer 主要通过两个方法进行序列化和反序列化：WriteObject 和 ReadObject。这里需要着重介绍一下 DataContractSerializer 的5个重要属性成员。

**DataContractSurrogate:** 这是一个实现了 IDataContractSurrogate 接口的数据契约代理类的对象。契约代理会参与到 DataContractSerializer 的序列化、反序列化，以及契约的导入和导出的过程中，实现对象和类型的替换。

**IgnoreExtensionDataObject:** 扩展数据对象（ExtensionDataObject）旨在解决双方数据契约不一致的情况下，在数据传送-回传（Round Trip）过程中造成的数据丢失。

**KnownTypes:** 由于序列化和反序列化依赖于定义在类型的元数据信息，所以在进行序列化或反序列化之前，需要确定被序列化对象，或者反序列化生成对象的所有相关的真实类型。为了确保序列化或反序列化的成功，须要将相关的类型添加到 KnownTypes 类型集合中。

**MaxItemsInObjectGraph:** 为了避免黑客生成较大数据，频繁地访问服务造成服务器不堪重负（我们一般把这种黑客行为称为拒绝服务 DoS-Denial of Service），可以通过 MaxItemsInObjectGraph 属性设置进行序列化和反序列化允许的最大对象数。MaxItemsInObjectGraph 的默认值为65536。

**PreserveObjectReferences:** 这个属性与 DataContractAttribute 的 IsReference 属性的含义一样，表示的是如果数据对象的多个属性或字段引用相同的对象，在序列化的时候是否需要在 XML 中保持一样的引用结构。

### 5.2.2 数据契约的定义与数据契约序列化器（2）

#### 基于 DataContractSerializer 的序列化规则

与在5.1节介绍 XmlSerializer 的序列化规则一样，现在我们通过一个具体的例子来介绍 DataContractSerializer 是如何进行序列化的，以及采用怎样的序列化规则。照例定义一个泛型的辅助方法进行专门的序列化工作，最终生成的 XML 保存到一个 XML 文件中。

```
1 public static void Serialize<T>(T instance, string fileName)
2 {
3     DataContractSerializer serializer = new
DataContractSerializer(typeof(T));
4     using (XmlWriter writer = new
XmlTextWriter(fileName, Encoding.UTF8))
5     {
6         serializer.WriteObject(writer, instance);
7     }
8     Process.Start(fileName);
9 }
```

我们需要对一个 Order 对象进行序列化，Order 类型的定义如下。实际上定义了两个数据契约：OrderBase 和 Order，Order 继承于 OrderBase。

```
10 namespace Artech.DataContractSerializerDemos
11 {
12     [DataContract]
13     public class OrderBase
14     {
15         [DataMember]
16         public Guid ID
17         { get; set; }
18
19         [DataMember]
20         public DateTime Date
```

```

21         { get; set; }
22
23         [DataMember]
24         public string Customer
25         { get; set; }
26
27         [DataMember]
28         public string ShipAddress
29         { get; set; }
30
31         public double TotalPrice
32         { get; set; }
33     }
34
35     [DataContract]
36     public class Order : OrderBase
37     {
38         [DataMember]
39         public string PaymentType
40         { get; set; }
41     }
42 }

```

通过下面的代码对创建的 Order 对象进行序列化，会生成一段 XML。

```

43 Order order = new Order()
44 {
45     ID = Guid.NewGuid(),
46     Date = DateTime.Today,
47     Customer = "NCS",
48     ShipAddress = "#328, Airport Rd, Industrial
Park, Suzhou JiangSu Province",
49     TotalPrice = 8888,
50     PaymentType = "Credit Card"
51 };
52 Serialize(order, @"E:\order.xml");
53 <Order xmlns:i="http://www.w3.org/2001/XMLSchema-
instance" xmlns="http://
54     schemas.datacontract.org/2004/07/Artech.
DataContractSerializerDemos">
55     <Customer>NCS</Customer>
56     <Date>2008-12-03T00:00:00+08:00</Date>
57     <ID>5fdbee36-e29e-48d2-b45f-6fd4beba54d6</ID>
58     <ShipAddress>#328, Airport Rd, Industrial Park,
Suzhou JiangSu

```

```

59         Province</ShipAddress>
60         <PaymentType>Credit Card</PaymentType>
61     </Order>

```

将数据契约与最终生成的 XML 结构进行对比，我们可以看出 DataContractSerializer 在默认的情况下采用了如下的序列化规则。

XML 的根节点名称为数据契约类型的名称，默认的命名空间采用这样的格式：  
<http://schemas.datacontract.org/2004/07/{数据契约类型的命名空间}>；

只有显式应用了 DataMemberAttribute 特性的字段或属性才能作为数据成员采用，才会参与序列化（比如 TotalPrice 属性的值不会出现在序列化后的 XML 中）；

所有数据成员均以 XML 元素的形式被序列化；

序列化后数据成员在 XML 的次序采用这样的规则：父类数据成员在前，子类数据成员在后；定义在同一个类型的数据成员按照字母排序。

如果默认序列化后的 XML 结构不能满足我们的要求，可以通过 DataContractAttribute 和 DataMemberAttribute 相应的属性对其进行修正。在重新定义的数据契约中，我们通过 DataContractAttribute 设置了数据契约的名称和命名空间；通过 DataMemberAttribute 的 Name 属性为 ID 和 Date 两个属性设置了不同于属性名称的数据成员名称，并通过 Order 控制了数据成员的先后次序。那么调用相同的程序，最终被序列化出来的 XML 将会如下所示。

```

62 namespace Artech.DataContractSerializerDemos
63 {
64     [DataContract(Namespace="http://www.artech.com/")]
65     public class OrderBase
66     {
67         [DataMember(Name = "OrderID", Order=1)]
68         public Guid ID
69         { get; set; }
70
71         [DataMember(Name = "OrderDate", Order = 2)]
72         public DateTime Date
73         { get; set; }
74
75         [DataMember(Order = 3)]
76         public string Customer
77         { get; set; }
78
79         [DataMember(Order = 4)]
80         public string ShipAddress
81         { get; set; }
82

```

```

83         public double TotalPrice
84         { get; set; }
85     }
86
87     [DataContract(Name="Ord", Namespace="
http://www.artech.com/")]
88     public class Order : OrderBase
89     {
90         [DataMember(Order = 1)]
91         public string PaymentType
92         { get; set; }
93     }
94 }
95 <Ord xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
96     xmlns="http://www.artech.com">
97     <OrderID>ba3bc051-6c02-41dd-9f97-ae745ac5f1dd</OrderID>
98     <OrderDate>2008-12-03T00:00:00+08:00</OrderDate>
99     <Customer>NCS</Customer>
100    <ShipAddress>#328, Airport Rd, Industrial
Park, Suzhou JiangSu
101        Province</ShipAddress>
102    <PaymentType>Credit Card</PaymentType>
103 </Ord>

```

### 5.2.2 数据契约的定义与数据契约序列化器（3）

#### 通过 MaxItemsInObjectGraph 限定序列化对象的数量

拒绝服务（DoS- Denial of Service）是一种常用的黑客攻击行为，黑客通过生成大量的数据频繁地对服务器发送请求，最终导致服务器不堪重负而崩溃。对于 WCF 的序列化或反序列化来说，数据的容量越大、成员越多、层次越深，序列化的时间就越长，耗用的资源就越多，如果黑客频繁地发送一个海量的数组过来，那么服务就会因为忙于进行反序列化的工作而没有足够的资源处理正常的请求，从而导致瘫痪。

在这种情况下，可以通过 MaxItemsInObjectGraph 这个属性设置 DataContractSerializer 允许被序列化或反序列化对象的数量上限，一旦超过设定的这个上限，序列化或反序列化的工作将会立即中止，从而在一定程度上解决了通过发送大集合数据形式的拒绝服务攻击。DataContractSerializer 中定义了以下3个重载的构造函数使我们能够设定 MaxItemsInObjectGraph 属性。

```

1 public sealed class DataContractSerializer :
  XmlObjectSerializer

```

```

2  {
3      //其他成员
4      public DataContractSerializer(Type type,
IEnumerable<Type> knownTypes,
5          int maxItemsInObjectGraph, bool
ignoreExtensionDataObject, bool
6          preserveObjectReferences, IDataContractSurrogate
dataContractSurrogate);
7
8      public DataContractSerializer(Type type, string
rootName, string
9          rootNamespace, IEnumerable<Type> knownTypes,
int maxItemsInObjectGraph,
10          bool ignoreExtensionDataObject, bool
preserveObjectReferences,
11          IDataContractSurrogate dataContractSurrogate);
12
13      public DataContractSerializer(Type type,
XmlDictionaryString rootName,
14          XmlDictionaryString rootNamespace,
IEnumerable<Type> knownTypes, int
15          maxItemsInObjectGraph, bool
ignoreExtensionDataObject, bool
16          preserveObjectReferences, IDataContractSurrogate
dataContractSurrogate);
17
18      public int MaxItemsInObjectGraph { get; }
19  }

```

那么 DataContractSerializer 在进行具体的序列化时，对象的个数如何计算呢？经过我的实验，发现采用的计算规则是这样的：对象自身算一个对象，所有成员及所有内嵌的成员都算一个对象。我们通过一个具体的例子来证实这一点，在上面定义的泛型 Serialize 方法上加另一个参数 maxItemsInObjectGraph，并调用另一个构造函数来创建 DataContractSerializer 对象。

```

20 public static void Serialize<T>(T instance,
string fileName, int
21     maxItemsInObjectGraph)
22 {
23     DataContractSerializer serializer = new
DataContractSerializer(typeof(T),
24         null,maxItemsInObjectGraph,false,false,null);
25     using (XmlWriter writer = new
XmlTextWriter(fileName, Encoding.UTF8))

```



```

26     {
27         serializer.WriteObject(writer, instance);
28     }
29     Process.Start(fileName);
30 }

```

我们现在准备调用上面的方法对一个集合对象进行序列化，为此定义了一个 OrderCollection 的类型，它直接继承了 List<Order>。

```

31 public class OrderCollection : List<Order>
32 { }
33
34 [DataContract]
35 public class Order
36 {
37     [DataMember]
38     public Guid ID
39     { get; set; }
40
41     [DataMember]
42     public DateTime Date
43     { get; set; }
44
45     [DataMember]
46     public string Customer
47     { get; set; }
48
49     [DataMember]
50     public string ShipAddress
51     { get; set; }
52 }

```

在下面的代码中，创建了 OrderCollection 对象，并添加了10个 Order 对象，如果该对象被序列化，最终被序列化对象数量是多少呢？应该这样来算，OrderCollection 对象本身算一个，每一个 Order 对象自身也算一个，Order 对象具有4个属性，各算一个，那么最终计算出来的个数是 $10 \times 5 + 1 = 51$ 个。但是在调用 Serialize 方法的时候，指定的上限仅仅是 $10 \times 5 = 50$ 。所以当调用 DataContractSerializer 的 WriteObject 方法时，会抛出如图5-2所示的 SerializationException 异常。如果 maxItemsInObjectGraph 设为51则一切正常。

```

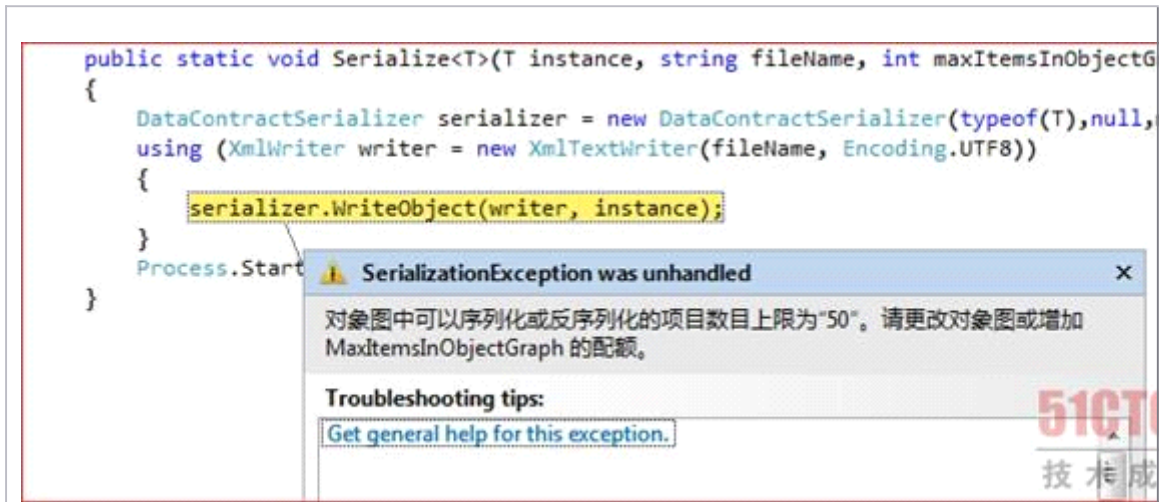
53 OrderCollection orders = new OrderCollection();
54 for (int i = 0; i < 10; i++)
55 {
56     Order order = new Order()
57     {

```

```

58         ID = Guid.NewGuid(),
59         Date = DateTime.Today,
60         Customer = "NCS",
61         ShipAddress = "#328, Airport Rd,
Industrial Park, Suzhou JiangSu
62         Province",
63     };
64     orders.Add(order);
65 }
66 Serialize(orders, @"E:\order.xml", 10*5);

```



### 5.2.2 数据契约的定义与数据契约序列化器（4）

在 WCF 应用中，MaxItemsInObjectGraph 的值可以通过 ServiceBehaviorAttribute 进行设置，在下面的代码中，将 OrderService 的 MaxItemsInObjectGraph 设为 51。

```

1  [ServiceBehavior(MaxItemsInObjectGraph = 51)]
2  public class OrderService : IOrder
3  {
4      public void ProcessOrder(OrderCollection orders)
5      {
6          //省略实现
7      }
8  }

```

MaxItemsInObjectGraph 也可以通过配置方式进行设置，MaxItemsInObjectGraph 通过 serviceBehavior 的 dataContractSerializer 配置项进行设置。

```

9  <?xml version="1.0" encoding="utf-8" ?>
10 <configuration>
11   <system.serviceModel>
12     <behaviors>

```

```

13     <serviceBehaviors>
14         <behavior name="serializationLimitationBehavior">
15             <dataContractSerializer maxItemsInObjectGraph="51" />
16         </behavior>
17     </serviceBehaviors>
18 </behaviors>
19 <services>
20     <service behaviorConfiguration="
serializationLimitationBehavior"
21         name="OrderService">
22         <endpoint address="http://
127.0.0.1:9999/orderservice" binding=
23             "basicHttpBinding" bindingConfiguration="" contract="IOrder" />
24     </service>
25 </services>
26 </system.serviceModel>
27 </configuration>

```

### 如何保持对象现有的引用结构

数据类型有值类型和引用类型之分，那么对于一个数据契约类型对象，如果多个数据成员同时引用同一个对象，那应该采用怎样的序列化规则呢？是保留现有的引用结构呢，还是将它们序列化成具有相同内容的 XML 片断呢？DataContractSerializer 的这种特性通过只读属性 `PreserveObjectReferences` 表示，默认值为 `false`，所以在默认的情况下采用的是后一种序列化方式。DataContractSerializer 定义了以下3个重载的构造函数，供我们显式地指定该参数。

```

28 public sealed class DataContractSerializer : XmlObjectSerializer
29 {
30     //其他成员
31     public DataContractSerializer(Type type,
IEnumerable<Type> knownTypes,
32         int maxItemsInObjectGraph, bool
ignoreExtensionDataObject, bool
33         preserveObjectReferences, IDataContractSurrogate
dataContractSurrogate);
34
35     public DataContractSerializer(Type type, string
rootName, string
36         rootNamespace, IEnumerable<Type> knownTypes,
int maxItemsInObjectGraph,
37         bool ignoreExtensionDataObject, bool
preserveObjectReferences,
38         IDataContractSurrogate dataContractSurrogate);

```

```

39
40     public DataContractSerializer(Type type,
XmlAttributeString rootName,
41         XmlAttributeString rootNamespace,
IEnumerable<Type> knownTypes, int
42         maxItemsInObjectGraph, bool
ignoreExtensionDataObject, bool
43         preserveObjectReferences, IDataContractSurrogate
DataContractSurrogate);
44
45     public bool PreserveObjectReferences { get; }
46 }

```

我们通过下面一个简单的例子来看看对于一个数据契约对象，在保留对象引用和不保留引用的情况下，序列化出来的 XML 到底有什么不同。在这里需要对上面定义的 `Serialize<T>` 辅助方法作相应的修正，加入 `preserveObjectReferences` 参数，并通过该参数创建相应的 `DataContractSerializer` 对象。

```

47 static void Serialize<T>(T instance, string fileName, bool
preserveReference)
48 {
49     DataContractSerializer serializer = new
DataContractSerializer(typeof(T),
50         null, int.MaxValue, false, preserveReference, null);
51     using (XmlWriter writer = new XmlTextWriter(fileName,
Encoding.UTF8))
52     {
53         serializer.WriteObject(writer, instance);
54     }
55     Process.Start(fileName);
56 }

```

上面的例子需要对一个 `Customer` 对象进行序列化，`Customer` 的定义如下。须要注意的是 `Customer` 类中定义了两个属性：`CompanyAddress` 和 `ShipAddress`，它们的类型均为 `Address`。

```

57 namespace Artech.DataContractSerializerDemos
58 {
59     [DataContract]
60     public class Customer
61     {
62         [DataMember]
63         public string Name
64         { get; set; }
65     }

```

```

66         [DataMember]
67         public string Phone
68         { get; set; }
69
70         [DataMember]
71         public Address CompanyAddress
72         { get; set; }
73
74         [DataMember]
75         public Address ShipAddress
76         { get; set; }
77     }
78
79     [DataContract]
80     public class Address
81     {
82         [DataMember]
83         public string Province
84         { get; set; }
85
86         [DataMember]
87         public string City
88         { get; set; }
89
90         [DataMember]
91         public string District
92         { get; set; }
93
94         [DataMember]
95         public string Road
96         { get; set; }
97     }
98 }

```

### 5.2.2 数据契约的定义与数据契约序列化器 (5)

现在创建 Customer 对象，让 CompanyAddress 和 ShipAddress 属性引用同一个 Address 对象，先后通过 Serialize<T>方法将参数 preserveReference 分别设置为 false 和 true。

```

1  Address address = new Address()
2  {
3      Province = "Jiang Su",
4      City = "Su Zhou",
5      District = "Industrial Park",

```

```

6      Road = "Airport Rd #328"
7  };
8
9  Customer customer = new Customer()
10 {
11     Name = "Foo",
12     Phone = "8888-888888888",
13     ShipAddress = address,
14     CompanyAddress = address
15 };
16
17 Serialize<Customer>(customer, @"E:\customer1.xml", false);
18 Serialize<Customer>(customer, @"E:\customer2.xml", true);

```

下面两段 XML 片断分别表示两次序列化生成的 XML。可以很明显地看出，在不保留对象引用的情况下，CompanyAddress 和 ShipAddress 对应着两段具有相同内容的 XML 片断，而在保留对象引用的情况下，它们则引用同一个 XML 元素。

```

19 <Customer xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
20     xmlns="http://schemas.datacontract.org/2004/07/Artech.
21     DataContractSerializerDemos">
22     <CompanyAddress>
23         <City>Su Zhou</City>
24         <District>Industrial Park</District>
25         <Province>Jiang Su</Province>
26         <Road>Airport Rd #328</Road>
27     </CompanyAddress>
28     <Name>Foo</Name>
29     <Phone>8888-888888888</Phone>
30     <ShipAddress>
31         <City>Su Zhou</City>
32         <District>Industrial Park</District>
33         <Province>Jiang Su</Province>
34         <Road>Airport Rd #328</Road>
35     </ShipAddress>
36 </Customer>
37
38 <Customer xmlns:i="http://www.w3.org/2001/XMLSchema-instance" z:Id="1"
39     xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
40     xmlns="http://schemas.datacontract.org/2004/07/Artech.
41     DataContractSerializerDemos">
42     <CompanyAddress z:Id="2">
43         <City z:Id="3">Su Zhou</City>
44         <District z:Id="4">Industrial Park</District>
45         <Province z:Id="5">Jiang Su</Province>

```

```

46         <Road z:Id="6">Airport Rd #328</Road>
47     </CompanyAddress>
48     <Name z:Id="7">Foo</Name>
49     <Phone z:Id="8">8888-88888888</Phone>
50     <ShipAddress z:Ref="2" i:nil="true" />
51 </Customer>

```

前面介绍 `DataContractAttribute` 的时候，我们说到 `DataContractAttribute` 的属性 `IsReference` 起到 `PreserveObjectReferences` 属性一样的作用。在没有对 `DataContractSerializer` 的 `PreserveReference` 属性显式设置的情况下，将应用在 `Address` 上的 `DataContractAttribute` 的 `IsReference` 属性设为 `true`，同样可以实现保留对象引用的目的。

```

52 [DataContract(IsReference = true)]
53 public class Address
54 {
55     //类型成员
56 }

```

### 5.3 已知类型 (Known Type)

前面不止一次地强调，WCF 下的序列化与反序列化解决的是数据在两种状态之间相互转化的问题：托管类型对象和 XML。由于类型定义了对象的数据结构，所以无论是对于序列化还是反序列化，都必须事先确定对象的类型。如果被序列化对象或被反序列化生成的对象包含不可知的类型，序列化或反序列化将会失败。为了确保 `DataContractSerializer` 的正常序列化和反序列化，须要将“未知”类型加入 `DataContractSerializer`“已知”类型列表中。

#### 5.3.1 未知类型导致序列化失败

.NET 的类型可以分为两种：声明类型和真实类型。我们提倡面向接口的编程，对象的真实类型往往需要在运行时才能确定，在编程的时候只须指明类型的声明类型，比如类型实现的接口或抽象类。当我们使用基于接口或抽象类创建的 `DataContractSerializer` 去序列化一个实现了该接口或继承该抽象类的实例时，往往会因为无法识别对象的真实类型造成不能正常地序列化。比如在下面的代码中，我们定义了3个类型，一个接口、一个抽象类和一个具体类。

```

1 namespace Artech.DataContractSerializerDemos
2 {
3     public interface IOrder
4     {
5         Guid ID
6         { get; set; }
7     }

```

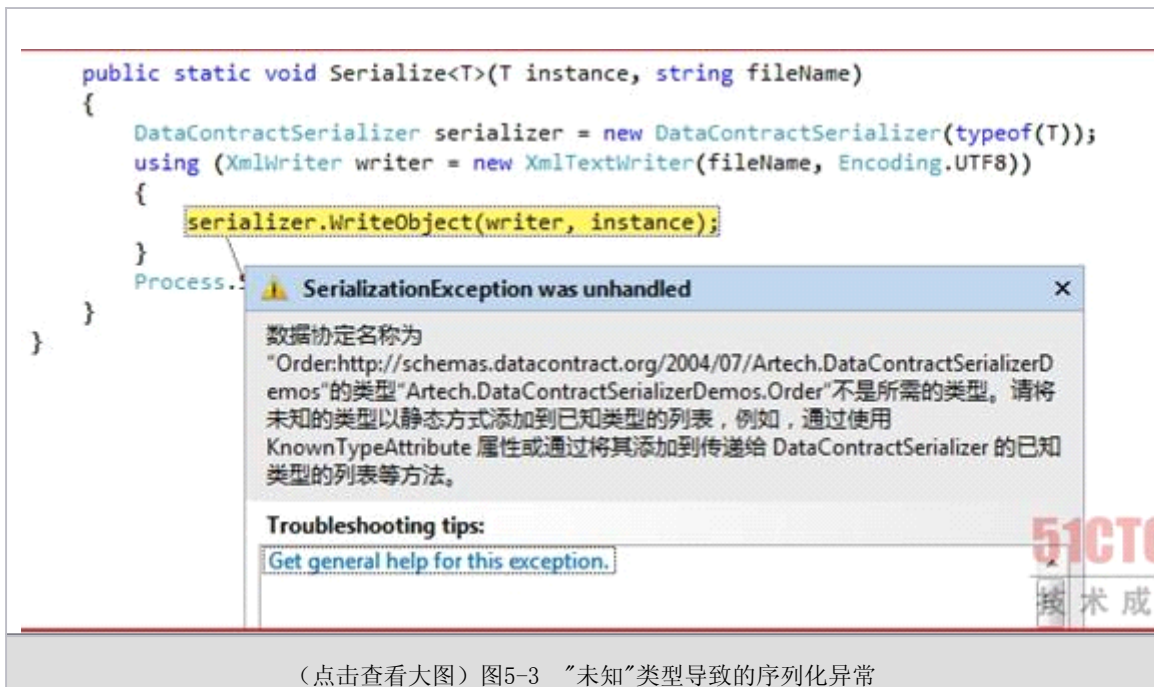
```

8         DateTime Date
9         { get; set; }
10
11        string Customer
12        { get; set; }
13
14        string ShipAddress
15        { get; set; }
16    }
17
18    [DataContract]
19    public abstract class OrderBase : IOrder
20    {
21        [DataMember]
22        public Guid ID
23        { get; set; }
24
25        [DataMember]
26        public DateTime Date
27        { get; set; }
28
29        [DataMember]
30        public string Customer
31        { get; set; }
32
33        [DataMember]
34        public string ShipAddress
35        { get; set; }
36    }
37
38    [DataContract]
39    public class Order : OrderBase
40    {
41        [DataMember]
42        public double TotalPrice
43        { get; set; }
44    }
45 }

```

当我们通过下面的方式去序列化一个 Order 对象（注意泛型类型为 IOrder 或 OrderBase）时，将会抛出如图5-3所示的 SerializationException 异常，提示 Order 类型无法识别。





```
46 Order order = new Order()
47 {
48     ID = Guid.NewGuid(),
49     Customer = "NCS",
50     Date = DateTime.Today,
51     ShipAddress = "#328, Airport Rd,
Industrial Park, Suzhou Jiangsu Province",
52     TotalPrice = 8888.88
53 };
54
55 Serialize<IOrder>(order, @"E:\order.xml");
56 //或者
57 Serialize<OrderBase>(order, @"E:\order.xml");
```

### 5.3.2 DataContractSerializer 的已知类型集合

解决上面这个问题的唯一途径就是让 DataContractSerializer 能够识别 Order 类型，成为 DataContractSerializer 的已知类型 (Known Type)。DataContractSerializer 内部具有一个已知类型的列表，我们只需要将 Order 的类型添加到这个列表中，就能从根本上解决这个问题。下面6个重载构造函数中的任意一个，均可以通过 KnownTypes 参数指定 DataContractSerializer 的已知类型集合，该集合最终反映在 DataContractSerializer 的制度属性 KnownTypes 上。

```
1 public sealed class DataContractSerializer :
XmlObjectSerializer
2 {
3     public DataContractSerializer(Type type,
IEnumerable<Type> knownTypes);
```

```

4     public DataContractSerializer(Type type,
string rootName, string
5         rootNamespace, IEnumerable<Type> knownTypes);
6     public DataContractSerializer(Type type,
XmlDictionaryString rootName,
7         XmlDictionaryString rootNamespace,
IEnumerable<Type> knownTypes);
8     public DataContractSerializer(Type type,
IEnumerable<Type> knownTypes,
9         int maxItemsInObjectGraph, bool
ignoreExtensionDataObject, bool
10        preserveObjectReferences, IDataContractSurrogate
dataContractSurrogate);
11    public DataContractSerializer(Type type,
string rootName, string
12        rootNamespace, IEnumerable<Type> knownTypes,
int maxItemsInObjectGraph,
13        bool ignoreExtensionDataObject, bool
preserveObjectReferences,
14        IDataContractSurrogate dataContractSurrogate);
15    public DataContractSerializer(Type type,
XmlDictionaryString rootName,
16        XmlDictionaryString rootNamespace,
IEnumerable<Type> knownTypes, int
17        maxItemsInObjectGraph, bool ignoreExtensionDataObject, bool
18        preserveObjectReferences, IDataContractSurrogate
dataContractSurrogate);
19
20    public ReadOnlyCollection<Type> KnownTypes { get; }
21 }

```

为了方便后面的演示，我们对使用的泛型服务方法 `Serialize<T>` 为已知类型作相应的修正，通过第3个参数指定 `DataContractSerializer` 的已知类型列表。

```

22 public static void Serialize<T>(T instance,
string fileName, IList<Type>
23     knownTypes)
24 {
25     DataContractSerializer serializer = new
DataContractSerializer(typeof(T),
26         knownTypes, int.MaxValue, false, false, null);
27     using (XmlWriter writer = new XmlTextWriter
(fileName, Encoding.UTF8))
28     {
29         serializer.WriteObject(writer, instance);

```

```

30     }
31     Process.Start(fileName);
32 }

```

### 5.3.3 基于接口的序列化

DataContractSerializer 的创建必须基于某个确定的类型，这里的类型既可以是接口，也可以是抽象类或具体类。不过基于接口的 DataContractSerializer 与基于抽象数据契约类型的 DataContractSerializer，在进行序列化时表现出来的行为是不相同的。

在下面的代码中，在调用 `Serialize<T>` 的时候，将泛型类型分别设定为接口 `IOrder` 和抽象类 `OrderBase`。虽然是对同一个 `Order` 对象进行序列化，但是序列化生成的 XML 却各有不同。文件 `order.interface.xml` 的根节点为 `<z:anyType>`，这是因为 `DataContractAttribute` 不能应用于接口上面，所以接口不具有数据契约的概念。`<z:anyType>` 表明能够匹配任意类型，相当于类型 `object`。

```

1  Order order = new Order()
2  {
3      ID = Guid.NewGuid(),
4      Customer = "NCS",
5      Date = DateTime.Today,
6      ShipAddress = "#328, Airport Rd, Industrial
Park, Suzhou Jiangsu Province",
7      TotalPrice = 8888.88
8  };
9
10 Serialize<IOrder>(order, @"E:\order.interface.
xml", new List<Type>{typeof
11     (Order)});
12 Serialize<OrderBase>(order, @"E:\order.class.
xml", new List<Type>
13     { typeof(Order) });
14 <z:anyType xmlns:i="http://www.w3.org/2001/
XMLSchema-instance"
15     xmlns:d1p1="http://schemas.datacontract.org/2004/07/Artech.
DataContractSerializerDemos" i:type="d1p1:Order"
16     xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
17     <d1p1:Customer>NCS</d1p1:Customer>
18     <d1p1:Date>2008-12-04T00:00:00+08:00</d1p1:Date>
19     <d1p1:ID>04c07e41-6302-48d1-ac06-87ebbf2b75f</d1p1:ID>
20     <d1p1:ShipAddress>#328, Airport Rd,
Industrial Park, Suzhou Jiangsu
21     Province</d1p1:ShipAddress>
22     <d1p1:TotalPrice>8888.88</d1p1:TotalPrice>

```

```

24 </z:anyType>
25 <OrderBase xmlns:i="http://www.w3.org/2001/
XMLSchema-instance" i:type="Order"
26   xmlns="http://schemas.datacontract.org/2004/07/Artech.
DataContractSerializerDemos">
27   <Customer>NCS</Customer>
28   <Date>2008-12-04T00:00:00+08:00</Date>
29   <ID>04c07e41-6302-48d1-ac06-87ebbf2b75f</ID>
30   <ShipAddress>#328, Airport Rd, Industrial Park, Suzhou Jiangsu
31   Province</ShipAddress>
32   <TotalPrice>8888.88</TotalPrice>
33 </OrderBase>

```

实际上，在 WCF 应用中，如果服务契约的操作参数定义为接口，在发布出来的元数据中，接口类型就相当于 object，并且当客户端通过添加服务引用生成客户端服务契约的时候，相应的参数类型就是 object 类型。比如对于下面的服务契约的定义，当客户端导出后将变成后面的样式。

```

35 [ServiceContract (Namespace="http://www.artech.com/")]
36 public interface IOrderManager
37 {
38     [OperationContract]
39     void ProcessOrder(IOrder order);
40 }
41 [System.CodeDom.Compiler.GeneratedCodeAttribute
("System.ServiceModel",
42   "3.0.0.0")]
43 [System.ServiceModel.ServiceContract
Attribute(ConfigurationName =
44   "ServiceReferences.IOrderManager")]
45 public interface IOrderManager
46 {
47
48     [System.ServiceModel.OperationContract
Attribute(Action =
49       "http://www.artech.com/IOrderManager/
ProcessOrder", ReplyAction =
50       "http://www.artech.com/IOrderManager/
ProcessOrderResponse")]
51     void ProcessOrder(object order);
52 }

```

#### 5.3.4 KnownTypeAttribute 与 ServiceKnownTypeAttribute

对于已知类型，可以通过两个特殊的自定义特性进行设置：KnownTypeAttribute 和

ServiceKnownTypeAttribute。KnownTypeAttribute 应用于数据契约中，用于设置继承于该数据契约类型的子数据契约类型，或者引用其他潜在的类型。ServiceKnownTypeAttribute 既可以应用于服务契约的接口和方法上，也可以应用在服务实现的类和方法上。应用的目标元素决定了定义的已知类型的作用范围。在下面的代码中，在基类 OrderBase 指定了子类的类型 Order。

```
1  [DataContract]
2  [KnownType(typeof(Order))]
3  public abstract class OrderBase : IOrder
4  {
5      //省略成员
6  }
```

而 ServiceKnownTypeAttribute 特性，不仅可以使用时在服务契约类型上，也可以应用在服务契约的操作方法上。如果应用在服务契约类型上，已知类型在所有实现了该契约的服务操作中有有效，如果应用于服务契约的操作方法上，则定义的已知类型在所有实现了该契约的服务对应的操作中有有效。

```
7  [ServiceContract]
8  [ServiceKnownType(typeof(Order))]
9  public interface IOrderManager
10 {
11     [OperationContract]
12     void ProcessOrder(OrderBase order);
13 }
14 [ServiceContract]
15 public interface IOrderManager
16 {
17     [OperationContract]
18     [ServiceKnownType(typeof(Order))]
19     void ProcessOrder(OrderBase order);
20 }
```

ServiceKnownTypeAttribute 也可以应用于具体的服务类型和方法上面。对于前者，通过 ServiceKnownTypeAttribute 定义的已知类型在整个服务的所有方法中有有效，而对于后者，则已知类型仅限于当前方法。

```
21 [ServiceKnownType(typeof(Order))]
22 public class OrderManagerService : IOrderManager
23 {
24     public void ProcessOrder(OrderBase order)
25     {
26         //省略成员
27     }
28 }
```

```

27     }
28 }
29 public class OrderManagerService : IOrderManager
30 {
31     [ServiceKnownType(typeof(Order))]
32     public void ProcessOrder(OrderBase order)
33     {
34         //省略成员
35     }
36 }

```

除了通过自定义特性的方式设置已知类型外，已知类型还可以通过配置的方式进行指定。已知类型定义在<system.runtime.serialization>配置节中，它采用如下的定义方式。这和我们在上面通过 ServiceKnownTypeAttribute 指定 Order 类型是完全等效的。

```

37 <?xml version="1.0" encoding="utf-8" ?>
38 <configuration>
39     <system.runtime.serialization>
40         <dataContractSerializer>
41             <declaredTypes>
42                 <add type="Artech.Data
ContractSerializerDemos.OrderBase,
43                     Artech.DataContractS
erializerDemos.KnownTypes">
44                     <knownType type="Artech.
DataContractSerializerDemos.
45                         Order, Artech.
DataContractSerializerDemos.
46                         KnownTypes"/>
47                 </add>
48             </declaredTypes>
49         </dataContractSerializer>
50     </system.runtime.serialization>
51 </configuration>

```

## 5.4 泛型数据契约与集合数据契约

在 .NET Framework 2.0 中，泛型第一次被引入。我们可以定义泛型接口、泛型类型、泛型委托和泛型方法。序列化依赖于真实具体的类型，而泛型则刻意模糊了具体类型概念。集合代表一组对象的组合，集合具有可迭代（Enumerable）的特性，它可以通过某个迭代规则遍历集合中的每一个元素。由于泛型类型和集合类型在序列化和反序列化上具有一些特殊的行为和规则，在本节中，我们特意进行单独的介绍。

### 5.4.1 泛型数据契约

面向对象通过继承实现了代码的重用，而泛型则实现了“算法的重用”。我们定义一种算法，比如排序、搜索、交换、比较或转换等，为了实现尽可能的重用，我们并不限定该算法操作对象的具体类型，而通过一个泛型类型来表示。在真正创建泛型对象或者调用该方法的时候，才指定其具体的类型。

就实现来说，泛型是 CLR 和编程语言（或者是基于编程语言的编译器）共同实现的一种特殊机制；就泛型的概念来说，这是面向对象的范畴。而我们现在介绍的数据契约，则属于面向服务的概念。两者有一些冲突，就像面向服务没有继承、重载的概念一样，面向服务通常也无法理解泛型。

但是基于 WCF 的编程语言是 C#、VB.NET 这样的完全面向对象的编程语言，而 WCF 服务却是基于面向服务的。所以，从某种意义上讲，WCF 的一个重大的作用就是弥合面向对象编程（OOP）和面向服务架构（SOA）之间的差异。我们现在就来看看 WCF 做了些什么，使我们能够以泛型类型的形式来定义数据契约。

首先通过一个简单的例子看看 DataContractSerializer 是如何序列化一个泛型对象的。为此我定义一个泛型类型 Bill<BillHeader, BillDetail>，它代表一个一般意义上的单据，BillHeader 和 BillDetail 代表单据报头的明细类型。两个属性 Header 和 Details 表示单据报头和明细列表。

```
1 namespace Artech.DataContractSerializerDemos
2 {
3     [DataContract(Namespace="http://www.artech.com/")]
4     public class Bill<BillHeader, BillDetail>
5     {
6         [DataMember(Order = 1)]
7         public BillHeader Header
8         { get; set; }
9
10        [DataMember(Order = 2)]
11        public BillDetail[] Details
12        { get; set; }
13    }
14 }
```

然后我们定义用于描述订单单据的报头和明细的类型：OrderBillHeader 和 OrderBillDetail。OrderBillHeader 描述订单的总体信息，OrderBillDetail 实际上表示订单中每一个产品的 ID、单价和数量。

```
15 namespace Artech.DataContractSerializerDemos
16 {
17     [DataContract(Namespace="http://www.artech.com/")]
18     public class OrderBillHeader
```

```

19     {
20         [DataMember]
21         public Guid OrderID
22         { get; set; }
23
24         [DataMember]
25         public DateTime Date
26         { get; set; }
27
28         [DataMember]
29         public string Customer
30         { get; set; }
31     }
32
33     [DataContract(Namespace="http://www.artech.com/")]
34     public class OrderBillDetail
35     {
36         [DataMember]
37         public Guid ProductID
38         { get; set; }
39
40         [DataMember]
41         public int Quantity
42         { get; set; }
43
44         [DataMember]
45         public double UnitPrice
46         { get; set; }
47     }
48 }

```

通过下面一个方法创建泛型类型 `Bill<BillHeader, BillDetail>` 对象，泛型类型指定为上面定义的 `OrderBillHeader` 和 `OrderBillDetail`。

```

49 private static Bill<OrderBillHeader, OrderBillDetail> CreateOrderBill()
50 {
51     OrderBillHeader header = new OrderBillHeader
52     {
53         OrderID    = Guid.NewGuid(),
54         Date       = DateTime.Today,
55         Customer   = "Foo"
56     };
57
58     IList<OrderBillDetail> details = new List<OrderBillDetail>();
59     OrderBillDetail detail = new OrderBillDetail

```



```

60     {
61         ProductID    = Guid.NewGuid(),
62         Quantity     = 20,
63         UnitPrice    = 888
64     };
65     details.Add(detail);
66     detail = new OrderBillDetail
67     {
68         ProductID    = Guid.NewGuid(),
69         Quantity     = 10,
70         UnitPrice    = 9999
71     };
72     details.Add(detail);
73
74
75     Bill<OrderBillHeader, OrderBillDetail> orderBill = new
76         Bill<OrderBillHeader, OrderBillDetail>()
77     {
78         Header       = header,
79         Details       = details.ToArray<OrderBillDetail>()
80     };
81     return orderBill;
82 }

```

借助在前面定义的 `Serialize<T>` 辅助方法，对创建 `Bill<OrderBillHeader, OrderBillDetail>` 对象进行序列化。最终对象将被序列化成如下的 XML。

```

83 Bill<OrderBillHeader, OrderBillDetail> orderBill = CreateOrderBill();
84 Serialize<Bill<OrderBillHeader, OrderBillDetail>>(orderBill,
85     @"orderbill.xml");
86 <BillOfOrderBillHeaderOrderBillDetail6Of3LqKh
87     xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
88     xmlns="http://www.artech.com/Artech.
DataContractSerializerDemos">
89     <Header>
90         <Customer>NCS</Customer>
91         <Date>2008-12-04T00:00:00+08:00</Date>
92         <OrderID>15a62aae-c955-4bc0-acb6-
e171fb9fe085</OrderID>
93     </Header>
94     <Details>
95         <OrderBillDetail>
96             <ProductID>f7679949-938a-40a0-
a32a-5dde5c85e55f</ProductID>
97             <Quantity>20</Quantity>

```

```

98         <UnitPrice>888</UnitPrice>
99     </OrderBillDetail>
100 </OrderBillDetail>
101     <ProductID>bbd750ff-8b0c-48f5-
ab1f-5ad7e51bd420</ProductID>
102     <Quantity>10</Quantity>
103     <UnitPrice>9999</UnitPrice>
104 </OrderBillDetail>
105 </Details>
106 </BillOfOrderBillHeaderOrderBillDetail60f3LqKh>

```

XML 整体的结构正是我们希望的，关键是根节点名称，也就是数据契约的名称，“BillOfOrderBillHeaderOrderBillDetail60f3LqKh”，会让有些人难以理解。仔细分析一下数据契约的名称，会发现它的组成结构是这样的：{类型名称（Bill）}+ Of + {第一个泛型参数的类型（OrderBillHeader）} + {第二个泛型参数的类型（OrderBillDetail）}+ {哈希值（60f3LqKh）}。

这里说泛型参数的类型，实际上是不对的，应该说 OrderBillHeader 和 OrderBillDetail 的泛型类型对应的数据契约的名称。在下面的代码中，通过 DataContractAttribute 特性修改了数据契约的名称（OrderHeader 和 OrderDetail），最终的数据契约的名称将会变成：BillOfOrderHeaderOrderDetail60f3LqKh。可以看出描述泛型数据契约的部分内容相应地改变了。可能仔细的读者已经发现了，哈希值部分却没有发生变化，依然是“60f3LqKh”，因为这是泛型类型（含命名空间）的哈希值，而不是数据契约名称的哈希值。所以我们可以将默认的基于泛型类型的命名规则表示成：[类型名称][泛型数据契约名称1][泛型数据契约名称2][...][含命名空间的泛型类型哈希值]。

```

107 namespace Artech.DataContractSerializerDemos
108 {
109     [DataContract(Name="OrderHeader")]
110     public class OrderBillHeader
111     {
112         //省略成员
113     }
114
115     [DataContract(Name = "OrderDetail")]
116     public class OrderBillDetail
117     {
118         //省略成员
119     }
120 }

```

WCF 之所以要采用这样的数据契约命名方式，是为了解决命名冲突，保证数据契约名称

的唯一性。前面说了，面向服务下的数据契约完全没有泛型的概念，对它来说所有的类型都是“实实在在”的具体类型。对于泛型类型 `Bill<BillHeader, BillDetail>`，不同的 `BillHeader` 和 `BillDetail` 组合代表不同的数据契约，所以最终的数据契约的名称需要由自身类型和泛型契约名称派生出来。由于在定义数据契约的时候，不同的 CLR 类型可以指定相同的数据契约名称，所以加上一个基于所有泛型类型（含命名空间）的哈希值可以确保数据契约的唯一性。

WCF 在进行元数据发布的时候，会自动按照这样的命名机制创建数据契约，并以 XSD 的形式发布出来。所以当客户端导入元数据生成客户端代码的时候，生成的等效数据契约的类型名称就是这个经过拼接的名称。下面是 `Bill<OrderBillHeader, OrderBillDetail>` 导入的形式。

```
121 public partial class BillOfOrderBill
HeaderOrderBillDetail6Of3LqKh : object,
122     System.Runtime.Serialization.IExtensibleDataObject,
123     System.ComponentModel.INotifyPropertyChanged
124 {
125     //省略成员
126 }
```

如果你能够确保命名不会发生冲突，可以通过 `DataContractAttribute` 特性的 `Name` 属性对数据契约的名称进行显式设置。比如在下面的代码中，将契约名称限定为“OrderBill”。不过这样设置就意味着已假定泛型类型只能表示基于订单的单据了，这相当于失去了泛型的意义。

```
127 namespace Artech.DataContractSerializerDemos
128 {
129     [DataContract(Name="OrderBill")]
130     public class Bill<BillHeader, BillDetail>
131     {
132         //省略成员
133     }
134 }
```

所以我们可以采用一种动态的设置方式，为数据契约的名称指定一个模板，使用表示泛型数据契约名称和泛型类型哈希值的占位符。其中 `{0}`、`{1}` 表示的是泛型数据契约的名称，数字表示相应的泛型参数出现的次序，而哈希值则通过 `{#}` 表示。所以下面两种泛型数据契约是完全等效的。

```
135 namespace Artech.DataContractSerializerDemos
136 {
137     [DataContract]
138     public class Bill<BillHeader, BillDetail>
```

```

139     {
140         //省略成员
141     }
142 }
143 namespace Artech.DataContractSerializerDemos
144 {
145     DataContract(Name="BillOf{0}{1}{#}")
146     public class Bill<BillHeader, BillDetail>
147     {
148         //省略成员
149     }
150 }

```

#### 5.4.2 数据契约对数组与集合的支持 (1)

在 .NET 中,所有的集合都实现了 `IEnumerable` 接口,比如 `Array`、`Hashtable`、`ArrayList`、`Stack`、`Queue` 等。有的集合要求元素具有相同的类型,这种集合一般通过泛型的方式定义,它们实现另一个接口 `IEnumerable<T>` (`IEnumerable<T>` 本身继承自 `IEnumerable`),这样的集合有 `List<T>`、`Dictionary<TKey, TValue>`、`Stack<T>`、`Queue<T>` 等。基于集合类型的序列化具有一些特殊的规则和行为,接下来我们将分析基于集合对象的序列化,以及基于集合类型的服务操作。

1 `IEnumerable<T>`、`Array` 与 `ICollection<T>`

一个集合对象能够被序列化的前提是集合中的每个元素都能被序列化,也就是要求元素的类型是一个数据契约(或者是应用了 `SerializableAttribute` 特性)。虽然集合具有各种各样的表现形式,由于其本质就是一组对象的组合, `DataContractSerializer` 在对它们进行序列化的时候,采用的序列化规则和在序列化过程中表现出来的行为是相似的。比如现在需要通过 `DataContractSerializer` 序列化一个 `Customer` 对象的集合, `Customer` 类型定义如下。

```

2 namespace Artech.DataContractSerializerDemos
3 {
4     [DataContract(Namespace="http://www.artech.com/")]
5     public class Customer
6     {
7         [DataMember(Order = 1)]
8         public Guid ID
9         { get; set; }
10
11         [DataMember(Order=2)]
12         public string Name

```

```

13         { get; set; }
14
15         [DataMember(Order = 3)]
16         public string Phone
17         { get; set; }
18
19         [DataMember(Order = 4)]
20         public string CompanyAddress
21         { get; set; }
22     }
23 }

```

现在通过我们前面定义的泛型 `Serialize<T>` 对以下3种不同类型的集合对象进行序列化: `IEnumerable<Customer>`、`IList<Cusomter>` 和 `Customer[]`。

```

24 Customer customerFoo
25 = new Customer
26     {
27         ID
28         = Guid.NewGuid(),
29         Name
30         = "Foo",
31         Phone
32         = "8888-88888888",
33         CompanyAddress = "#9981, West Sichuan
34         Rd, Xian Shanxi Province"
35     };
36 Customer customerBar = new Customer
37     {
38         ID
39         = Guid.NewGuid(),
40         Name
41         = "Bar",
42         Phone
43         = "9999-99999999",
44         CompanyAddress
45         = "#3721, Taishan Rd, Jinan ShanDong Province"
46     };
47 Customer[] customerArray = new Customer[]
48 { customerFoo, customerBar };
49 IEnumerable<Customer> customerCollection = customerArray;
50 IList<Customer> customerList = customerArray.ToList<Customer>();

```

```

51 Serialize<Customer[]>(customerArray,
    @"E:\Customer.Array.xml");
52
53 Serialize<IEnumerable<Customer>>( customerCollection,
54     @"E:\Customer.GenericIEnumerable.xml");
55 Serialize<IList<Customer>>( customerList,
    @"E:\Customer.GenericIList.xml");

```

我们最终发现，虽然创建 DataContractSerializer 对象使用的类型不一样，但是最终序列化生成出来的 XML 却是完全一样的，也就是说 DataContractSerializer 在序列化这3种类型对象时，采用了完全一样的序列化规则。从下面的 XML 的结构和内容中，我们可以总结出下面3条规则。

根节点的名称以 ArrayOf 为前缀，后面紧跟集合元素类型对应的数据契约名称；

集合元素对象用数据契约的命名空间作为整个集合契约的命名空间；

每个元素对象按照其数据契约定义进行序列化。

```

56 <ArrayOfCustomer xmlns:i="http://www.w3.
    org/2001/XMLSchema-instance"
57     xmlns="http://www.artech.com/">
58     <Customer>
59         <ID>8baed181-bcbc-493d-8592-3e08fd5ad1cf</ID>
60         <Name>Foo</Name>
61         <Phone>8888-88888888</Phone>
62         <CompanyAddress>#9981, West Sichuan Rd, Xian Shanxi
63             Province</CompanyAddress>
64     </Customer>
65     <Customer>
66         <ID>2fca9719-4120-430c-9dc2-3ef9dc7dffbf</ID>
67         <Name>Bar</Name>
68         <Phone>9999-99999999</Phone>
69         <CompanyAddress>#3721, Taishan Rd, Jinan ShanDong
70             Province</CompanyAddress>
71     </Customer>
72 </ArrayOfCustomer>

```

我们从根节点的名称 ArrayOfCustomer 可以看出，WCF 将这3个类型的对象 IEnumerable<Customer>、IList<Customer>和 Customer[]都作为 Customer 数组了。实际上，如果你在定义服务契约的时候，将某个服务操作的参数类型设为 IEnumerable<T>或<IList>，在默认导出生成的服务契约中，相应的参数类型就是数组类型。比如，在同一个服务契约中，定义了如下3个操作，它们的参数类型分别为 IEnumerable<Customer>、IList<Customer>和 Customer[]。当客户端通过添加服务引用导出服务契约后，3个操作的

参数类型都变成 Customer[] 了。

```
73 [ServiceContract]
74 public interface ICustomerManager
75 {
76     [OperationContract]
77     void AddCustomerArray(Customer[] customers);
78     [OperationContract]
79     void AddCustomerCollection(IEnumerable<Customer> customers);
80     [OperationContract]
81     void AddCustomerList(IList<Customer> customers);
82 }
83 [ServiceContract]
84 [ServiceContract]
85 public interface ICustomerManager
86 {
87     [OperationContract]
88     void AddCustomerArray(Customer[] customers);
89     [OperationContract]
90     void AddCustomerCollection(Customer[] customers);
91     [OperationContract]
92     void AddCustomerList(Customer[] customers);
93 }
```

#### 5.4.2 数据契约对数组与集合的支持（2）

由于对于 DataContractSerializer 来说，IEnumerable<Customer>、IList<Customer> 和 Customer[] 这3种形式的数据表述方式是等效的，那么就意味着当客户端在通过添加服务引用导入服务契约的时候，customers 通过 Customer[] 表示与通过 IList<Customer> 表示具有等效性，我们能否让数组类型变成 IList<T> 类型呢，毕竟从编程角度来看，它们还是不同的，很多时候使用 IList<T> 要比直接使用数组方便得多。

答案是肯定的，Visual Studio 允许我们在添加服务引用的时候进行一些定制，其中生成的集合类型和字典集合类型的定制就包含其中。如图5-4所示，VS 提供了6种不同的集合类型供我们选择：Array、ArrayList、LinkedList、GenericList、Collection、BindingList。

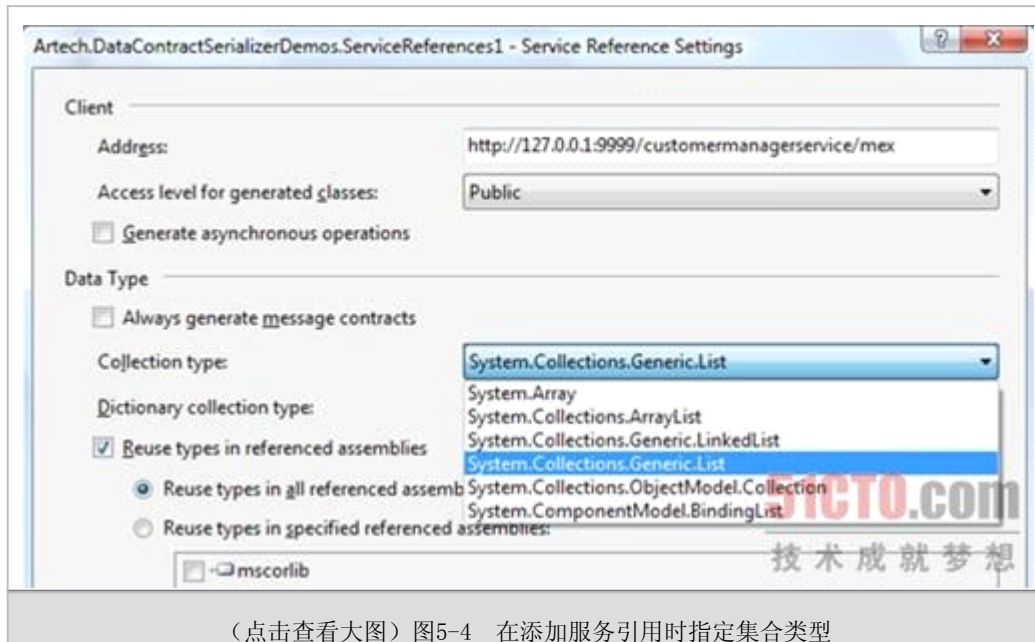
对于上面定义的服务契约 ICustomerManager，如果在添加服务引用时使用 GenericList 选项，导入的服务契约的所有操作参数类型全部会变成 List<Customer>。

```
1 [ServiceContract]
2 public interface ICustomerManager
3 {
4     [OperationContract]
5     void AddCustomerArray(List<Customer> customers);
```

```

6      [OperationContract]
7      void AddCustomerCollection(List<Customer> customers);
8      [OperationContract]
9      void AddCustomerList(List<Customer> customers);
10 }

```



## IEnumerable 与 IList

上面我们介绍了 `IEnumerable<T>`、`Array` 与 `IList<T>` 这3种集合类型的序列化规则，这3种集合类型有一个共同的特点，那就是集合类型的申明指明了集合元素的类型。当基于这3种集合类型的 `DataContractSerializer` 被创建出来时，由于元素类型已经明确了，所以能够按照元素类型对应的数据契约的定义进行合理的序列化工作。但是对于不能预先确定元素类型的 `IEnumerable` 和 `IList` 则不能如此操作。

下面我将演示 `IEnumerable` 和 `IList` 两种类型的序列化。在介绍已知类型的时候，我们已经明确了无论是序列化还是反序列化都需要预先明确对象的真实类型，对于不能预先确定具体类型的情况下，我们需要把潜在的类型添加到 `DataContractSerializer` 的已知类型列表中，才能保证序列化和反序列化的正常进行。由于创建基于 `IEnumerable` 和 `IList` 的 `DataContractSerializer` 的时候，集合元素类型是不可知的，所以需要将潜在的元素类型添加到 `DataContractSerializer` 的已知类型列表中，为此我们使用下面一个包含已知类型列表参数的 `Serialize<T>` 辅助方法进行序列化工作。

```

11 public static void Serialize<T>(T instance,
12     string fileName, IList<Type>
13     knownTypes)
14 {

```



```

14     DataContractSerializer serializer = new
DataContractSerializer(typeof(T),
15         knownTypes, int.MaxValue, false, false, null);
16     using (XmlWriter writer = new XmlTextWriter
(fileName, Encoding.UTF8))
17     {
18         serializer.WriteObject(writer, instance);
19     }
20     Process.Start(fileName);
21 }

```

为了和基于 IEnumerable<T>、Array 与 IList<T>序列化做对比，将采用相同的编程方式，使用相同的数据。

```

22 Customer customerFoo
23 = new Customer
24     {
25         ID
26 = Guid.NewGuid(),
27         Name
28 = "Foo",
29         Phone
30 = "8888-88888888",
31         CompanyAddress = "#9981, West
Sichuan Rd, Xian Shanxi Province"
32     };
33
34 Customer customerBar = new Customer
35 {
36     ID
37 = Guid.NewGuid(),
38     Name
39 = "Bar",
40     Phone
41 = "9999-99999999",
42     CompanyAddress
43 = "#3721, Taishan Rd, Jinan ShanDong Province"
44 };
45 Customer[] customers = new Customer[] { customerFoo, customerBar };
46 IEnumerable customersCollection = customers;
47 IList customersList = customers.ToList();
48
49
50 Serialize<IEnumerable>(customersCollection,
@"E:\Customer.IEnumerable.xml",

```

```

51 new List<Type> { typeof(Customer) });
52 Serialize<IList>(customersList, @"E:\Customer.IList.xml", new
List<Type>
53 { typeof(Customer) });

```

无论是基于 IEnumerable 类型，还是基于 IList，最终序列化生成的 XML 都是一样的。对于 IEnumerable 和 IList，集合元素的类型没有限制，可以是任何类型的对象，所以根节点的名称为 ArrayOfanyType，每个子节点的名称为 anyType。在真正对具体的元素对象进行序列化的时候，通过反射并借助于已知类型，获得相应数据契约的定义，并以此为依据进行序列化。

```

54 <ArrayOfanyType xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
55   xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
56       <anyType xmlns:d2p1="http://www.artech.com"
i:type="d2p1:Customer">
57           <d2p1:ID>604cc219-d1fe-4e0c-92c8-83486e13b354</d2p1:ID>
58           <d2p1:Name>Foo</d2p1:Name>
59           <d2p1:Phone>8888-888888888</d2p1:Phone>
60           <d2p1:CompanyAddress>#9981, West Sichuan Rd, Xian Shanxi
61             Province</d2p1:CompanyAddress>
62       </anyType>
63       <anyType xmlns:d2p1="http://www.artech.com"
i:type="d2p1:Customer">
64           <d2p1:ID>ef3e2806-789a-4208-974d-a67f8ed92e4c</d2p1:ID>
65           <d2p1:Name>Bar</d2p1:Name>
66           <d2p1:Phone>9999-999999999</d2p1:Phone>
67           <d2p1:CompanyAddress>#3721, Taishan Rd, Jinan ShanDong
68             Province</d2p1:CompanyAddress>
69       </anyType>
70 </ArrayOfanyType>

```

#### 5.4.2 数据契约对数组与集合的支持（3）

ArrayOfanyType 的含义是任何类型的数组，在 .NET 中，就是 object[]。而实际上，对于服务契约来说，如果某个操作包含 IEnumerable 或 IList 类型的参数，当该服务契约被客户端导入时，IEnumerable 或 IList 参数类型将会自动转换成 object[]。比如对于下面的服务契约，其导入形式如后面的代码所示。当然也可以通过修改服务引用输出集合类型，使参数类型按照你希望的形式输出（如果选择 GenericList，那么参数类型将会转换为 List<object>）。

```

1 [ServiceContract]
2 [ServiceKnownType(typeof(Customer))]
3 public interface ICustomerManager
4 {

```

```

5      void AddCustomerCollection(IEnumerable customers);
6      [OperationContract]
7      void AddCustomerList(IList customers);
8  }
9  [ServiceContract]
10 [ServiceKnownType(typeof(Customer))]
11 public interface ICustomerManager
12 {
13     void AddCustomerCollection(object[] customers);
14     [OperationContract]
15     void AddCustomerList(object[] customers);
16 }

```

### 自定义集合数据契约类型

前面我们基本上都是在介绍基于系统定义集合类型的序列化问题，而在很多情况下，我们会自定义一些集合类型。那么在 WCF 下对自定义集合类型有哪些限制，DataContractSerializer 对于自定义集合类型又有着怎样的序列化规则呢？接下来就来讨论这些问题。

为了演示基于自定义集合类型的序列化，我定义了下面一个集合类型：CustomerCollection 它，表示一组 Customer 对象的组合。Customer 的列表通过 IList<Customer> 类型成员保存；定义了两个构造函数，无参构造函数没有任何实现，另一个则提供 Customer 对象列表；Add 方法方便添加 Customer 对象成员。

```

17 namespace Artech.DataContractSerializerDemos
18 {
19     public class CustomerCollection : IEnumerable<Customer>
20     {
21         private IList<Customer> _customers = new List<Customer>();
22
23         public CustomerCollection(IList<Customer> customers)
24         {
25             this._customers = customers;
26         }
27
28         public CustomerCollection()
29         { }
30
31         public void Add(Customer customer)
32         {
33             this._customers.Add(customer);
34         }
35

```

```

36         //IEnumerable<Customer> 成员
37         public IEnumerator<Customer> GetEnumerator()
38         {
39             return this._customers.GetEnumerator();
40         }
41
42         //IEnumerable 成员
43         IEnumerator IEnumerable.GetEnumerator()
44         {
45             return this._customers.GetEnumerator();
46         }
47     }
48 }

```

借助于前面定义的 `Serialize<T>` 辅助方法，对创建出来的 `CustomerCollection` 对象进行序列化。

```

49 IList<Customer> customers = new List<Customer>
50 {
51     new Customer
52     {
53         ID
54         = Guid.NewGuid(),
55         Name
56         = "Foo",
57         Phone
58         = "8888-88888888",
59         CompanyAddress = "#9981, West Sichuan Rd, Xian Shanxi
Province"
60     },
61     new Customer
62     {
63         ID
64         = Guid.NewGuid(),
65         Name
66         = "Bar",
67         Phone
68         = "9999-99999999",
69         CompanyAddress = "#3721, Taishan Rd, Jinan ShanDong Province"
70     }
71 };
72
73 Serialize<CustomerCollection>(new CustomerCollection(customers),
74     @"e:\customers.xml");

```

执行上面的代码，将会得到下面一段被序列化生成的 XML。通过与上面生成的 XML 比较，我们发现基于自定义 CustomerCollection 对象序列化的 XML 与基于 IEnumerable<Customer>、IList<Customer>和 Customer[]的 XML 完全是一样的。这是因为 CustomerCollection 实现了 IEnumerable<Customer>接口。所以从数据契约的角度来看 CustomerCollection 和 IEnumerable<Customer>、IList<Customer>与 Customer[]，它们是完全等效的。

```
75 <ArrayOfCustomer xmlns:i="http://
www.w3.org/2001/XMLSchema-instance"
76   xmlns="http://www.artech.com">
77   <Customer>
78     <ID>504afb71-c765-48c2-97f4-a1100e81able</ID>
79     <Name>Foo</Name>
80     <Phone>8888-88888888</Phone>
81     <CompanyAddress>#9981, West Sichuan Rd, Xian Shanxi
82       Province</CompanyAddress>
83   </Customer>
84   <Customer>
85     <ID>5c1a3469-fc80-4a28-8d17-79f2c849193d</ID>
86     <Name>Bar</Name>
87     <Phone>9999-99999999</Phone>
88     <CompanyAddress>#3721, Taishan Rd, Jinan ShanDong
89       Province</CompanyAddress>
90   </Customer>
91 </ArrayOfCustomer>
92 CollectionDataContractAttribute
```

既然 CustomerCollection 和 IEnumerable<Customer>、IList<Customer>与 Customer[] 完全等效，那么自定义集合类型对于数据契约有什么意义呢？因为 Customer 是一个数据契约，IEnumerable<Customer>、IList<Customer>与 Customer[] 只能算是数据契约的集合，其本身并不算是一个数据契约。而通过自定义集合类型，我们可以将集合整体定义成一个数据契约，把基于集合的数据契约称为集合数据契约（Collection Data Contract）。集合数据契约通过 System.Runtime.Serialization.CollectionDataContractAttribute 特性定义。我们先来看看 CollectionDataContractAttribute 的定义。Name、Namespace 和 IsReference，与 DataContractAttribute 中的同名属性具有相同的含义。额外的3个属性成员分别表示为：

ItemName: 集合元素的名称，默认值为集合元素数据契约的名称。

KeyName: 针对于字典型（Key-Value Pair）集合，表示每个 Item 的 Key 的名称。

ValueName: 针对于字典型（Key-Value Pair）集合，表示每个 Item 的 Value 的名称。

```

93 [AttributeUsage(AttributeTargets.Struct
| AttributeTargets.Class, Inherited
94 = false, AllowMultiple = false)]
95 public sealed class CollectionDataContractAttribute : Attribute
96 {
97     public CollectionDataContractAttribute();
98
99     public string    Name { get; set; }
100    public string    Namespace { get; set; }
101    public bool      IsReference { get; set; }
102
103    public string    ItemName { get; set; }
104    public string    KeyName { get; set; }
105    public string    ValueName { get; set; }
106 }

```

#### 5.4.2 数据契约对数组与集合的支持（4）

我们通过 `CollectionDataContractAttribute` 对 `CustomerCollection` 进行如下的改造，将集合契约名称指定为 `CustomerList`，集合元素的名称为 `CustomerEntry`，重写命名空间 <http://www.artech.com/collection>。后面的 XML 反映出了改造的成果。

```

1 namespace Artech.DataContractSerializerDemos
2 {
3     [CollectionDataContract(Name = "CustomerList", ItemName =
"CustomerEntry",
4         Namespace = "http://www.artech.com/collection/")]
5     public class CustomerCollection:IEnumerable<Customer>
6     {
7         //省略成员
8     }
9 }
10 <CustomerList xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
11     xmlns:d1p1="http://www.artech.com"
12     xmlns="http://www.artech.com/collection/">
13     <CustomerEntry>
14         <d1p1:ID>d780f6c4-7d3d-427b-a6e7-10220c77d349</d1p1:ID>
15         <d1p1:Name>Foo</d1p1:Name>
16         <d1p1:Phone>8888-888888888</d1p1:Phone>
17         <d1p1:CompanyAddress>#9981, West Sichuan Rd, Xian Shanxi
18             Province</d1p1:CompanyAddress>
19     </CustomerEntry>
20     <CustomerEntry>
21         <d1p1:ID>d0e82e4d-702a-40cd-a58f-540eb8213578</d1p1:ID>
22         <d1p1:Name>Bar</d1p1:Name>

```

```
23      <d1p1:Phone>9999-99999999</d1p1:Phone>
24      <d1p1:CompanyAddress>#3721, Taishan Rd, Jinan ShanDong
25          Province</d1p1:CompanyAddress>
26    </CustomerEntry>
27  </CustomerList>
```

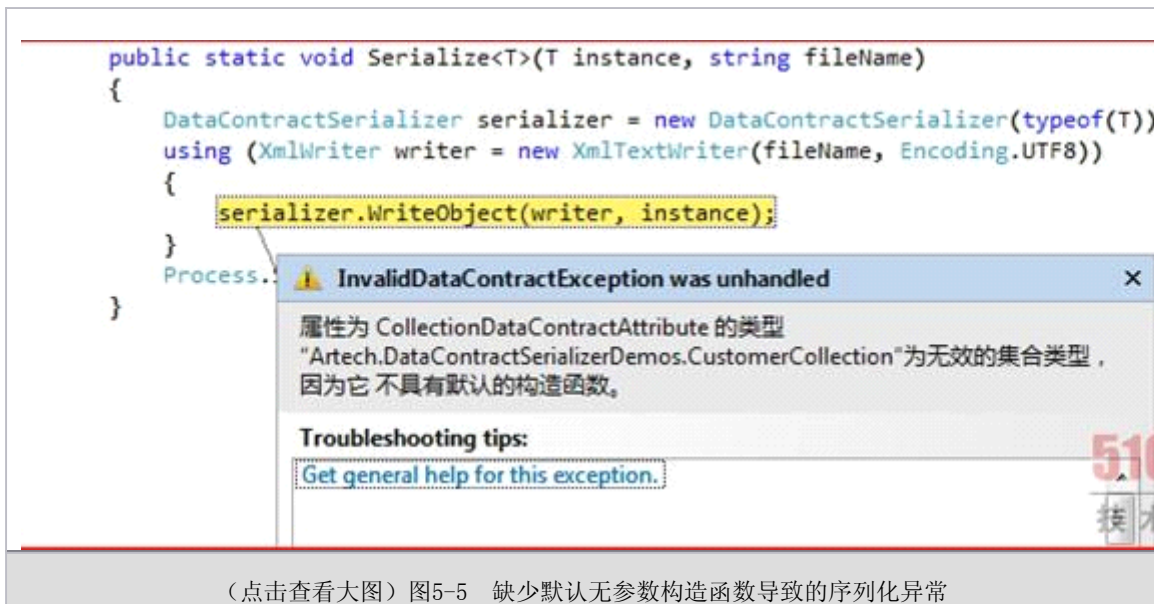
#### 默认无参数构造函数的必要性

我想有的读者可能会觉得奇怪，在定义 CustomerCollection 的时候，为什么加上一个默认无参的构造函数，这不是多此一举吗？根本就没有添加任何代码在此构造函数中。

```
28 namespace Artech.DataContractSerializerDemos
29 {
30     public class CustomerCollection:IEnumerable<Customer>
31     {
32         //其他成员
33         public CustomerCollection()
34         { }
35     }
36 }
```

实际上，这个默认无参的构造函数并非随意加上去的，这是为了保证 DataContractSerializer 正常工作所必须的。在使用 DataContractSerializer 对某个对象进行序列化的时候，我们不能光看到序列化本身，还要看到与之相对的操作：反序列化。如果不同时保证正常的反序列化，序列化实际上没有太大的意义。而默认无参的构造函数的存在就是为反序列化服务的，因为 DataContractSerializer 在将 XML 反序列化成某种类型的对象时，需要通过反射调用默认的构造函数，并创建对象。所以对于 CustomerCollection 来说，默认的构造函数是必须的。

如果我们将此默认无参的构造函数去掉，运行程序将会抛出如图 5-5 所示的 InvalidDataContractException 异常。这表明没有默认构造函数的 CustomerCollection 是不合法的集合契约。



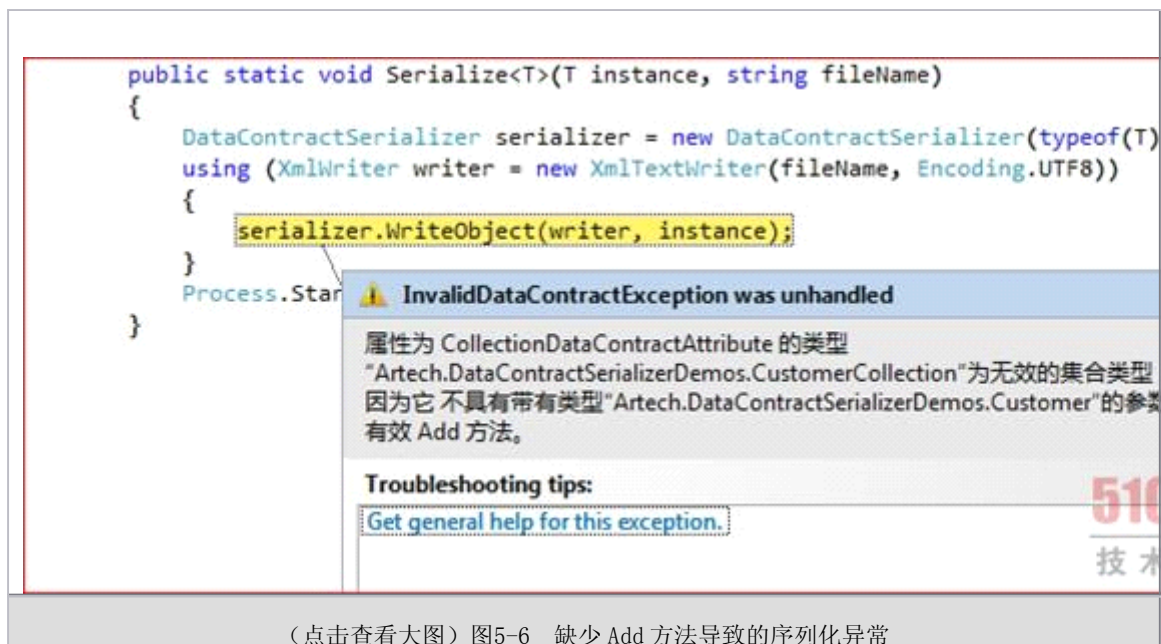
### Add 方法的必要性

在 CustomerCollection 类型中，为了更加方便地添加 Customer 对象到集合中，我定义了 Add 方法。实际上，此 Add 方法的意义并非仅仅是一个辅助方法，它和默认构造函数一样，是集合数据契约所必须的。

```
37 namespace Artech.DataContractSerializerDemos
38 {
39     public class CustomerCollection:IEnumerable<Customer>
40     {
41         //其他成员
42         public void Add(Customer customer)
43         {
44             this._customers.Add(customer);
45         }
46     }
47 }
```

如果我们将此 Add 方法从 CustomerCollection 中去掉。会抛出如图 5-6 所示的 InvalidDataContractException 异常，这表明没有 Add 方法的 CustomerCollection 是不合法的集合数据契约。所以在定义集合数据契约的时候，哪怕你不需要，都必须加上一个空的 Add 方法。





### 简化自定义集合数据契约定义

为了演示默认构造函数和 Add 方法对于集合数据契约的必要性，再定义 CustomerCollection 的实现，它仅仅是实现了 IEnumerable<Customer>结构，并通过封装一个 IList<Customer>对象实现了 IEnumerable<Customer>的方法。实际上，只须让 CustomerCollection 继承 List<Customer>就可以了。

```
48 namespace Artech.DataContractSerializerDemos
49 {
50     public class CustomerCollection:List<Customer>
51     {
52     }
53 }
```

#### 5.4.3 IDictionary 与 Hashtable (1)

IDictionary 与 Hashtable 是一种特殊的集合类型，它的集合元素类型是一个键-值对 (Key-Value Pair)，前者通过泛型参数指明了 Key 和 Value 的类型，后者则可以使用任何类型，或者说 Key 和 Value 的类型都是 object。

照例通过一个具体的例子看看 WCF 在通过 DataContractSerializer 序列化 IDictionary 与 Hashtable 对象的时候，采用怎样的序列化规则。我们使用基于 Customer 对象的 IDictionary 和 Hashtable，Key 和 Value 分别使用 Customer 的 ID 和 Customer 对象本身，Customer 类型在前面已经定义了。借助前面定义的两个 Serialize 辅助方法，对表

示相同 Customer 集合的 IDictionary 与 Hashtable 对象进行序列化，由于对于 Hashtable 来说，无法确定集合元素的具体类型，我们需要将 Customer 类型作为 DataContractSerializer 的已知类型。

```
1  Customer customerFoo    = new Customer
2      {
3          ID            = Guid.NewGuid(),
4          Name          = "Foo",
5          Phone         = "8888-88888888",
6          CompanyAddress = "#9981, West Sichuan
Rd, Xian Shanxi Province"
7      };
8
9  Customer customerBar    = new Customer
10     {
11         ID            = Guid.NewGuid(),
12         Name          = "Bar",
13         Phone         = "9999-99999999",
14         CompanyAddress = "#3721, Taishan Rd,
Jinan ShanDong Province"
15     };
16
17  IDictionary<Guid, Customer> customerDictionary
= new Dictionary<Guid,
18  Customer>();
19  Hashtable customerHashtable = new Hashtable();
20
21  customerDictionary.Add(customerFoo.ID, customerFoo);
22  customerDictionary.Add(customerBar.ID, customerBar);
23
24  customerHashtable.Add(customerFoo.ID, customerFoo);
25  customerHashtable.Add(customerBar.ID, customerBar);
26
27  Serialize<IDictionary<Guid, Customer>>(customerDictionary,
28  @"e:\customers.dictionary.xml");
29  Serialize<Hashtable>(customerHashtable,
@"e:\customers.hashtable.xml", new
30  List<Type>{typeof(Customer)});
```

先来看看 IDictionary 对象经过序列化会产生怎样的 XML。从下面的 XML，我们可以总结出相应的序列化规则。

根节点名称为 ArrayOfKeyValueOfguidCustomer2af2CULK，原因很简单。IDictionary 的集合元素类型是 KeyValuePair，按照基于泛型数据契约的命名方式，须要加上泛型数据

契约的名称和泛型类型的哈希值以解决命名冲突，所以 `KeyValueOfguidCustomer2af2CULK` 与 `KeyValuePair` 类型相匹配，作为 `KeyValuePair` 的集合，`IDictionary` 的名称自然就是 `ArrayOfKeyValueOfguidCustomer2af2CULK` 了；

每个元素名称为 `KeyValueOfguidCustomer2af2CULK`，是一个 Key 和 Value 节点的组合。Key 和 Value 内容按照相应类型数据契约的定义进行序列化。

```
31 <ArrayOfKeyValueOfguidCustomer2af2CULK xmlns:i="
http://www.w3.org/2001/
32   XMLSchema-instance" xmlns="http://schemas.
microsoft.com/2003/10/
33   Serialization/Arrays">
34   <KeyValueOfguidCustomer2af2CULK>
35       <Key>a2718c6f-fce4-46df-909b-64a62d30387b</< SPAN>Key>
36       <Value xmlns:d3p1="http://www.artech.com/">
37           <d3p1:ID>a2718c6f-fce4-46df-909b-
64a62d30387b</< SPAN>d3p1:ID>
38           <d3p1:Name>Foo</< SPAN>d3p1:Name>
39           <d3p1:Phone>8888-88888888</< SPAN>d3p1:Phone>
40           <d3p1:CompanyAddress>#9981, West
Sichuan Rd, Xian Shanxi
41           Province</< SPAN>d3p1:CompanyAddress>
42       </< SPAN>Value>
43   </< SPAN>KeyValueOfguidCustomer2af2CULK>
44   <KeyValueOfguidCustomer2af2CULK>
45       <Key>0f1defe1-59a7-447e-96dc-03b4ae4ba31c</< SPAN>Key>
46       <Value xmlns:d3p1="http://www.artech.com/">
47           <d3p1:ID>0f1defe1-59a7-447e-96dc-
03b4ae4ba31c</< SPAN>d3p1:ID>
48           <d3p1:Name>Bar</< SPAN>d3p1:Name>
49           <d3p1:Phone>9999-99999999</< SPAN>d3p1:Phone>
50           <d3p1:CompanyAddress>#3721, Taishan Rd, Jinan ShanDong
Province</< SPAN>d3p1:CompanyAddress>
51       </< SPAN>Value>
52   </< SPAN>KeyValueOfguidCustomer2af2CULK>
53 </< SPAN>ArrayOfKeyValueOfguidCustomer2af2CULK>
54
```

再来看看 `Hashtable` 对象被序列化后的 XML。从下面的 XML 中可以看出，由于 `Hashtable` 与 `IDictionary` 是同一数据在 CLR 类型上的不同表现形式，所以它们最终序列化出来的结构都是一样的，不同的仅仅是根节点与集合元素节点的命名而已。由于 `Hashtable` 元素的 Key 和 Value 没有类型限制，所以将根节点和元素节点的名称转换为 `ArrayOfKeyValueOfanyTypeanyType` 和 `KeyValueOfanyTypeanyType`，这是很好理解的。

```
55 <ArrayOfKeyValueOfanyTypeanyType xmlns:i="http://www.w3.org/2001/
```

```

56 XMLSchema-instance" xmlns="http://schemas.microsoft.com/2003/10/
57 Serialization/Arrays">
58     <KeyValueOfanyTypeanyType>
59         <Key xmlns:d3p1="http://schemas.microsoft.com/2003/10/
60             Serialization/" i:type="d3p1:guid">a2718c6f-fce4-46df-909b-
61             64a62d30387b</< SPAN>Key>
62                 <Value xmlns:d3p1="http://www.artech.com"
i:type="d3p1:Customer">
63                     <d3p1:ID>a2718c6f-fce4-46df-909b-64a62d30387b</<
SPAN>d3p1:ID>
64                     <d3p1:Name>Foo</< SPAN>d3p1:Name>
65                     <d3p1:Phone>8888-88888888</< SPAN>d3p1:Phone>
66                     <d3p1:CompanyAddress>#9981, West Sichuan Rd, Xian Shanxi
67                         Province</< SPAN>d3p1:CompanyAddress>
68                     </< SPAN>Value>
69                 </< SPAN>KeyValueOfanyTypeanyType>
70             <KeyValueOfanyTypeanyType>
71                 <Key xmlns:d3p1="http://schemas.microsoft.com/2003/10/
72                     Serialization/" i:type="d3p1:guid">0f1defe1-59a7-447e-96dc-
73                     03b4ae4ba31c</< SPAN>Key>
74                     <Value xmlns:d3p1="http://www.artech.com"
i:type="d3p1:Customer">
75                         <d3p1:ID>0f1defe1-59a7-447e-96dc-03b4ae4ba31c</<
SPAN>d3p1:ID>
76                         <d3p1:Name>Bar</< SPAN>d3p1:Name>
77                         <d3p1:Phone>9999-99999999</< SPAN>d3p1:Phone>
78                         <d3p1:CompanyAddress>#3721, Taishan Rd, Jinan ShanDong
79                             Province</< SPAN>d3p1:CompanyAddress>
80                         </< SPAN>Value>
81                     </< SPAN>KeyValueOfanyTypeanyType>
82 </< SPAN>ArrayOfKeyValueOfanyTypeanyType>

```

#### 5.4.3 IDictionary 与 Hashtable (2)

从 WCF 的序列化来讲，所有的集合类型都可以看成是数组，无论是上面介绍的 IEnumerable、IEnumerable、IList、IList，还是现在介绍的 Hashtable 和 IDictionary，最终序列化的都是 ArrayOfXxx。不过与其他集合类型不同的是，对于服务契约定义，如果操作参数类型为 Hashtable 和 IDictionary，最终在客户端导入的不再是数组，而是 IDictionary，不过前者对应的永远是 IDictionary，后者的泛型参数类型可以被成功导入。比如下面两段代码片断就是相同的服务契约在定义和导入时表现出来的不同形态。

```

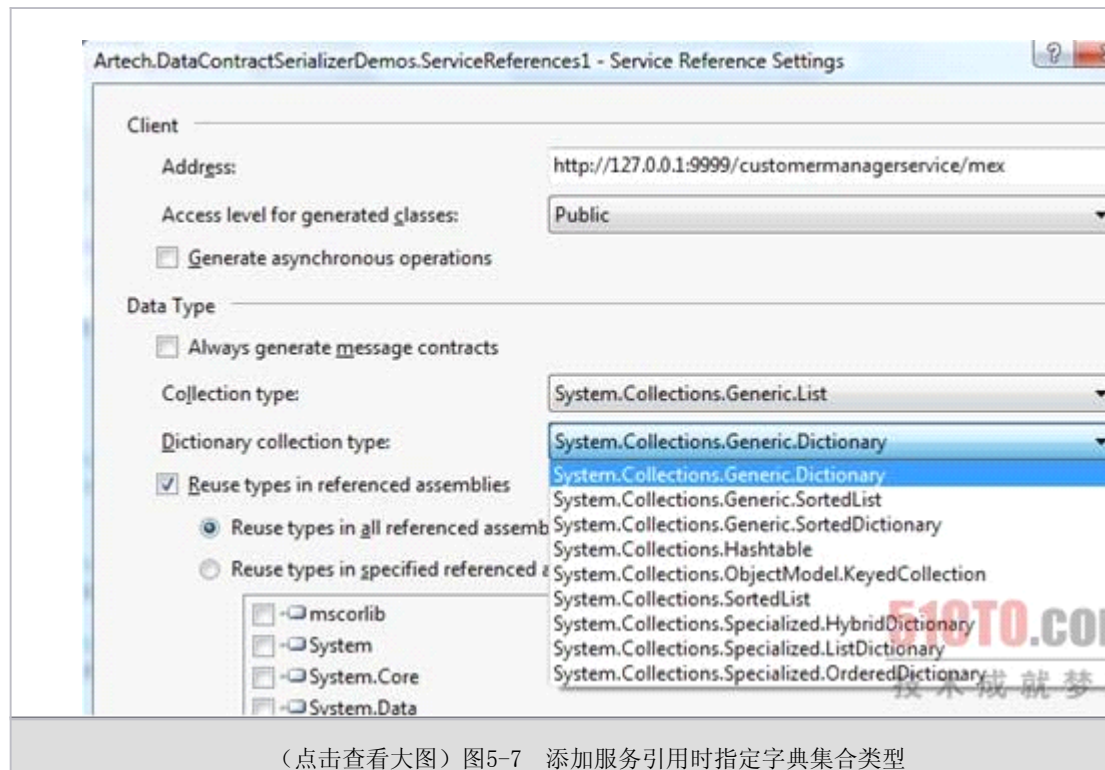
2    [ServiceKnownType(typeof(Customer))]
3    public interface ICustomerManager
4    {
5        [OperationContract]
6        void AddCustomerHashtable(Hashtable customers);
7        [OperationContract]
8        void AddCustomerDictionary(IDictionary<Guid,
Customer> customers);
9    }
10   [ServiceContract]
11   [ServiceKnownType(typeof(Customer))]
12   public interface ICustomerManager
13   {
14       [OperationContract]
15       void AddCustomerHashtable(Dictionary<object,
object> customers);
16       [OperationContract]
17       void AddCustomerDictionary(Dictionary<Guid,
Customer> customers);
18   }

```

在通过 VS 添加服务引用的时候，对于一般的集合类型，可以通过相关服务引用的设置，选择你希望生成的集合类型，对于基于字典类型的集合，VS 同样提供了这样的设置。如图5-7所示，对于字典集合类型，我们具有更多的选择。

### 自定义字典数据契约类型

在5.3节中，我们通过 `CollectionDataContractAttribute` 特性实现了自定义集合数据契约，同样可以通过该特性自定义字典类型的集合数据契约。在下面的例子中，定义了一个直接继承了 `Dictionary` 类型的数据契约。并通过 `CollectionDataContractAttribute` 的 `ItemName`、`KeyName` 和 `ValueName` 属性定义了集合元素的名称，以及集合元素 `Key` 和 `Value` 的名称。



```

19 namespace Artech.DataContractSerializerDemos
20 {
21     [CollectionDataContract (Name="CustomerCollection",
Namespace="
22         http://www.artech.com/", ItemName="CustomerEntry", KeyName =
23         "CustomerID", ValueName="Customer")]
24     public class CustomerDictionary:Dictionary<Guid, Customer>
25     {
26     }
27 }

```

如果通过下面的代码对 CustomerDictionary 对象进行序列化，最终生成的 XML 将会如下面所示。

```

28 CustomerDictionary customers = new CustomerDictionary();
29 Customer customerFoo = new Customer
30 {
31     ID = Guid.NewGuid(),
32     Name = "Foo",
33     Phone = "8888-88888888",
34     CompanyAddress = "#9981, West Sichuan
Rd, Xian Shanxi Province"
35 };
36
37 Customer customerBar = new Customer
38 {

```

```

39         ID             = Guid.NewGuid(),
40         Name            = "Bar",
41         Phone            = "9999-999999999",
42         CompanyAddress   = "#3721, Taishan Rd,
Jinan ShanDong Province"
43     };
44     customers.Add(customerFoo.ID, customerFoo);
45     customers.Add(customerBar.ID, customerBar);
46     Serialize<CustomerDictionary>(customers, @"e:\customers.xml");
47     <CustomerCollection xmlns:i="http://www.w3.org/
2001/XMLSchema-instance"
48         xmlns="http://www.artech.com">
49
50         <CustomerEntry>
51             <CustomerID>38b5ebb3-654d-4100-b4fc-
8614a952b836</SPAN>CustomerID>
52             <Customer>
53                 <ID>38b5ebb3-654d-4100-b4fc-8614a952b836</SPAN>ID>
54                 <Name>Foo</SPAN>Name>
55                 <Phone>8888-88888888</SPAN>Phone>
56                 <CompanyAddress>#9981, West Sichuan Rd, Xian Shanxi
57                     Province</SPAN>CompanyAddress>
58             </SPAN>Customer>
59         </SPAN>CustomerEntry>
60         <CustomerEntry>
61             <CustomerID>cda1f0e3-c10e-4e76-affb-
6dbf5a3b4381</SPAN>CustomerID>
62             <Customer>
63                 <ID>cda1f0e3-c10e-4e76-affb-6dbf5a3b4381</SPAN>ID>
64                 <Name>Bar</SPAN>Name>
65                 <Phone>9999-99999999</SPAN>Phone>
66                 <CompanyAddress>#3721, Taishan Rd, Jinan ShanDong
67                     Province</SPAN>CompanyAddress>
68             </SPAN>Customer>
69         </SPAN>CustomerEntry>
70     </SPAN>CustomerCollection>

```

## 5.5 等效数据契约与数据契约版本控制

数据契约是对用于交换的数据结构的描述，是数据序列化和反序列化的依据。在一个 WCF 应用中，客户端和服务端必须通过等效的数据契约进行有效的数据交换。随着时间的推移，我们会面临着数据契约版本的变化，比如数据成员的添加和删除、成员名称或命名空间的修正等，如何避免数据契约这种版本的变化对客户端现有程序造成影响，就是本节着重讨论的问题。

### 5.5.1 数据契约的等效性

数据契约就是采用一种厂商中立、与平台无关的形式（XSD）定义了数据的结构，而 WCF 通过 `DataContractAttribute` 和 `DataMemberAttribute` 给相应的类型加上了一些元数据，帮助 `DataContractSerializer` 将相应类型的对象序列化成我们希望的 XML 结构。在客户端，WCF 的服务调用并不完全依赖于某个具体的类型，客户端如果具有与服务端完全相同的数据契约类型定义，固然最好。如果客户端现有的数据契约类型与发布出来数据契约有一些差异，仍然可以通过 `DataContractAttribute` 和 `DataMemberAttribute` 这两个特性使该数据契约与之等效。

简言之，如果承载相同数据的两个不同数据契约类型对象最终能够序列化出相同的 XML，那么这两个数据契约就可以看成是等效的数据契约。等效的数据契约具有相同的契约名称、命名空间和数据成员，同时要求数据成员出现的先后次序一致。比如，下面两种形式的数据契约定义，虽然它们的类型和成员命名不一样，甚至对应成员在各自类型中定义的次序都不一样，但是由于合理使用了 `DataContractAttribute` 和 `DataMemberAttribute` 这两个特性，确保了它们的对象最终序列化后具有相同的 XML 结构，所以它们是两个等效的数据契约。

```
1  [DataContract(Namespace = "http://www.artech.com/")]
2  public class Customer
3  {
4      [DataMember(Order=1)]
5      public string FirstName
6      {get;set;}
7
8      [DataMember(Order = 2)]
9      public string LastName
10     { get; set; }
11
12     [DataMember(Order = 3)]
13     public string Gender
14     { get; set; }
15 }
16 [DataContract(Name = "Customer", Namespace = "
http://www.artech.com/")]
17 public class Contact
18 {
19     [DataMember(Name = "LastName", Order = 2)]
20     public string Surname
21     { get; set; }
22
23     [DataMember(Name = "FirstName", Order = 1)]
```



```

24     public string Name
25     { get; set; }
26
27     [DataMember(Name = "Gender", Order = 3)]
28     public string Sex
29     { get; set; }
30 }

```

### 5.5.2 数据成员的添加与删除（1）

接下来，把本节余下的内容用于介绍数据契约的版本控制问题。数据契约版本的差异最主要的表现形式是数据成员的添加和删除。如何保证在数据契约中添加一个新的数据成员，或者是从数据契约中删除一个现有的数据成员的情况下，还能保证现有客户端的正常服务调用（对于服务提供者），或者对现有服务的正常调用（针对服务消费者），这是数据契约版本控制需要解决的问题。

#### 数据成员的添加

先来谈谈添加数据成员的问题，如下面的代码所示，在现有数据契约（CustomerV1）基础上，在服务端添加了一个新的数据成员：Address。但是客户端依然通过数据契约 CustomerV1 进行服务调用。那么，客户端按照 CustomerV1 的定义对于 Customer 对象进行序列化，服务端则按照 CustomerV2 的定义对接收的 XML 进行反序列化，这时会发现缺少 Address 成员。那么在这种数据成员缺失的情况下，DataContractSerializer 又会表现出怎样的序列化与反序列化行为呢？

```

1  [DataContract(Name = "Customer", Namespace = "http://www.artech.com")]
2  public class CustomerV1
3  {
4      [DataMember]
5      public string Name
6      { get; set; }
7
8      [DataMember]
9      public string PhoneNo
10     { get; set; }
11 }
12 [DataContract(Name = "Customer", Namespace = "http://www.artech.com")]
13 public class CustomerV2
14 {
15     [DataMember]
16     public string Name
17     { get; set; }
18
19     [DataMember]

```

```

20     public string PhoneNo
21     { get; set; }
22
23     [DataMember]
24     public string Address
25     { get; set; }
26 }

```

为了探求 DataContractSerializer 在数据成员缺失的情况下如何进行序列化与反序列化，我写了下面一个辅助方法 Deserialize<T>用于反序列化工作。

```

27 public static T Deserialize<T>(string fileName)
28 {
29     DataContractSerializer serializer = new DataContractSerializer
30         (typeof(T));
31     using (XmlReader reader = new XmlTextReader(fileName))
32     {
33         return (T)serializer.ReadObject(reader);
34     }
35 }

```

通过下面的代码来模拟 DataContractSerializer 在 XML 缺少了数据成员 Address 时能否正常的反序列化：先将创建的 CustomerV1 对象序列化到一个 XML 文件中，然后读取该文件，按照 CustomerV2 的定义进行反序列化。从运行的结果可以得知，在数据成员缺失的情况下，反序列化依然可以顺利进行，只是会保留 Address 属性的默认值。

```

36 string fileName = @"e:\customer.xml";
37 CustomerV1 customerV1 = new CustomerV1
38 {
39     Name      = "Foo",
40     PhoneNo   = "9999-99999999"
41 };
42 Serialize<CustomerV1>(customerV1, fileName);
43
44 CustomerV2 customerV2 = Deserialize<CustomerV2>(fileName);
45 Console.WriteLine("customerV2.Name: {0}\ncustomerV2.PhoneNo:
46     {1}\ncustomerV2.Address: {2}",
47     customerV2.Name ?? "Empty", customerV2.PhoneNo ?? "Empty",
48     customerV2.Address ?? "Empty");

```

输出结果：

```

49 customerV2.Name: Foo
50 customerV2.Phone: 9999-99999999
51 customerV2.Address: Empty

```

如果我们从数据契约的另外一种表现形式（XSD）来解释这种序列化和反序列化行为，就会更加容易理解。下面是数据契约 CustomerV2 通过 XSD 的表示，从中可以看出对于表示数据成员的每一个 XML 元素，其 minOccurs 属性为“0”，这意味着所有的成员都是可以默认的。由于基于 CustomerV1 对象序列化后的 XML 依然符合基于 CustomerV2 的 XSD，所以能够确保反序列化的正常进行。

```
52 <?xml version="1.0" encoding="utf-8"?>
53 <xs:schema elementFormDefault="qualified" targetNamespace="http://www.
54   artech.com" xmlns:xs="http://www.w3.org/
2001/XMLSchema" xmlns:tns=
55   "http://www.artech.com">
56   <xs:complexType name="Customer">
57     <xs:sequence>
58       <xs:element minOccurs="0" name="
Address" nillable="true"
59         type="xs:string"/>
60       <xs:element minOccurs="0" name="
Name" nillable="true"
61         type="xs:string"/>
62       <xs:element minOccurs="0" name="
PhoneNo" nillable="true"
63         type="xs:string"/>
64     </xs:sequence>
65   </xs:complexType>
66   <xs:element name="Customer" nillable="
true" type="tns:Customer"/>
67 </xs:schema>
```

在很多情况下，要对这些缺失的成员设置一些默认值。我们可以通过注册序列化回调方法的方式来初始化这些值。WCF 允许通过自定义特性的方式注册序列化的回调方法，这些 DataContractSerializer 在进行序列化或反序列化过程中，会回调你注册的回调方法。WCF 中定义了 4 个这样的特性：OnSerializingAttribute、OnSerializedAttribute、OnDeserializingAttribute 和 OnDeserializedAttribute，相应的回调方法分别会在序列化之前、之后，以及反序列化之前、之后调用。

注：上面 4 个特性只能用在方法上面，而且方法必须具有这样的签名：void DoSomething(StreamingContext context)，即返回类型为 void，具有唯一一个 StreamingContext 类型参数。

比如在下面的代码中，通过一个应用了 OnDeserializingAttribute 特性的方法，为缺失成员 Address 指定了一个默认值。

```
68 [DataContract(Name = "Customer", Namespace = "http://www.artech.com")]
```

```

69 public class CustomerV2
70 {
71     //其他成员
72     [OnDeserializing]
73     void OnDeserializing(StreamingContext context)
74     {
75         this.Address = "Temp Address...";
76     }
77 }

```

### 5.5.2 数据成员的添加与删除（2）

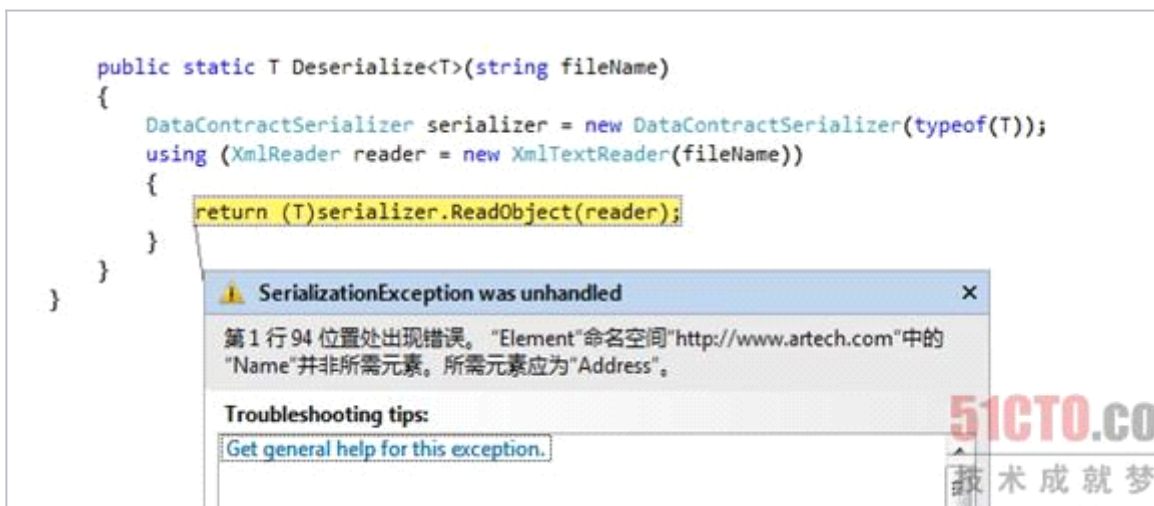
但是对于那些必备数据成员（DataMemberAttribute 特性的 IsRequired 属性为 true）缺失的情况，还能够保证正常的序列化与反序列化吗？

```

1  [DataContract(Name = "Customer", Namespace = "http://www.artech.com")]
2  public class CustomerV2
3  {
4      //其他成员
5      [DataMember(IsRequired =true)]
6      public string Address
7      { get; set; }
8  }

```

在上面的代码中，通过 DataMemberAttribute 的 IsRequired 属性将 Address 定义成数据契约的必备数据成员。如果我们运行上面的程序，将会抛出如图 5-8 所示的 SerializationException 异常，提示找不到 Address 元素。



（点击查看大图）图5-8 缺少必须数据成员导致反序列化异常

对于上面的异常，仍然可以从 XSD 找原因。下面是包含必备成员 Address 的数据契约在 XSD 中的表示。我们可以清楚地看到 Address 元素的 minOccurs="0"没有了，表明该元素是不能

缺失的。由于 XML 不再符合 XSD 的定义，反序列化不能成功进行。

```
9  <?xml version="1.0" encoding="utf-8"?>
10 <xs:schema elementFormDefault="qualified" targetNamespace="http://www.
11   artech.com" xmlns:xs="http://www.w3.org/2001
12   /XMLSchema" xmlns:tns=
13     "http://www.artech.com">
14   <xs:complexType name="Customer">
15     <xs:sequence>
16       <xs:element name="Address" nillable="
17         true" type="xs:string"/>
18       <xs:element minOccurs="0" name="Name" nillable="true"
19         type="xs:string"/>
20       <xs:element minOccurs="0" name="PhoneNo" nillable="true"
21         type="xs:string"/>
22     </xs:sequence>
23   </xs:complexType>
24   <xs:element name="Customer" nillable="true" type="tns:Customer"/>
25 </xs:schema>
```

### 数据成员的删除

讨论了数据成员添加的情况，接着讨论数据成员删除的情况。依然沿用 Customer 数据契约的例子，在这里，两个版本需要做一下转变：CustomerV1中定义了3个数据成员，在 CustomerV2 中数据成员 Address 从成员列表中移除。如果DataContractSerializer 按照 CustomerV2的定义对 CustomerV1的对象进行序列化，那么 XML 中将不会包含 Address 成员；同理，如果 DataContractSerializer 按照 CustomerV2的定义反序列化基于 CustomerV1的 XML，则仍然能够正常创建 CustomerV2对象，因为 CustomerV2的所有成员都存在于 XML 中。

```
24 [DataContract(Name = "Customer", Namespace = "http://www.artech.com")]
25 public class CustomerV1
26 {
27     [DataMember]
28     public string Name
29     { get; set; }
30
31     [DataMember]
32     public string PhoneNo
33     { get; set; }
34
35     [DataMember]
36     public string Address
37     { get; set; }
38
39 }
```

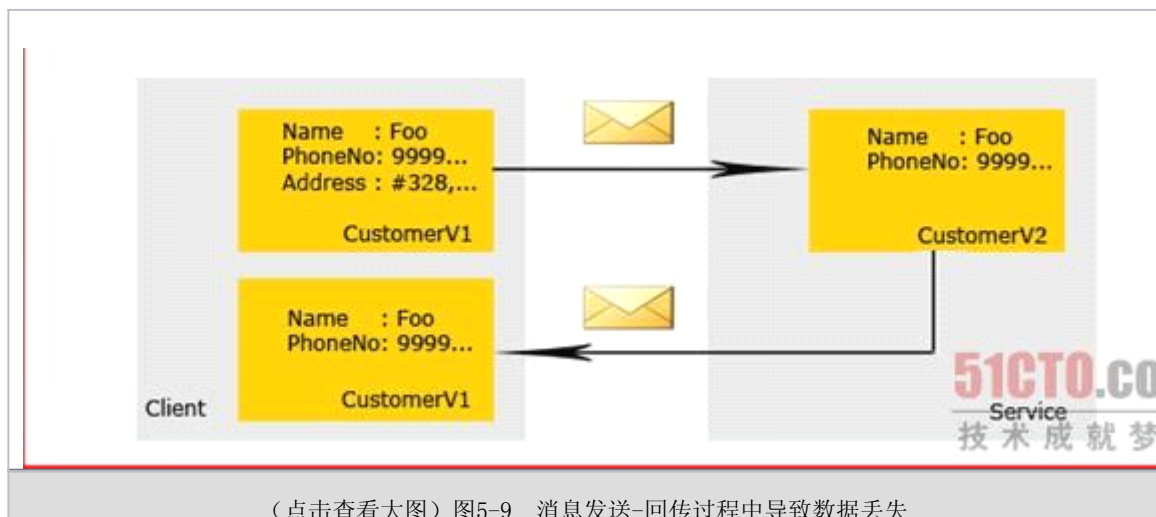
```

40 [DataContract(Name = "Customer", Namespace = "http://www.artech.com")]
41 public class CustomerV2
42 {
43     [DataMember]
44     public string Name
45     { get; set; }
46
47     [DataMember]
48     public string PhoneNo
49     { get; set; }
50 }

```

### 5.5.2 数据成员的添加与删除 (3)

在这里着重讨论的是由于数据契约成员的移除导致在发送-回传 (Round Trip) 过程中数据的丢失问题。如图5-9所示，客户端基于数据契约 CustomerV1进行服务调用，而服务的实现却是基于 CustomerV2的。那么序列化的 CustomerV1对象生成的 XML 通过消息传到服务端，服务端会按照 CustomerV2进行反序列化，毫无疑问 Address 的数据会被丢弃。如果 Customer 的信息需要返回到客户端，服务需要对 CustomerV2对象进行序列化，则序列化生成的 XML 肯定已无 Address 数据成员存在。当回复消息返回到客户端时，客户端按照 CustomerV1进行反序列化生成 CustomerV1对象，会发现原本赋了值的 Address 属性现在变成 null 了。对于客户端来说，这是一件很奇怪、也是不可接受的事情：“为何数据经过发送-回传后会无缘无故丢失呢？”



(点击查看大图) 图5-9 消息发送-回传过程中导致数据丢失

为了解决这类问题，WCF 定义了一个特殊的接口 `System.Runtime.Serialization.IExtensibleDataObject`，`IExtensibleDataObject` 中仅仅定义了一个 `ExtensionDataObject` 类型属性成员。对于实现了 `IExtensibleDataObject` 的数据契约，`DataContractSerializer` 在进行序列化时会把 `ExtensionData` 属性的值也序列化到 XML 中；在反序列化过程中，如果发现 XML 包含有数据契约中没有的数据，会将多余的数据进行反序列化，并将其放入

ExtensionData 属性中保存起来，由此解决数据丢失的问题。

```
1 public interface IExtensibleDataObject
2 {
3     ExtensionDataObject ExtensionData { get; set; }
4 }
```

比如，让 CustomerV2 实现 IExtensibleDataObject 接口。

```
5 [DataContract(Name = "Customer", Namespace
= "http://www.artech.com")]
6 public class CustomerV2 : IExtensibleDataObject
7 {
8     //其他成员
9     public ExtensionDataObject ExtensionData
10     { get; set; }
11 }
```

我们通过下面的程序来演示 IExtensibleDataObject 接口的作用。将 CustomerV1 对象序列化到第一个 XML 文件中，然后读取该文件基于 CustomerV2 进行反序列化创建的 CustomerV2 对象，最后序列化 CustomerV2 对象到第 2 个 XML 文件中。会发现尽管 CustomerV2 没有定义 Address 属性，最终序列化出来的 XML 却包含 Address XML 元素。

```
12 string fileNameV1 = @"e:\customer.v1.xml";
13 string fileNameV2 = @"e:\customer.v2.xml";
14 CustomerV1 customerV1 = new CustomerV1
15 {
16     Name = "Foo",
17     PhoneNo = "9999-99999999",
18     Address="#328, Airport Rd, Industrial Park,
Suzhou Jiangsu Proivnce"
19 };
20 Serialize<CustomerV1>(customerV1, fileNameV1);
21 CustomerV2 customerV2 = Deserialize<CustomerV2>(fileNameV1);
22 Serialize<CustomerV2>(customerV2, fileNameV2);
23 <Customer xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
24     xmlns="http://www.artech.com/">
25     <Address>#328, Airport Rd, Industrial Park, Suzhou Jiangsu
26     Proivnce</Address>
27     <Name>Foo</Name>
28     <PhoneNo>9999-99999999</PhoneNo>
29 </Customer>
```

在介绍 DataContractSerializer 的时候，知道 DataContractSerializer 具有只读的属性 IgnoreExtensionDataObject（该属性在相应的构造函数中指定），它表示对于实现了 IExtensibleDataObject 接口的数据契约，在序列化或反序列化时是否忽略 ExtensionData

属性的值，该属性默认为 false。如果将其设为 true，DataContractSerializer 在反序列化的时候会忽略多余的 XML 元素，而在序列化时会丢弃 ExtensionData 属性中保存的值。

```
30 public sealed class DataContractSerializer :  
    XmlObjectSerializer  
31 {  
32     //其他成员  
33     public bool IgnoreExtensionDataObject { get; }  
34 }
```

对于 WCF 服务，可以通过 ServiceBehaviorAttribute 的 IgnoreExtensionDataObject 设置是否忽略 ExtensionData。如下面的代码所示。

```
35 [AttributeUsage(AttributeTargets.Class)]  
36 public sealed class ServiceBehaviorAttribute :  
    Attribute, IServiceBehavior  
37 {  
38     //其他成员  
39     public bool IgnoreExtensionDataObject { get; set; }  
40 }  
41 [ServiceBehavior(IgnoreExtensionDataObject = true)]  
42 public class CustomerManagerService : ICustomerManager  
43 {  
44     public void AddCustomer(CustomerV2 customer)  
45     {  
46         //省略实现  
47     }  
48 }
```

IgnoreExtensionDataObject 属性同样可以通过配置的方式进行设定。

```
49 <?xml version="1.0" encoding="utf-8" ?>  
50 <configuration>  
51     <system.serviceModel>  
52         <behaviors>  
53             <serviceBehaviors>  
54                 <behavior name="IgnoreExtensionDataBehavior">  
55                                     <dataContractSerializer  
ignoreExtensionDataObject="true" />  
56                 </behavior>  
57             </serviceBehaviors>  
58         </behaviors>  
59         <services>  
60             <service behaviorConfiguration="Ignore  
ExtensionDataBehavior"  
61                 name="Artech.DataContractSerializerDemos.
```



```

62         CustomerManagerService">
63         <endpoint address="http://127.0.0.1:9999/
64             customermanagerservice"
65             binding="basicHttpBinding" bindingConfiguration=""
66             contract="Artech.DataContractSerializerDemos.
67                 ICustomerManager" />
68     </service>
69 </services>
70 </system.serviceModel>
71 </configuration>

```

### 5.5.3 数据契约代理 (Surrogate) (1)

上面谈到的数据契约版本之间的差异都是一些诸如数据成员的添加或删除这样的小差异。如果一个类型，不一定是数据契约，和给定的数据契约具有很大的差异，而我们要将该类型的对象序列化成基于数据契约对应的 XML；或者对于一段给定的基于数据契约的 XML，要通过反序列化生成该类型的对象，又将怎么办呢？比如下面定义了两个类型 Contact 和 Customer，其中 Customer 是数据契约，Contact 的 Sex 属性相当于 Customer 的 Gender 属性，而 Contact 的 FullName 可以看成是 Customer 的 FirstName 和 LastName 的组合。现在我们要做的是将一个 Contact 对象序列化成基于 Customer 数据契约对应的 XML 结构，或者对于一段基于 Customer 数据契约对应结构的 XML，将其反序列化生成 Contact 对象。

```

1  public class Contact
2  {
3      public string FullName
4      { get; set; }
5
6      public string Sex
7      { get; set; }
8
9      public override bool Equals(object obj)
10     {
11         Contact contact = obj as Contact;
12         if (contact == null)
13         {
14             return false;
15         }
16
17         return this.FullName == contact.
18             FullName && this.Sex == contact.Sex;
19     }
20     public override int GetHashCode()
21     {

```

```

22         return this.FullName.GetHashCode()
   ^ this.Sex.GetHashCode();
23     }
24 }
25 [DataContract(Namespace = "http://www.artech.com")]
26 public class Customer
27 {
28     [DataMember(Order = 1)]
29     public string FirstName
30     { get; set; }
31
32     [DataMember(Order = 2)]
33     public string LastName
34     { get; set; }
35
36     [DataMember(Order = 3)]
37     public string Gender
38     { get; set; }
39 }

```

为实现上面的要求，要涉及 WCF 中一个特殊的概念：数据契约代理（DataContract Surrogate）。WCF 通过一个接口 System.Runtime.Serialization.IDataContractSurrogate 来表示数据契约代理。IDataContractSurrogate 用于实现在序列化、反序列化、数据契约的导入和导出过程中对对象或类型的替换。以上面 Contact 和 Customer 为例。在正常的情况下，DataContractSerializer 针对类型 Customer 对一个真正的 Customer 对象进行序列化，现在要求的是通过 DataContractSerializer 序列化一个 Contact 对象，并且要生成与 Customer 等效的 XML，就要在序列化的过程中实现类型的替换（由 Contact 类型替换成 Customer 类型）和对象的替换（由 Contact 对象替换成 Customer 对象）。

先来看看 IDataContractSurrogate 的定义，序列化相关的方法有以下3个，如果想具体了解 IDataContractSurrogate 在数据契约导入、导出，以及在代码生成方面的应用可以参考 MSDN 相关文档，在这里就不多作介绍了。

**GetDataContractType:** 获取用于序列化、反序列化或数据契约导入导出数据契约类型，实现此方法相当于实现了类型的替换。

**GetObjectToSerialize:** 在序列化之前获取序列化的对象，实现了此方法相当于为序列化实现了对象替换。

**GetDeserializedObject:** 当完成反序列化工作时，通过方法获得被反序列化生成的对象，通过此方法可以用新的对象替换掉真正被反序列化生成的原对象。

```

40 public interface IDataContractSurrogate

```

```

41 {
42     Type GetDataContractType(Type type);
43     object GetObjectToSerialize(object obj,
Type targetType);
44     object GetDeserializedObject(object obj,
Type targetType);
45
46     object GetCustomDataToExport(MemberInfo
memberInfo, Type dataContractType);
47     object GetCustomDataToExport(Type clrType,
Type dataContractType);
48     void GetKnownCustomDataTypes(Collection<Type>
customDataTypes);
49     Type GetReferencedTypeOnImport(string typeName,
string typeNamespace,
50     object customData);
51     CodeTypeDeclaration ProcessImportedType(CodeTypeDeclaration
52     typeDeclaration, CodeCompileUnit compileUnit);
53 }

```

现在专门为 Contact 和 Customer 之间的转换创建一个自定义的 DataContractSurrogate: ContractSurrogate。在 GetDataContractType 中, 如果发现类型是 Contact, 则替换成 Customer。在 GetObjectToSerialize 方法中, 将用于序列化的 Contact 对象用 Customer 对象替换, 而在 GetDeserializedObject 中则用 Contact 对象替换反序列化生成的 Customer 对象。

```

54 public class ContractSurrogate : IDataContractSurrogate
55 {
56
57     public object GetCustomDataToExport
(Type clrType, Type dataContractType)
58     {
59         return null;
60     }
61
62     public object GetCustomDataToExport
(MemberInfo memberInfo, Type
63     dataContractType)
64     {
65         return null;
66     }
67
68     public Type GetDataContractType(Type type)
69     {

```

```

70         if (type == typeof(Contact))
71         {
72             return typeof(Customer);
73         }
74
75         return type;
76     }
77
78     public object GetDeserializedObject
79     (object obj, Type targetType)
80     {
81         Customer customer = obj as Customer;
82         if (customer == null)
83         {
84             return obj;
85         }
86
87         return new Contact
88         {
89             FullName = string.Format("{0}
90 {1}", customer.FirstName,
91             customer.LastName),
92             Sex = customer.Gender
93         };
94     }
95
96     public void GetKnownCustomDataTypes
97     (Collection<Type> customDataTypes)
98     {
99     }
100
101     public object GetObjectToSerialize(object obj,
102     Type targetType)
103     {
104         Contact contact = obj as Contact;
105         if (contact == null)
106         {
107             return obj;
108         }
109
110         return new Customer
111         {

```

```

110         FirstName = contact.FullName.Split
(" ").ToCharArray()[0],
111         LastName = contact.FullName.Split
(" ").ToCharArray()[1],
112         Gender = contact.Sex
113     };
114 }
115
116 public Type GetReferencedTypeOnImport
(string typeName, string
117     typeNameSpace, object customData)
118 {
119     return null;
120 }
121
122 public CodeTypeDeclaration ProcessImportedType
(CodeTypeDeclaration
123     typeDeclaration, CodeCompileUnit compileUnit)
124 {
125     return typeDeclaration;
126 }
127 }

```

### 5.5.3 数据契约代理 (Surrogate) (2)

为了演示 ContractSurrogate 在序列化和反序列化中所起的作用, 创建了 Serialize<T> 和 Deserialize<T> 方法, 通过创建 DataContractSerializer 进行序列化和反序列化。方法中的 dataContractSurrogate 参数被传入 DataContractSerializer 的构造函数中。

```

1 public static void Serialize<T>(T instance, string fileName,
2     IDataContractSurrogate dataContractSurrogate)
3 {
4     DataContractSerializer serializer = new
DataContractSerializer(typeof(T),
5         null, int.MaxValue, false, false,
dataContractSurrogate);
6     using (XmlWriter writer = new XmlText
Writer(fileName, Encoding.UTF8))
7     {
8         serializer.WriteObject(writer, instance);
9         Process.Start(fileName);
10    }
11 }
12
13 public static T Deserialize<T>(string fileName,

```

```

IDataContractSurrogate
14     dataContractSurrogate)
15 {
16     DataContractSerializer serializer = new
DataContractSerializer(typeof(T),
17         null, int.MaxValue, false, false,
dataContractSurrogate);
18     using (XmlReader reader = new XmlTextReader(fileName))
19     {
20         return (T)serializer.ReadObject(reader);
21     }
22 }

```

借助于上面定义的 ContractSurrogate 和两个辅助方法，我们通过下面的程序演示 IDataContractSurrogate 在序列化和反序列化过程中所起的作用。

```

23 string fileName = @"E:\contact.xml";
24 Contact contactToSerialize = new Contact
25 {
26     FullName    = "Bill Gates",
27     Sex         = "Male"
28 };
29 IDataContractSurrogate dataContractSurrogate
= new ContractSurrogate();
30 Serialize<Contact>(contactToSerialize,
fileName, dataContractSurrogate);
31
32 Contact contactToDeserialize =
Deserialize<Contact>(fileName,
33     dataContractSurrogate);
34 Console.WriteLine("contactToSerialize.
Equals(contactToDeserialize) = {0}",
35     contactToSerialize.Equals(contactToDeserialize) );

```

下面的 XML 表述 Contract 对象被序列化后的结果，显而易见，这和真正序列化一个 Customer 对象是完全一样的。不仅如此，基于下面一段 XML 反序列化生成的 Contact 对象和用于序列化的对象是相等的，这通过最终的输出结果可以看出来。

```

36 <Customer xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
37     xmlns="http://www.artech.com">
38     <FirstName>Bill</FirstName>
39     <LastName>Gates</LastName>
40     <Gender>Male</Gender>
41 </Customer>

```

输出结果:

```
42 contactToSerialize.Equals(contactToDeserialize) = True
```

在进行服务寄宿的时候, 可以通过如下代码指定 IDataContractSurrogate。

```
43 using (ServiceHost serviceHost = new ServiceHost
    (typeof(InventoryCheck)))
44     foreach (ServiceEndpoint ep in serviceHost.
        Description.Endpoints)
45     {
46         foreach (OperationDescription op in ep.
            Contract.Operations)
47         {
48             DataContractSerializerOperationBehavior dataContractBehavior
                =
49                 op.Behaviors.Find<Data
                    ContractSerializerOperationBehavior>()
50                 as DataContractSerializerOperationBehavior;
51             if (op.Behaviors.Find<Data
                ContractSerializerOperationBehavior>()
52                 != null)
53                 dataContractBehavior.DataContractSurrogate = new
54                     ContractSurrogate();
55             op.Behaviors.Add(op.Behaviors.
56                 Find<DataContractSerializerOperationBehavior>());
57
58             dataContractBehavior = new
59                 DataContractSerializerOperationBehavior(op);
60             dataContractBehavior.DataContractSurrogate =
61                 new ContractSurrogate();
62             op.Behaviors.Add(dataContractBehavior);
63         }
64     }
```

## 5.6 序列化 WCF 框架中的实现

以上的内容基本上都是围绕着DataContract的定义介绍DataContractSerializer进行序列化的规则。在默认的情况下, WCF 绝大部分的序列化与反序列化工作都通过DataContractSerializer 承担。但是在整个 WCF 体系结构中, 是如何利用DataContractSerializer进行序列化与反序列化的呢? 这就是接下来要讨论的话题。

### 5.6.1 MessageFormatter

这个 WCF 体系大体分为两部分, 客户端和服务端。由于消息是 WCF 通信的唯一媒介, 但

是无论是服务的调用，还是服务的实现都通过方法调用（Method Call）来实现的，所以 WCF 必须实现消息和方法调用之间的转换。WCF 客户端框架要将服务方法调用转化成请求消息，并将回复消息转换成方法调用的返回值，或者引用、输出参数；WCF 服务端框架则要将接收到的请求消息转换成相应服务操作方法的调用，并将方法执行的结果（返回值，或者引用、输出参数）转换成回复消息。

要实现这样的转换，序列化和反序列化是一个重要环节。在客户端，要实现对输入参数（包括引用参数）的序列化，以及对返回值（包括引用参数和输出参数）的反序列化；在服务端则与此相反，即要实现对输入参数的反序列化和对返回值的序列化。WCF 通过一组特殊的对象提供对序列化和反序列化的实现：MessageFormatter。根据在客户端和服务端所起的不同作用，客户端的 MessageFormatter 又称为 ClientMessageFormatter，服务端则称为 DispatchMessageFormatter。

所有的 ClientMessageFormatter 实现同一个接口 System.ServiceModel.Dispatcher.IClientMessageFormatter，IClientMessageFormatter 定义了如下两个方法成员。SerializeRequest 用于将输入参数序列化到 Message 对象中。而 DeserializeReply 则将回复消息反序列化成相应的返回值或 ref、out 参数。

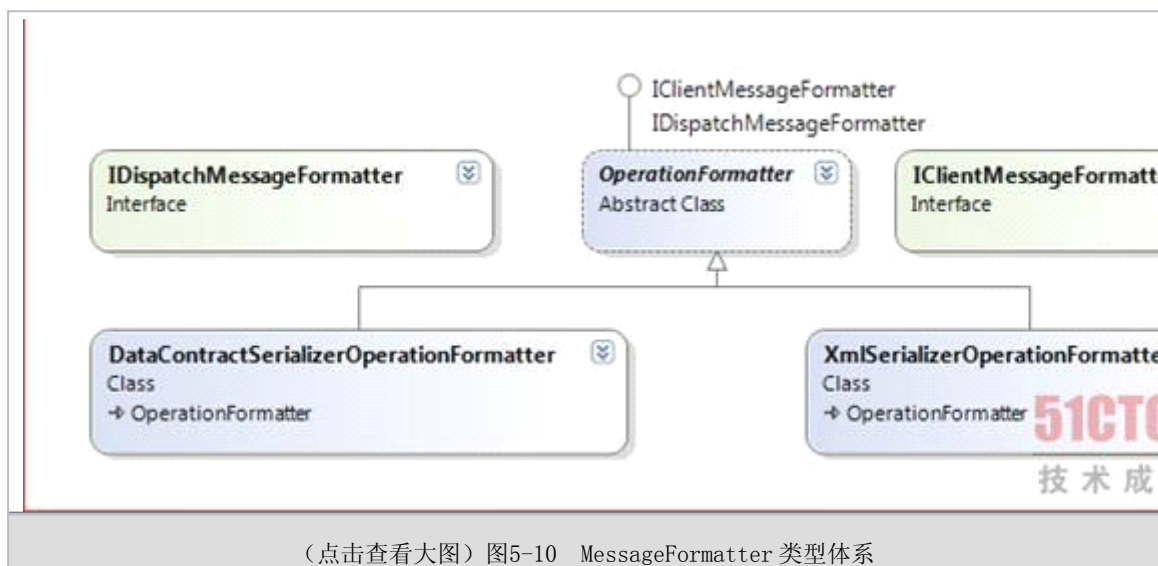
```
1 public interface IClientMessageFormatter
2 {
3     Message SerializeRequest(MessageVersion
messageVersion, object[]
4         parameters);
5
6     object DeserializeReply(Message message,
object[] parameters);
7 }
```

DispatchMessageFormatter 也具有相应的接口：System.ServiceModel.Dispatcher.IDispatchMessageFormatter。IDispatchMessageFormatter 定义的两个方法成员正好与 IClientMessageFormatter 相对。DeserializeRequest 方法将请求消息反序列化成输入参数，而 SerializeReply 方法则将方法返回值或 ref、out 参数序列化到回复消息中。

```
8 public interface IDispatchMessageFormatter
9 {
10     void DeserializeRequest(Message message,
object[] parameters);
11     Message SerializeReply(MessageVersion
messageVersion, object[] parameters,
12         object result);
13 }
```



WCF 定义了一个抽象类 `OperationFormatter`，专门用于客户端和服务端基于操作调用的消息格式化工作，`OperationFormatter` 同时实现了 `IClientMessageFormatter` 和 `IDispatchMessageFormatter` 两个接口。具体的格式化功能实现在两个具体的 `MessageFormmatter` 类型上：`DataContractSerializerOperationFormatter` 和 `XmlSerializerOperationFormatter` 上，它们继承自 `OperationFormatter`。从名称就可猜出来，`DataContractSerializerOperationFormatter` 利用 `DataContractSerializer` 进行序列化与反序列化，`XmlSerializerOperationFormatter` 使用的序列化器则是 `XmlSerializer`。使用 `DataContractSerializerOperationFormatter` 是默认选项。具体的结构通过如图5-10所示的类图表示。



### 5.6.2 MessageFormatter 在 WCF 框架中的应用

有了对 `MessageFormmatter` 的了解，我们进一步来看 `MessageFormmatter` 在 WCF 体系整个处理流程中是如何被使用的。图5-11基本上反映了 `MessageFormmatter` 在整个体系中的位置。客户端对操作方法的调用被 `ClientMessageFormmatter` 截获，调用 `SerializeRequest` 方法将输入参数列表序列化成 XML，放入请求消息中；服务端在成功接收到请求消息后，通过 `InstanceProvider` 创建或获取服务实例，在通过 `OperationSelector` 选择出具体的操作方法后，则被 `DispatchMessageFormatter` 截获，`DispatchMessageFormatter` 调用 `DeserializeRequest` 方法将消息反序列化为输入参数，最终通过 `OperationInvoker` 对象利用反射的原理调用相应的操作方法。如果执行结果需要返回，可将结果传入 `DispatchMessageFormatter`，调用 `SerializeReply` 将其序列化到回复消息中，客户端接收到回复消息，通过调用 `ClientMessageFormmatter` 的 `DeserializeReply` 方法将消息反序列化成方法的返回值或 `ref/out` 参数对象。

## 第6章 消息、消息契约与消息编码

(Message, Message Contract and Message Encoding)

消息交换是 WCF 进行通信的唯一手段，通过方法调用（Method Call）形式体现的服务访问需要转化成具体的消息，并通过相应的编码才能经传输通道发送到服务端；服务操作执行的结果也只能以消息的形式才能被正常地返回到客户端。所以，消息在整个 WCF 体系结构中处于一个核心的地位，WCF 可以看成是一个消息处理的管道。

尽管消息在整个 WCF 体系中具有如此重要的意义，可是一般的 WCF 编程人员，却意识不到消息的存在。原因很简单，WCF 设计的目标就是实现消息通信的所有细节，为最终的编程人员提供一个完全面向对象的编程模型。所以对于一般的编程人员来说，他们面对的是接口，却不知道服务契约对于服务的描述；面对的是数据类型，却不知道数据契约对序列化的作用；面对的是方法调用和返回值的获取，却不了解底层消息交换的过程。

鼓励大家深入了解 WCF 关于消息处理的流程有两个目的：第一，只有在对整个消息处理流程具有清晰认识的基础上才能写出高质量的 WCF 程序；第二，WCF 是一个极具可扩展性的通信框架，可以灵活地创建一些自定义 WCF 扩展（WCF Extension）以实现你所需要的功能。如同 WCF 的插件一样，这些自定义的 WCF 扩展以即插即用的方式参与到 WCF 整个消息处理流程之中。了解 WCF 整个消息处理流程是灵活进行 WCF 扩展的前提。

## 6.1 SOAP 与 WS-Addressing

从本质上讲，消息就是基于某种标准化的结构对于数据的封装。从数据的表现形式来看，XML 由于具有强大的表意能力和与平台无关的特性，已经成为了事实上的数据表现和传输的标准，所以 WCF 在设计之初就是以 XML 为中心的。但是随着近年来 AJAX 的盛行，WCF 的消息为另一种非 XML 格式的数据表现形式也提供了支持，那就是 JSON（JavaScript Object Notation）。

对于基于 XML 的消息来说，SOAP 是一种主要的表现形式，随着 SOAP 规范（SOAP Spec 1.1 和 1.2）和建立在 SOAP 规范上的 WS-\* 标准的不断完善，以及 SOA 不断深入人心，SOAP 已经成为一种被业界普遍接受和使用的消息交换协议。但是近两年来，REST 已经成了挑战 SOAP 的一股最大的力量，有人甚至基于 REST 提出一种全新的架构理念，即面向 Web 架构（WOA: Web Oriented Architect）和面向资源架构（ROA: Resource Oriented Architect）。所以除了传统的基于 SOAP 的 Message，WCF 还对单纯的 XML 提供支持（POX: Plain Old XML）。对于非 XML 的 JSON 和非 SOAP 的 POX，SOAP 毕竟还是当今的主流，本书的绝大部分内容都是直接面向 SOAP 的。下面简单介绍一下 SOAP。

### 6.1.1 SOAP（基于 SOAP 1.2 标准）（1）

SOA 强调以消息为中心的应用设计，消息交换是 SOA 最基本的行为，所以需要一种规范对消息的格式进行标准化。对于一个真正企业级的应用，消息交换不仅局限于业务相关数据

的传递，它还承载着一系列非业务功能的实现，比如安全（Security）、可靠消息通信（Reliable Messaging）、事务的流转（Transaction Flow）等。这就为消息的可扩展性提出了一个新的要求，而 SOAP 满足这样的要求。

按照最新的 SOAP 规范（SOAP 1.2）的定义，SOAP 是一个轻量的协议，用以规范在一个分布式环境下进行结构化的信息交换。整个 SOAP 1.2 规范基本上分两个部分：第一部分为消息框架（Messaging Framework），主要介绍基本的 SOAP 处理模型（SOAP Processing Model）、SOAP 的可扩展模型（SOAP Extensibility Model）、SOAP 协议绑定框架（SOAP Protocol Binding Framework）、SOAP 消息的结构（SOAP Message Construct）等；第二部分为附件（Adjunction），用以规定一些在消息框架中漏掉的附属规范，比如 SOAP 数据模型（SOAP Data Model）、SOAP 编码（SOAP Encoding）、SOAP 的 RPC 表示（SOAP RPC Representation）、SOAP 支持的消息交换模式等。接下来会介绍一些与 SOAP 消息密切相关的规范，如果希望对 SOAP 规范有一个全面、深入的了解，可以在 W3C 官方网站下载相关的文档。

### SOAP 封套（SOAP Envelope）

一个 SOAP 消息是一个 XML InfoSet 对象，它的元素、属性、内容字符可以序列化成一个基于 XML 1.0 规范的 XML 片断。SOAP 消息不允许包含任何 XML 处理指令（XML Processing Instruction，比如 `<?xml version="1.0" encoding="utf-8" ?>`）。整个 SOAP 消息被封装到一个称为 SOAP 封套（SOAP Envelope）的 XML 元素中。该封套元素具有如下的要求：

元素名称必须为“Envelope”，命名空间名称为“<http://www.w3.org/2003/05/soap-envelope>”；

有零个或多个命名空间有效属性（Namespace Qualified Attribute）作为元素的属性（XML Attribute）；

该元素只能包含两种类型的元素：可选的报头（Optional Header）和必须的主体（Mandatory Body）。

### SOAP 报头（SOAP Header）

SOAP 消息之所以具有超强的可扩展性，其根本原因在于它具有一个可扩展的报头集合。在消息路由的过程中，无论是消息的最初发送者还是消息的最终接收者，甚至是消息路由的中间结点（Intermediary），都可以添加和修改相应的报头。绝大部分 WS-\* 规范都是通过添加相应的报头得以实现的。SOAP 1.2 为 SOAP 报头定义了如下的规范：

元素名称必须为“Header”，命名空间名称为“<http://www.w3.org/2003/05/soap-envelope>”；

可以包含零个或多个命名空间有效的 XML 元素 (Namespace Qualified Element) 作为报头的子元素;

可以包含零个或者多个命名空间有效的 XML 属性 (Namespace Qualified Attribute) 作为报头元素的属性 (XML Attribute)。

SOAP 1.2还指定了与 SOAP 报头相关的预定义属性 (Attribute), 它们在 SOAP 消息的处理中具有重要的指导性作用。

**SOAP“role”属性**

该属性的名称为“role”，命名空间名称为“<http://www.w3.org/2003/05/soap-envelope>”，类型为“xs:anyURI”。在 SOAP 消息的路由路径中，除了最初的发送者和最终的接收者之外，往往还有一些中间结点 (Intermediary)，我们把这些都统称为 SOAP 结点 (SOAP Node)。不同的结点可以看成是在 SOAP 消息处理过程中的不同的角色 (Role)，该角色通过 SOAP 报头的 role 属性表示。表6-1列出了 SOAP 1.2定义的3种典型的“角色”：next、none 和 ultimateReceiver。

表6-1

名称	全名	描述
next	"http://www.w3.org/2003/05/soap-envelope/role/next"	所有的中间结点和最终接收结点
none	"http://www.w3.org/2003/05/soap-envelope/role/none"	不包括任何结点
ultimateReceiver	"http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver"	最终的接收结点

**SOAP“mustUnderstand”属性**

属性名称为“mustUnderstand”，命名空间名称为“<http://www.w3.org/2003/05/soap-envelope>”，类型为“xs:boolean”。该属性表示目标结点（通过 role 属性指定）是否必须理解并处理相应的 SOAP 报头。输入值为“true”或是“1”，则表明 SOAP 报头必须被目标结点理解并处理，反之，如果是“false”或“0”，则表明目标结点对 SOAP 报头的处理是可选的。

**SOAP 主体 (SOAP Body)**

SOAP 主体是一个 SOAP 消息所必须的，一个 SOAP 消息可以没有报头，但是不能没有主体。SOAP 的主体通过 XML 表示，体现的是 SOAP 消息有效负载。SOAP 1.2 对 SOAP 主体作了如下的规范：

元素名称必须为“Body”，命名空间名称为“<http://www.w3.org/2003/05/soap-envelope>”；

可以包含零个或多个命名空间有效的 XML 元素（Namespace Qualified Element）作为主体的子元素；

可以包含零个或多个命名空间有效的 XML 属性（Namespace Qualified Attribute）作为主体元素的属性（XML Attribute）。

对于 SOAP 主体的子元素的要求，SOAP 1.2 是这样规定的：

应该（不是必须）包含一个命名空间，用以防止命名冲突；

可以包含任意的字符信息项目（比如<subElement>dummy text</subElement>）；

可以包含任意的 XML 元素，每个元素可以具有各自的命名空间；

可以包含任意的 XML 属性。

下面的 XML 表示的就是一个符合 SOAP 1.2 规范的 SOAP 消息。该消息具有一个 SOAP 报头（n:alertcontrol），包含在主体部分的子元素（m:alert）是整个 SOAP 消息的有效负载。

```
1  <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
2      <env:Header>
3          <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
4              <n:priority>1</n:priority>
5              <n:expires>2001-06-22T14:00:00-05:00</n:expires>
6          </n:alertcontrol>
7      </env:Header>
8      <env:Body>
9          <m:alert xmlns:m="http://example.org/alert">
10             <m:msg>Pick up Mary at school at 2pm</m:msg>
11          </m:alert>
12      </env:Body>
13 </env:Envelope>
14 SOAP 错误 (SOAP Fault)
```

### 6.1.1 SOAP（基于 SOAP 1.2 标准）（2）

无论对于何种类型的应用，异常的处理都是不可避免的。不同于一般信息的传递，在一个基于 SOAP 的通信框架中，异常或错误信息是通过错误 SOAP 消息（Fault Message）进行

传递的。比如下面就是一个典型的 Fault 消息，错误信息被封装到 SOAP 主体的子元素 <env:Fault>中。

```
1  <env:Envelope xmlns:env="http://www.w3.org
   /2003/05/soap-envelope"
2      xmlns:m="http://www.
   example.org/timeouts"
3      xmlns:xml="http://
   www.w3.org/XML/1998/namespace">
4      <env:Body>
5          <env:Fault>
6              <env:Code>
7                  <env:Value>env:Sender</env:Value>
8                  <env:Subcode>
9                      <env:Value>m:MessageTimeout</env:Value>
10                     </env:Subcode>
11                 </env:Code>
12                 <env:Reason>
13                     <env:Text xml:lang="en">Sender Timeout</env:Text>
14                 </env:Reason>
15                 <env:Detail>
16                     <m:MaxTime>P5M</m:MaxTime>
17                 </env:Detail>
18             </env:Fault>
19         </env:Body>
20     </env:Envelope>
```

错误 SOAP 消息具有其固有的结构，SOAP1.2对 SOAP Fault 的结构进行了如下的规定：

元素名称必须为 "Fault" ，命名空间名称为 "<http://www.w3.org/2003/05/soap-envelope>"。

Fault 元素中包含如下的子元素：

一个必须的 Code 元素表示错误的代码；

一个必须的 Reason 元素表示错误的原因；

一个可选的 Node 元素表示导致错误的 SOAP 结点；

一个可选的 Role 元素表述导致错误的 SOAP 角色；

一个可选的 Detail 元素表示对错误的消息描述。

### 1. Fault Code 元素

SOAP Fault 的 Code 元素，是一个用以表示错误类型的代码，SOAP 1.2对 Code 元素的格式作了如下的规范：

元素名称必须为"Code"，命名空间名称为"<http://www.w3.org/2003/05/soap-envelope>"。

Code 元素只能先后包含如下两个类型的子元素：

一个必须的 Value 元素用以定义错误代码；

一个可选的 SubCode 元素用以定义错误子代码。

而对于 Value 元素的格式，又具有如下的规范：

元素名称必须为 "Value" ，命名空间名称为 "<http://www.w3.org/2003/05/soap-envelope>"。

元素类型为"env:faultCodeEnum"枚举，表6-2列出了所有的可选枚举值。

表6-2

枚举值	含义
VersionMismatch	命名空间或名称和规定的 SOAP 规范不匹配
MustUnderstand	目标 SOAP 结点不能理解并处理 mustUnderstand 属性为 “true” 或 “1” 的 SOAP 报头
DataEncodingUnknown	SOAP 报头或主体的数据编码方式不被目标 SOAP 结点支持
Sender	消息格式不合法或缺少必要的的数据
Receiver	SOAP 结点处理消息出现错误

而 SubCode 元素相关的规范定义如下：

元素名称必须为 "SubCode"，命名空间名称为 "<http://www.w3.org/2003/05/soap-envelope>"。

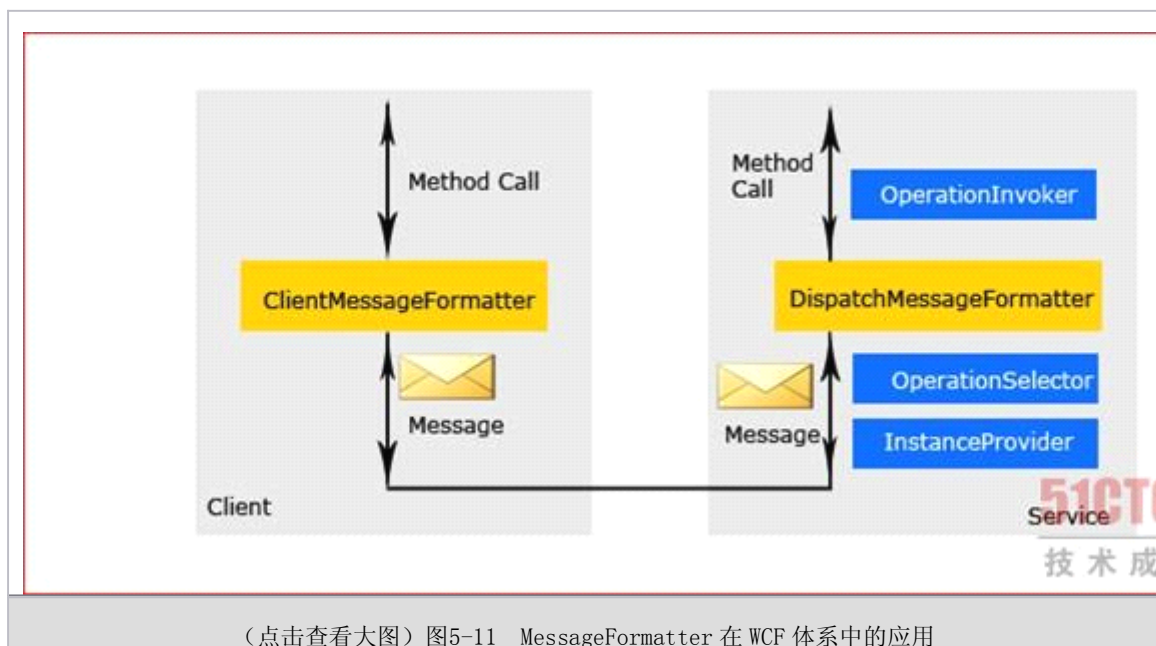
SubCode 元素只能包含以下两种类型的子元素：

必须的 Value 元素：名称为 "Value"，命名空间名称为 "<http://www.w3.org/2003/05/soap-envelope>"，类型为 "xs:QName"，一般将具体应用定义错误代码用作该元素的值；

可选的 Subcode 元素（嵌套）。

比如对于上面给出的 SOAP Fault 消息，就具有如下一个典型的<env:Code>元素。

```
21 <env:Code>
22   <env:Value>env:Sender</env:Value>
23   <env:Subcode>
24     <env:Value>m:MessageTimeout</env:Value>
25   </env:Subcode>
26 </env:Code>
```



### 6.1.1 SOAP（基于 SOAP 1.2标准）（3）

#### 2. Fault Reason 元素

对于一个 SOAP Fault 消息，除了必须有一个表示错误代码的 Code 元素之外，还需要具有一个 Reason 元素用以表示导致错误的原因。SOAP 1.2对 Reason 元素的格式作了如下的规范：

元素名称必须为“SubCode”，命名空间名称为“<http://www.w3.org/2003/05/soap-envelope>”；

包含一个或多个 Text 元素用以描述错误。

比如对于上面给出的 SOAP Fault 消息，就具有如下一个典型的<env:Reason>元素。

```
1 <env:Reason>
2   <env:Text xml:lang="en">Sender Timeout</env:Text>
3 </env:Reason>
```



### 3. Fault Node 元素

由于在整个 SOAP 消息的路由过程中, 错误可能发生在最终接收结点上, 也可能发生在中间结点上。为了使 SOAP Fault 消息的接收者能够判断导致错误的 SOAP 结点类型, 在生成 Fault 消息的时候, 可以通过 Node 元素指定结点的类型。SOAP 1.2对 Node 元素作如下的规范:

元素名称必须为"Node", 命名空间名称为"<http://www.w3.org/2003/05/soap-envelope>";

元素值的类型为"xs:anyURI", 即通过 URI 表示的 SOAP 结点(参考 SOAP 报头的 Role 属性)。

对于一个 SOAP Fault, Node 元素是可选的。在 Node 元素默认的情况下, 暗含的意思是最终接收结点导致错误的发生。换句话说, Node 元素的默认值为"<http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver>". 对于中间结点导致错误的情况, 在生成 SOAP Fault 消息的时候, 需要显示指定 Node 元素的值。

### 4. Fault Role 元素

在介绍 SOAP 报头的时候, 我们说 SOAP 结点在处理 SOAP 消息的过程中担当着不同的角色。SOAP Fault 的 Role 元素即用以表述导致错误的 SOAP 结点对应的角色。SOAP 1.2对 Node 元素的格式作了如下的规范:

元素名称必须为"Role", 命名空间名称为"<http://www.w3.org/2003/05/soap-envelope>";

元素值的类型为"xs:anyURI", 即通过 URI 表示的 SOAP 结点对应的角色(参考 SOAP 报头的 Role 属性)。

### 5. Fault Detail 元素

在很多基于 SOAP 通信的应用中, SOAP Fault 消息的接收者除了须要了解通过上面介绍的基本错误元素表示的错误信息之外, 往往还需要一些对错误信息更加详尽的描述。这样的描述可以通过 Detail 元素来表示。SOAP 1.2对 Detail 元素作了如下的规范:

元素名称必须为"Detail", 命名空间名称为"<http://www.w3.org/2003/05/soap-envelope>";

可以包含任意的 XML 元素, 每个元素可以具有各自的命名空间;

可以包含任意的 XML 属性。

比如对于上面给出的 SOAP Fault 消息，就具有如下一个<env:Detail>元素。

```
4 <env:Detail>
5   <m:MaxTime>P5M</m:MaxTime>
6 </env:Detail>
```

### 6.1.2 WS-Addressing (基于 WS-Addressing 1.0)

对于一个消息通信框架来说，SOAP 规范了消息的格式，解决了如何用消息表示信息的问题；而 WS-寻址 (WS-Addressing) 为基于消息通信的寻址定了规范，接下来就谈论 WS-Addressing 的一些基本规范。

同 SOAP 规范一样，WS-Addressing 也经历了两个主要的版本：WS-Addressing 2004 和 WS-Addressing 1.0。接下来基于 WS-Addressing 的介绍完全是基于 WS-Addressing 1.0。WS-Addressing 提供了一种与传输无关 (Transport-Neutral) 的机制，实现了 Web Service 和 SOAP 消息的寻址问题，它为 Web Service 的地址提供了一套基于 XML InfoSet 的表示，以促进借助消息实现端到端的通信。WS-Addressing 的核心大体可以分为两个部分：终结点引用 (Endpoint Reference) 和消息寻址属性 (Message Addressing Properties)。

#### 终结点引用 (Endpoint Reference)

通过前面章节的介绍，读者应该具有了这样的认识：消息交换的参与者是终结点 (Endpoint)，消息从一个终结点发送到另一个终结点。所以要进行正常的消息交换，须要解决的是终结点引用的问题。WS-Addressing 1.0 对一个终结点引用 (Endpoint Reference) 作了相应的规范，规定一个终结点引用有如下3种类型的属性 (Property) 构成：

**Address:** 每个终结点引用必须包含一个 (也是唯一一个) 通过绝对 URI 表示的地址，该地址即为终结点的地址。终结点地址可以是一个真实存在的物理地址，也可以是一个不存在的逻辑地址。

**Reference Parameters:** 每一个终结点引用可以包含零到多个引用参数。每个引用参数可以看成是对终结点的辅助性描述，以利于更好地进行终结点交互。引用参数列表具有次序无关性，也就是说，不管引用参数在列表的次序如何，只要具有相同的成员，两个引用参数列表就是等效的。每个引用参数最终会体现在 SOAP 消息中，至于引用参数和 SOAP 的绑定问题，定义在 WS-Addressing SOAP 绑定规范中。

**Metadata:** 每一个终结点引用可以包含零到多个元数据，元数据用以描述终结点的行为 (Behavior)、策略 (Policy) 和能力 (Capability)。

基于上面介绍的终结点引用的三元素，一个终结点引用可以通过下面的 XML InfoSet 表示，其中 wsa 表示的命名空间是：“<http://www.w3.org/2005/08/addressing>”。

```
1 <wsa:EndpointReference>
```

```

2      <wsa:Address>xs:anyURI</wsa:Address>
3      <wsa:ReferenceParameters>xs:any*</wsa:ReferenceParameters>
4      <wsa:Metadata>xs:any*</wsa:Metadata>
5  </wsa:EndpointReference>

```

消息寻址属性 (Message Addressing Properties)

为了保障正常的消息交换,我们往往需要一些必要的寻址方面的信息,用以定位相应的终结点。比如请求消息发送的原终结点地址,回复消息发送的目标终点地址,错误消息发送的目标终结点地址等。WS-Addressing 为之定义了一系列的消息寻址方面的属性:

目标终结点地址 (Destination Endpoint Address): (必须) 以一个绝对 URI 的形式表示消息发送的目标终结点地址。

原终结点地址 (Source Endpoint Address): (可选) 表示发送消息的终结点地址。

回复终结点地址 (Reply Endpoint Address): (可选) 表示回复消息的目标终结点地址。

Action: (必须) 用以唯一标识消息体现的意图 (Intend)。

Message ID: (可选) 消息的唯一标识。

Relationship: (可选) 对于 Message ID 的引用,用以实现两个消息之间的匹配问题。

基于上面介绍的消息寻址属性,一个消息的寻址属性可以通过下面的 XML InfoSet 表示,其中 wsa 表示的命名空间是 "<http://www.w3.org/2005/08/addressing>"。

```

6  <wsa:To>xs:anyURI</wsa:To>
7  <wsa:From>wsa:EndpointReferenceType</wsa:From>
8  <wsa:ReplyTo>wsa:EndpointReferenceType</wsa:ReplyTo>
9  <wsa:FaultTo>wsa:EndpointReferenceType</wsa:FaultTo>
10 <wsa:Action>xs:anyURI</wsa:Action>
11 <wsa:MessageID>xs:anyURI</wsa:MessageID>
12 <wsa:RelatesTo RelationshipType="xs:anyURI"?>xs:anyURI</wsa:RelatesTo>
<wsa:
ReferenceParameters>xs:any*</wsa:ReferenceParameters>

```

当消息寻址属性绑定一个具体的 SOAP 消息时,这些属性分别对应一个 SOAP 报头。在下面的 SOAP 消息中, <wsa:MessageID>、<wsa:ReplyTo>、<wsa:To>和<wsa:Action>报头分别对应着上面介绍的 Message ID、回复终结点地址、目的终结点地址和 Action。

```

13 <S:Envelope xmlns:S="http://www.w3.org/
2003/05/soap-envelope"
14   xmlns:wsa="http://www.w3.org/2005/08/addressing">

```

```

15      <S:Header>
16          <wsa:MessageID>http://example.
com/6B29FC40-CA47-1067-B31D-
17          00DD010662DA</wsa:MessageID>
18          <wsa:ReplyTo> <wsa:Address>http://example.com/business/client1
19              </wsa:Address>
20          </wsa:ReplyTo>
21          <wsa:To>http://example.com/
fabrikam/Purchasing</wsa:To>
22          <wsa:Action>http://example.com/
fabrikam/SubmitPO</wsa:Action>
23      </S:Header>
24      <S:Body>
25          ...省略 SOAP 主体部分...
26      </S:Body>
27 </S:Envelope>

```

## 6.2 消息 (Message)

通过上面一节的介绍，我们知道了消息具有不同的表现形式，既有基于 XML InfoSet 的消息，也有非 XML 的消息（比如基于 JSON 消息）；对于基于 XML InfoSet 的消息，又可以进一步划分为基于 SOAP 的消息和 POX (Plain Old XML)；对于 SOAP 消息来说，由于 SOAP 规范 (SOAP 1.1 和 SOAP 1.2) 和 WS-Addressing 规范有不同版本，因此，SOAP 消息又有很多结构化的差别。

在 WCF 中，定义了一个 `System.ServiceModel.Channels.Message` 类，用以表示这些具有不同表现形态的消息。在本节中，我们会着重来讨论这个 `Message` 类型。首先来介绍消息的版本。

### 6.2.1 消息版本 (Message Version)

由于消息基于不同的格式或结构，不同的格式决定了对消息不同的处理方式，所以对一个消息进行正确处理的前提是确定消息的格式或结构。在 WCF 中消息的格式与结构由消息的版本决定，在 `Message` 中定义了一个类型为 `MessageVersion` 的 `Version` 属性来表示消息的版本。

```

1  public abstract class Message : IDisposable
2  {
3      //其他成员
4      public abstract MessageVersion Version { get; }
5  }

```

`MessageVersion` 类型定义在 `System.ServiceModel.Channels` 命名空间下。由于 SOAP 规范的版本和 WS-Addressing 规范的版本是决定消息格式与结构的两个主要因素，所以，

MessageVersion 由 SOAP 规范和 WS-Addressing 规范共同决定。WCF 通过 System.ServiceModel.EnvelopeVersion 和 System.ServiceModel.AddressingVersion 两个类分别定义 SOAP 规范的版本和 WS-Addressing 的版本。

MessageVersion 中定义了两个静态的方法 CreateVersion，用以创建相应的 MessageVersion 对象，两个属性 Envelope 和 Addressing 分别表示通过 EnvelopeVersion 和 AddressingVersion 体现的 SOAP 规范版本和 WS-Addressing 规范版本。

```
6  public sealed class MessageVersion
7  {
8      //其他成员
9      public static MessageVersion CreateVersion
10     (EnvelopeVersion envelopeVersion);
11     public static MessageVersion CreateVersion(EnvelopeVersion
12         envelopeVersion, AddressingVersion addressingVersion);
13
14     public AddressingVersion Addressing { get; }
15     public EnvelopeVersion Envelope { get; }
16 }
```

到目前为止 SOAP 和 WS-Addressing 各有两个版本:SOAP 1.1 和 SOAP 1.2, WS-Addressing 2004 和 WS-Addressing 1.0。它们分别通过定义在 EnvelopeVersion 和 AddressingVersion 中相应的静态只读属性表示。Soap11 和 Soap12 代表 SOAP 1.1 和 SOAP 1.2, 而 WSAddressingAugust2004 和 WSAddressing10 则表示 WS-Addressing 2004 和 WS-Addressing 1.0。EnvelopeVersion.None 表示消息并非一个 SOAP 消息, 比如非 XML 结构的消息(比如基于 JSON 格式)及 POX (Plain Old XML) 消息。AddressingVersion.None 则表示消息不遵循 WS-Addressing 规范, 比如通过手工方式解决寻址问题。

```
16 public sealed class EnvelopeVersion
17 {
18     //其他成员
19     public static EnvelopeVersion None { get; }
20     public static EnvelopeVersion Soap11 { get; }
21     public static EnvelopeVersion Soap12 { get; }
22 }
23 public sealed class AddressingVersion
24 {
25     //其他成员
26
27     public static AddressingVersion None { get; }
28     public static AddressingVersion WSAddressing10 { get; }
29     public static AddressingVersion WSAddressingAugust2004 { get; }
30 }
```

注：MessageVersion 的静态方法 CreateVersion(EnvelopeVersion envelopeVersion) 默认采用的 AddressingVersion 为 WSAddressing10。

由于 EnvelopeVersion 和 AddressingVersion 共同决定了 MessageVersion。所以 EnvelopeVersion 和 AddressingVersion 的两两组合就得到相应的 MessageVersion。这些两者组合得到的 MessageVersion 通过静态只读属性定义在 MessageVersion 类中。Soap11WSAddressing10、Soap11WSAddressingAugust2004、Soap12WSAddressing10 和 Soap12WSAddressingAugust2004 的含义都是一目了然的，而 None、Soap11 和 Soap12 表示的 EnvelopeVersion 和 Addressing 组合分别是：

```
31     None: EnvelopeVersion.None + AddressingVersion.None
32     Soap11: EnvelopeVersion.Soap11+ AddressingVersion.None
33     Soap12: EnvelopeVersion.Soap12 + AddressingVersion.None
34 public sealed class MessageVersion
35 {
36     //其他成员
37     public static MessageVersion Default { get; }
38
39     public static MessageVersion None { get; }
40     public static MessageVersion Soap11 { get; }
41     public static MessageVersion Soap11WSAddressing10 { get; }
42     public static MessageVersion Soap11WSAddressingAugust2004 { get; }
43     public static MessageVersion Soap12 { get; }
44     public static MessageVersion Soap12WSAddressing10 { get; }
45     public static MessageVersion Soap12WSAddressingAugust2004 { get; }
46 }
```

WS-Addressing 是建立在 SOAP 之上的，所以 EnvelopeVersion.None 和 AddressingVersion.WSAddressingAugust2004 及 AddressingVersion.WSAddressing10 的组合是无效的。此外在 MessageVersion 中还定义了一个静态只读属性 Default，它表示默认的 MessageVersion，目前该值为 MessageVersion.Soap12WSAddressing10。

### 6.2.2 如何创建消息（1）

由于 Message 是一个抽象类型，不能直接实例化。Message 类中定义了一系列静态 CreateMessage 方法，使我们能够方便快捷地以不同的方式进行消息的创建。对于如此众多的 CreateMessage 方法，按照具体的消息创建方式的不同，大体上可以分为5类：

创建空消息；

将对象序列化成消息的主体（Body）；

通过 XMLWriter 将内容“写”到消息中；

通过 XMLReader 将内容“读”到消息中；

创建 Fault 消息。

创建空消息

下面是所有 CreateMessage 静态方法中最简单的一个，包含两个输入参数：消息的版本和 Action。通过该方法可以创建一个只包含 Action 报头的 SOAP 消息。

```
1 public abstract class Message : IDisposable
2 {
3     //其他成员
4     public static Message CreateMessage
5     (MessageVersion version, string
6         action);
7 }
```

为演示消息的创建及创建后的消息结构，我写了下面一个辅助方法 WriteMessage。该方法将一个 Message 对象写入一个文件中，并通过开启进程的方式将文件打开。

```
7 static void WriteMessage(Message message, string fileName)
8 {
9     using (XmlWriter writer = new XmlTextWriter
10     (fileName, Encoding.UTF8))
11     {
12         message.WriteMessage(writer);
13     }
14     Process.Start(fileName);
15 }
```

通过下面的代码，调用 Message 的 CreateMessage 方法，并设置消息版本为 MessageVersion.Soap12WSAddressing10，Action 设置为 <http://www.artech.com/myaction>。最终将会生成如后面 XML 片断所示的 SOAP 消息。

```
15 string fileName = @"E:\message.xml";
16 Message message = Message.CreateMessage
17 (MessageVersion.Soap12WSAddressing10,
18     "http://www.artech.com/myaction");
19 WriteMessage(message, fileName);
20 <s:Envelope xmlns:a="http://www.w3.org/2005/
21 08/addressing" xmlns:s=
22     "http://www.w3.org/2003/05/soap-envelope">
23     <s:Header>
24         <a:Action s:mustUnderstand="1">http://www.artech.com/myaction
25         </a:Action>
26     </s:Header>
```

```
25     <s:Body />
26 </s:Envelope>
```

由于消息报头（Header）仅仅限于 SOAP 消息，所以如果将消息的版本改成 `MessageVersion.None`，制定的 Action 不会被包含在消息中。实际上创建的 `Message` 对象不包含任何内容，最终生成的 XML 文件也不会包含任何文本信息。

```
27 string fileName = @"E:\message.xml";
28 Message message = Message.CreateMessage(MessageVersion.None,
29     "http://www.artech.com/myaction");
30 WriteMessage(message, fileName);
```

将对象序列化成消息的主体

现在来关注 `Message` 第2个重载的 `CreateMessage` 静态方法。如下面代码所示，该方法在上面的重载方法的基础上加了一个 `object` 类型的 `body` 参数，它表示消息的主体（Body）。在执行该方法的时候，相应的序列化器会被调用，将对象序列化成 XML 并将其置于消息的主体部分。默认的序列化器就是前面介绍的 `DataContractSerializer`。

```
31 public abstract class Message : IDisposable
32 {
33     //其他成员
34     public static Message CreateMessage
35     (MessageVersion version, string action,
36         object body);
37 }
```

为了演示对象的序列化，我定义了下面一个数据契约 `Order`，并定义了4个数据成员：`OrderNo`、`OrderDate`、`Customer` 和 `ShipAddress`。

```
37 [DataContract(Namespace = "http://www.artech.com")]
38 public class Order
39 {
40     [DataMember(Name = "OrderNo", Order = 1)]
41     public Guid ID
42     { get; set; }
43
44     [DataMember(Name = "OrderDate", Order = 2)]
45     public DateTime Date
46     { get; set; }
47
48     [DataMember(Order = 3)]
49     public string Customer
50     { get; set; }
51 }
```



```

52     [DataMember(Order = 4)]
53     public string ShipAddress
54     { get; set; }
55 }

```

通过下面的代码，创建 Order 对象，并将其传入 CreateMessage 方法，作为 body 参数。最终将会生成如后面所示的 SOAP 消息。

```

56 string fileName = @"E:\message.xml";
57 Order order = new Order
58 {
59     ID = Guid.NewGuid(),
60     Date = DateTime.Today,
61     Customer = "Foo",
62     ShipAddress = "#328, Airport Rd, Industrial Park, Suzhou Jiangsu
Province"
63 };
64 Message message =
Message.CreateMessage(MessageVersion.Soap12WSAddressing10,
65     "http://www.artech.com/myaction", order);
66 WriteMessage(message, fileName);
67 <s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
68     xmlns:s="http://www.w3.org/2003/05/soap-envelope">
69     <s:Header>
70         <a:Action s:mustUnderstand="1">http://www.artech.com/myaction
71     </a:Action>
72     </s:Header>
73     <s:Body>
74         <Order xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns=
75     "http://www.artech.com">
76         <OrderNo>104a0213-1a0b-4d0b-b084-e912a991f908</OrderNo>
77         <OrderDate>2008-12-17T00:00:00+08:00</OrderDate>
78         <Customer>Foo</Customer>
79         <ShipAddress>#328, Airport Rd, Industrial Park, Suzhou
Jiangsu
80     Province</ShipAddress>
81     </Order>
82     </s:Body>
83 </s:Envelope>

```

### 6.2.2 如何创建消息 (2)

从上面生成的 XML 可以看出，SOAP 的主体部分就是 Order 对象通过 DataContractSerializer 序列化生成的 XML。如果消息不是一个 SOAP 消息呢？为了演示非

SOAP 消息的创建，我们将消息的版本替换成 MessageVersion.None。从最终产生的 XML 结构来看，消息的整个部分就是 Order 对象序列化后的 XML。

```
1  //其他代码
2  Message message = Message.CreateMessage
    (MessageVersion.Soap12WSAddressing10,
3      "http://www.artech.com/myaction", order);
4  WriteMessage(message, fileName);
5  <Order xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns=
6      "http://www.artech.com">
7      <OrderNo>ae6047b2-6154-4b77-9153-6ffae03ac7c6</OrderNo>
8      <OrderDate>2008-12-17T00:00:00+08:00</OrderDate>
9      <Customer>Foo</Customer>
10     <ShipAddress>#328, Airport Rd, Industrial Park, Suzhou Jiangsu
11         Province</ShipAddress>
12 </Order>
```

通过 BodyWriter 将内容写入消息

接下来，介绍另一个包含 BodyWriter 参数的 CreateMessage 方法重载。

```
13 public abstract class Message : IDisposable
14 {
15     //其他成员
16     public static Message CreateMessage
    (MessageVersion version, string action, BodyWriter body);
17 }
```

BodyWriter，顾名思义，就是消息主体的写入器。BodyWriter 是一个抽象类，定义在 System.ServiceModel.Channels 命名空间下，下面的代码简单地描述了 BodyWriter 的定义。构造函数参数 (isBuffered) 和只读属性 IsBuffered 表示消息是否被缓存。消息主体内容的写入实现在 OnWriteBodyContents 方法中。

```
18 public abstract class BodyWriter
19 {
20     //其他成员
21     protected BodyWriter(bool isBuffered);
22     protected abstract void OnWriteBodyContents
    (XmlDictionaryWriter writer);
23
24     public bool IsBuffered { get; }
25 }
```

为了演示基于 BodyWriter 的 Message 的创建过程，我自定义了一个简单的 BodyWriter: XmlReaderBodyWriter。它实现的功能很简单，就是从一个 XML 文件中读取内容作为消息主体的内容。XmlReaderBodyWriter 的定义如下：

```

26 public class XmlReaderBodyWriter : BodyWriter
27 {
28     private String _fileName;
29     internal XmlReaderBodyWriter(String fileName)
30         : base(true)
31     {
32         this._fileName = fileName;
33     }
34     protected override void OnWriteBodyContents
35     (XmlDictionaryWriter writer)
36     {
37         using (XmlReader reader = new XmlTextReader(this._fileName))
38         {
39             while (!reader.EOF)
40             {
41                 writer.WriteNode(reader, false);
42             }
43         }
44     }

```

假设现在有一个 XML 文件，具有下面列出的内容（即上面演示过程中 Order 对象序列化的结果），文件名为 E:\order.xml。

```

45 <Order xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
46     xmlns="http://www.artech.com">
47     <OrderNo>ae6047b2-6154-4b77-9153-6ffae03ac7c6</OrderNo>
48     <OrderDate>2008-12-17T00:00:00+08:00</OrderDate>
49     <Customer>FOO</Customer>
50     <ShipAddress>#328, Airport Rd, Industrial Park, Suzhou Jiangsu
51     Province</ShipAddress>
52 </Order>

```

那么我们就可以通过定义的 XmlReaderBodyWriter 进行消息的创建，具体代码实现如下所示。最终生成后面所示的 SOAP 消息。

```

53 string fileName1 = @"E:\order.xml";
54 string fileName2 = @"E:\message.xml";
55 XmlReaderBodyWriter writer = new XmlReader
56 BodyWriter(fileName1);
57 Message message = Message.CreateMessage
58 (MessageVersion.Soap12WSAddressing10,
59     "http://www.artech.com/myaction", writer);
60 WriteMessage(message, fileName2);
61 <s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"

```

```

60  xmlns:s="http://www.w3.org/2003/05/soap-envelope">
61      <s:Header>
62          <a:Action s:mustUnderstand="1">http://
www.artech.com/myaction
63      </a:Action>
64  </s:Header>
65  <s:Body>
66      <Order xmlns:i="http://www.w3.org/2001/
XMLSchema-instance" xmlns=
67          "http://www.artech.com">
68          <OrderNo>104a0213-1a0b-4d0b-b084-
e912a991f908</OrderNo>
69          <OrderDate>2008-12-17T00:00:00+08:00</OrderDate>
70          <Customer>FOO</Customer>
71          <ShipAddress>#328, Airport Rd,
Industrial Park, Suzhou Jiangsu
72              Province</ShipAddress>
73      </Order>
74  </s:Body>
75 </s:Envelope>

```

### 6.2.2 如何创建消息 (3)

#### 通过 XMLReader 将内容读到消息中

如果说基于 `BodyWriter` 进行消息的创建是采用一种“推”的模式将内容写入消息，那么基于 `XMLReader` 的方式就是采用一种“拉”的模式。`Message` 中定义了4个基于 `XmlReader` 的 `CreateMessage` 重载，其中两个是直接利用 `XmlReader` 的，其余两个则是通过 `XmlReader` 的子类 `XmlDictionaryReader` 进行消息内容的写入。关于 `XmlDictionaryReader`，将在本章的6.4节进行详细的介绍，在这里读者只须将其理解为一个特殊的 `XmlReader` 就可以了。

```

1  public abstract class Message : IDisposable
2  {
3      //其他成员
4      public static Message CreateMessage
(MessageVersion version, string action,
5          XmlDictionaryReader body);
6      public static Message CreateMessage
(MessageVersion version, string action,
7          XmlReader body);
8      public static Message CreateMessage
(XmlDictionaryReader envelopeReader,
9          int maxSizeOfHeaders, MessageVersion version);

```

```

10     public static Message CreateMessage
(XmlReader envelopeReader, int
11         maxSizeOfHeaders, MessageVersion version);
12 }

```

在下面的程序演示中，创建一个 XmlReader 对象，用于读取一个 XML 文件。将该 XmlReader 对象传入 CreateMessage 方法中，该方法将会利用该 XmlReader 读取相应的 XML，并将其作为消息的主体部分。

```

13 string fileName1 = @"E:\order.xml";
14 string fileName2 = @"E:\message.xml";
15
16 using (XmlReader reader = new XmlTextReader(fileName1))
17 {
18     Message message = Message.CreateMessage(MessageVersion.
19         Soap12WSAddressing10, "http://
www.artech.com/myaction", reader);
20     WriteMessage(message, fileName2);
21 }

```

### 创建 Fault 消息

在本章的6.1节中，我们讨论了 SOAP 1.2有关 SOAP Fault 的基本规范，相信读者对 SOAP Fault 的基本结构已有了一个基本的了解。接下来着重介绍如何创建一个 Fault 消息。在 Message 类中，定义了以下3个 CreateMessage 方法重载用以创建 Fault 消息。

```

22 public abstract class Message : IDisposable
23 {
24     //其他成员
25     public static Message CreateMessage
(MessageVersion version, MessageFault
26         fault, string action);
27     public static Message CreateMessage
(MessageVersion version, FaultCode
28         faultCode, string reason, string action);
29     public static Message CreateMessage
(MessageVersion version, FaultCode
30         faultCode, string reason, object detail, string action);
31 }

```

对于一个 Fault 消息来说，SOAP Code 和 SOAP Reason 是必须的元素。SOAP Reason 描述出错的基本原因，通过字符串的形式表示。SOAP Code 则通过一个特殊的类 System.ServiceModel.FaultCode 表示，定义如下。

```

32 public class FaultCode

```

```

33 {
34     public FaultCode(string name);
35     public FaultCode(string name, FaultCode subCode);
36     public FaultCode(string name, string ns);
37     public FaultCode(string name, string ns, FaultCode subCode);
38
39     public static FaultCode CreateReceiver
FaultCode(FaultCode subCode);
40     public static FaultCode CreateReceiver
FaultCode(string name, string ns);
41     public static FaultCode CreateSender
FaultCode(FaultCode subCode);
42     public static FaultCode CreateSender
FaultCode(string name, string ns);
43
44     public bool IsPredefinedFault { get; }
45     public bool IsReceiverFault { get; }
46     public bool IsSenderFault { get; }
47     public string Name { get; }
48     public string Namespace { get; }
49     public FaultCode SubCode { get; }
50 }

```

一个完整的 Fault Code 由一个必需的 Value 元素和一个可选的 SubCode 元素构成（如下面的 XML 片段所示）。而 Subcode 的规范和 Fault Code 一样，也就是说 Subcode 是一个 FaultCode，这实际上这是一个嵌套的结构。对应到 FaultCode 类中，属性 Name 和 Namespace 对应 Value 结点的内容，而 SubCode 则自然对应着 Fault Code 的 Subcode 结点。如果使用基于 SOAP 1.1 和 SOAP 1.2 的命名空间（SOAP 1.1 为 <http://schemas.xmlsoap.org/soap/envelope/>；SOAP 1.2 为 <http://www.w3.org/2003/05/soap-envelope>）或者是 <http://schemas.microsoft.com/ws/2005/05/envelope/none>（相当于 EnvelopeVersion.None），那么将被视为预定义错误（Fault）。对于一个 FaultCode，可以通过 IsPredefinedFault 属性判断是否为预定义错误。SOAP 1.1 和 SOAP 1.2 定义了一些预定义的 Fault Code，比如 VersionMismatch、MustUnderstand、DataEncodingUnknown、Sender、Receiver 等，其中 Sender 和 Receiver 表示发送端和接收端导致的错误。FaultCode 甚至定义了4个静态的方法（CreateSenderFaultCode 和 CreateReceiverFaultCode），方便开发者创建这两种特殊的 FaultCode。

```

51 <env:Code>
52     <env:Value>env:Sender</env:Value>
53     <env:Subcode>
54         <env:Value>m:MessageTimeout</env:Value>
55     </env:Subcode>

```

56 **</env:Code>**

下面的代码是一个典型的创建 Fault 消息例子，后面给出的 XML 是最终生成的 SOAP 消息。

```
57 string fileName
58 = @"E:\message.xml";
59 FaultCode subCode = new FaultCode("E0001", "http://www.artech.
60     com/faults/");
61 FaultCode faultCode = FaultCode.CreateSenderFaultCode(subCode);
62 Message message
63 = Message.CreateMessage(MessageVersion.Default,
64     faultCode, "Access is denied.", "http://
www.artech.com/myaction");
65 WriteMessage(message, fileName);
66 <s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
67     xmlns:s="http://www.w3.org/2003/05/soap-envelope">
68     <s:Header>
69         <a:Action s:mustUnderstand="1">http://
www.artech.com/myaction
70         </a:Action>
71     </s:Header>
72     <s:Body>
73         <s:Fault>
74             <s:Code>
75                 <s:Value>s:Sender</s:Value>
76             <s:Subcode>
77                 <s:Value xmlns:a="http://
www.artech.com/faults/">a:E0001
78                 </s:Value>
79             </s:Subcode>
80         </s:Code>
81         <s:Reason>
82             <s:Text xml:lang="en-US">Access is denied.</s:Text>
83         </s:Reason>
84     </s:Fault>
85 </s:Body>
86 </s:Envelope>
```

除了直接通过指定 Fault Code 和 Fault Reason 创建 Fault 消息之外，还可以利用 System.ServiceModel.Channels.MessageFault 对象的方式创建。实际上，MessageFault 就是 Fault 消息托管类型的表示。由于篇幅所限，在这里就不做详细介绍了，有兴趣的读者可以参阅 MSDN 在线文档。

### 6.2.3 消息的基本操作和消息状态（1）

知道了消息是如何创建的，接着讨论消息的一些基本操作。除了上面介绍的消息创建之外，一个消息涉及的操作大体分为以下4类。

读消息：读取整个消息的内容或有选择地读取报头或主体部分内容。

写消息：将整个消息的内容或主体部分内容写入文件或流。

拷贝消息：通过消息拷贝生成另一个具有相同内容的新消息。

关闭消息：关闭消息，回收一些非托管资源。

上述这些消息的基本操作都和消息的状态密切相关，消息操作和消息状态之间的关系体现在以下两个方面：

消息的状态决定了可以采取的操作；

消息操作伴随着消息状态的改变。

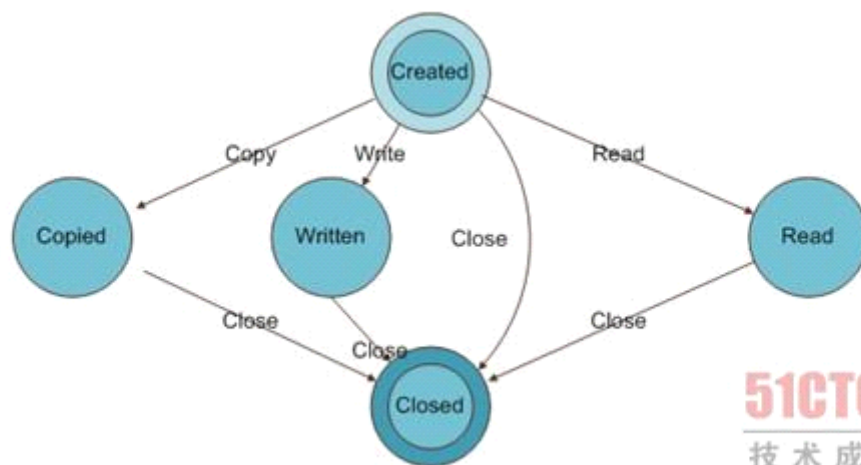
消息状态机（Message State Machine）

图6-1展示的是基于消息的状态机，从图中我们可以得出下面的一些关于 Message 对象状态转换的规则：

消息的读、写和拷贝操作只能作用在状态为 Created 的消息上；

消息的读、写和拷贝将消息状态从 Created 转换成 Read、Written 和 Copied；

所有状态的消息都可以直接关闭，关闭后消息的状态转换为 Closed。



（点击查看大图）图6-1 Message 对象状态机

在 WCF 中，消息的状态通过 `System.ServiceModel.Channels.MessageState` 枚举表示，



MessageState 定义了5种消息状态，与图6-1所示的5种状态一一对应。MessageState 的定义如下：

```
1  public enum MessageState
2  {
3      Created,
4      Read,
5      Written,
6      Copied,
7      Closed
8  }
```

## 消息的读取

读取消息主体部分的内容是最为常见的操作。如果主体部分的内容对应一个可以序列化的对象，可以通过 GetBody<T>方法读取消息主体并反序列化生成相应的对象。而通过 GetReaderAtBodyContents 得到一个 XmlDictionaryReader 对象，可以通过这个对象进一步提取消息主体部分的内容。

```
9  public abstract class Message : IDisposable
10 {
11     //其他成员
12     public T GetBody<T>();
13     public T GetBody<T>(XmlObjectSerializer serializer);
14     public XmlDictionaryReader GetReaderAtBodyContents();
15 }
```

我们演示一下 GetBody<T>方法的例子。假设消息主体部分对应的类型为下面所示的 Customer 类，这是一个数据契约。

```
16 [DataContract(Namespace = "http://www.artech.com")]
17 public class Customer
18 {
19     [DataMember]
20     public string Name
21     { get; set; }
22
23     [DataMember]
24     public string Compnay
25     { get; set; }
26
27     [DataMember]
28     public string Address
29     { get; set; }
30
31     public override bool Equals(object obj)
```

```

32     {
33         Customer customer = obj as Customer;
34         if (customer == null)
35         {
36             return false;
37         }
38         return this.Name == customer.Name && this.Compnay ==
customer.Compnay
39         && this.Address == customer.Address;
40     }
41 }

```

在下面的程序中，通过 Customer 对象创建 Message 对象，调用 GetBody<Customer>方法读取主体部分的内容并反序列化成 Customer 对象。可以想象开始创建的 Customer 对象和通过 GetBody<Customer>方法得到的 Customer 对象的值是相等的，输出的结果证明了这一点。

```

42 Customer customer = new Customer
43 {
44     Name      = "Foo",
45     Compnay    = "NCS",
46     Address    = "#328, Airport Rd,
47 Industrial Park, Suzhu Jiangsu Province"
48 };
49 Message message = Message.CreateMessage(MessageVersion.Default,
50     "http://www.artech.com/myaction", customer);
51 Customer cusomterToRead = message.GetBody<Customer>();
52 Console.WriteLine("customer.Equals(cusomterToRead) = {0}",
53     customer.Equals(cusomterToRead));

```

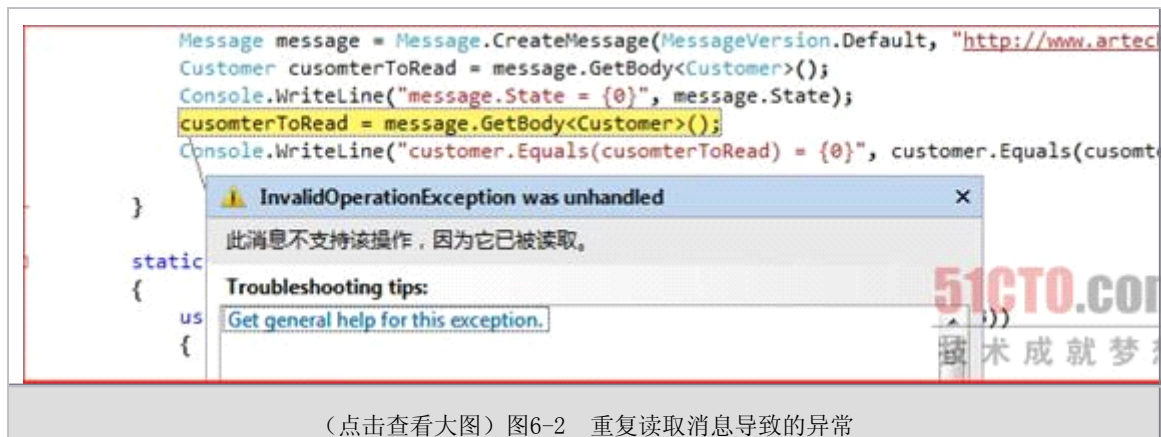
输出结果：

```

54 customer.Equals(cusomterToRead) = True

```

按照上面介绍的消息状态机的描述，只有状态为 Created 的消息才能执行读取操作，否则会抛出异常。无论是执行了 GetBody<T>方法还是 GetReaderAtBodyContents方法，Message 对象的状态都将转换为 Read。在上面代码的基础上，添加了两行额外的代码输出消息的状态，并再一次调用 Message 对象的 GetBody<T>方法。程序运行输出消息的状态（message.State = Read），在执行到第2个 GetBody<T>方法时，抛出如图6-2所示的 InvalidOperationException 异常。



```
55 //省略代码
56 Message message = Message.CreateMessage(MessageVersion.Default,
57     "http://www.artech.com/myaction", customer);
58 Customer cusomterToRead = message.GetBody<Customer>();
59 Console.WriteLine("message.State = {0}", message.State);
60 cusomterToRead = message.GetBody<Customer>();
61 Console.WriteLine("customer.Equals(cusomterToRead) = {0}",
62     customer.Equals(cusomterToRead));
```

输出结果:

```
63 message.State = Read
```

### 6.2.3 消息的基本操作和消息状态 (2)

#### 消息的写入

在 Message 类中, 定义了一系列 WriterXxx 方法用于消息的写操作。通过这些方法, 我们可以将整个消息或消息的主体部分内容写入 XmlWriter 或 XmlDictionaryWriter 中, 并最终写入文件或流。

```
1 public abstract class Message : IDisposable
2 {
3     //其他成员
4     public void WriteBody(XmlDictionaryWriter writer);
5     public void WriteBody(XmlWriter writer);
6     public void WriteBodyContents(XmlDictionaryWriter writer);
7     public void WriteMessage(XmlDictionaryWriter writer);
8     public void WriteMessage(XmlWriter writer);
9     public void WriteStartBody(XmlDictionaryWriter writer);
10    public void WriteStartBody(XmlWriter writer);
11    public void WriteStartEnvelope(XmlDictionaryWriter writer);
12 }
```

我们在前面演示时创建的辅助方法 WriteMessage（如下面的代码所示），就是通过调用 WriteMessage 方法将消息的内容写入一个指定的 XML 文件中的。同消息的读取一样，写操作只能作用于状态为 Created 的消息。成功执行了消息写入操作后，状态转换为 Written。

```
13 static void WriteMessage(Message message, string fileName)
14 {
15     using (XmlWriter writer = new XmlTextWriter
16         (fileName, Encoding.UTF8))
17     {
18         message.WriteMessage(writer);
19     }
20     Process.Start(fileName);
21 }
```

### 消息的拷贝

通过前面的介绍和演示，相信读者对消息的状态转换已有一个清晰的认识：消息的读写都会改变消息的状态，而读写操作只能作用于状态为 Created 的消息。由此就出现了这样一个问题：在真正的 WCF 应用中，我们往往需要将消息进行日志记录。如果按照正常的方式进行消息的读取和写入，会导致状态的改变，如果消息传递到 WCF 的处理管道，作用于该消息对象的读、写操作都将失败。在这种情况下，需要使用到消息的拷贝功能。Message 类中定义了一个 CreateBufferedCopy 方法，专门用于消息的拷贝。

```
21 public abstract class Message : IDisposable
22 {
23     //其他成员
24     public MessageBuffer CreateBufferedCopy(int maxBufferSize);
25 }
```

CreateBufferedCopy 方法的返回结果并不是我们想象的 Message 对象，而是一个 System.ServiceModel.Channels.MessageBuffer 对象，MessageBuffer 表示消息在内存中缓存。当 CreateBufferedCopy 成功执行时，消息的状态转换成 Copied，很显然后续的操作不能再使用该消息。但是却可以通过 MessageBuffer 对象创建一个新的 Message 对象，该对象具有与原来一样的内容，但是状态却是 Created。在 MessageBuffer 中，定义了如下一个 CreateMessage 方法，用于新消息的创建。

```
26 public abstract class MessageBuffer :
27     IXPathNavigable, IDisposable
28 {
29     //其他成员
30     public abstract Message CreateMessage();
31 }
```

比如，我们通过下面的方式解决前面所演示的重复读取的问题，将不会再有

InvalidOperationException 异常抛出。

```
31 Message message = Message.CreateMessage(MessageVersion.Default,
32     "http://www.artech.com/myaction", customer);
33 MessageBuffer messagemessageBuffer = message.
    CreateBufferedCopy(int.MaxValue);
34 message = messageBuffer.CreateMessage();
35 Customer cusomterToRead = message.GetBody<Customer>();
36 message = messageBuffer.CreateMessage();
37 cusomterToRead = message.GetBody<Customer>();
38 Console.WriteLine("customer.Equals(cusomterToRead) = {0}",
39     customer.Equals(cusomterToRead));
```

#### 6.2.4 消息报头集合 (1)

按照 SOAP 1.1或 SOAP 1.2规范,一个 SOAP 消息由若干 SOAP 报头和一个 SOAP 主体构成, SOAP 主体是 SOAP 消息的有效负载,一个 SOAP 消息必须包含一个唯一的消息主体。SOAP 报头是可选的,一个 SOAP 消息可以包含一个或多个 SOAP 报头, SOAP 报头一般用于负载一些控制信息。消息一经创建,其主体内容不能改变,而 SOAP 报头则可以自由地添加、修改和删除。正是因为 SOAP 这种高度可扩展的设计,使得 SOAP 成为实现 SOA 的首选(有这么一种说法: SOAP= SOA Protocol)。

按照 SOAP 1.2规范,一个 SOAP 报头集合由一系列 XML 元素组成,每一个报头元素的名称为 Header,命名空间为 <http://www.w3.org/2003/05/soap-envelope>。每一个报头元素可以包含任意的属性(Attribute)和子元素。在 WCF 中,定义了一系列类型用于表示 SOAP 报头。

```
1 MessageHeaders、MessageHeaderInfo、
    MessageHeader 和 MessageHeader<T>
```

在 Message 类中,消息报头集合通过只读属性 Headers 表示,类型为 System.ServiceModel.Channels.MessageHeaders。MessageHeaders 本质上就是一个 System.ServiceModel.Channels.MessageHeaderInfo 集合。

```
2 public abstract class Message : IDisposable
3 {
4     //其他成员
5     public abstract MessageHeaders Headers { get; }
6 }
7 public sealed class MessageHeaders :
    IEnumerable<MessageHeaderInfo>, IEnumerable
8 {
9     //省略成员
10 }
```

MessageHeaderInfo 是一个抽象类型，是所有消息报头的基类，它定义了一系列消息 SOAP 报头的基本属性。其中 Name 和 Namespace 分别表示报头的名称和命名空间，Actor、MustUnderstand、Reply 与 SOAP 1.1 或 SOAP 1.2 规定的 SOAP 报头的同名属性对应。需要对 SOAP 规范进行深入了解的读者可以从 W3C 官方网站下载相关文档。

```
11 public abstract class MessageHeaderInfo
12 {
13     protected MessageHeaderInfo();
14
15     public abstract string Actor { get; }
16     public abstract bool    IsReferenceParameter { get; }
17     public abstract bool    MustUnderstand { get; }
18     public abstract string Name { get; }
19     public abstract string Namespace { get; }
20     public abstract bool    Relay { get; }
21 }
```

当我们针对消息报头编程的时候，使用到的是另一个继承自 MessageHeaderInfo 的抽象类：System.ServiceModel.Channels.MessageHeader。除了实现 MessageHeaderInfo 定义的抽象只读属性外，MessageHeader 定义了一系列工厂方法（CreateHeader），方便开发人员创建 MessageHeader 对象。这些 CreateHeader 方法接受一个可序列化的对象，并以此作为消息报头的内容，WCF 内部会负责从对象到 XML InfoSet 的序列化工作。此外，可以通过相应的 WriteHeader 方法对 MessageHeader 对象执行写操作。MessageHeader 定义如下：

```
22 public abstract class MessageHeader : MessageHeaderInfo
23 {
24     public static MessageHeader CreateHeader
25     (string name, string ns, object
26         value);
27     public static MessageHeader CreateHeader
28     (string name, string ns, object
29         value, bool mustUnderstand);
30     //其他 CreateHeader 方法
31
32     public void WriteHeader(XmlDictionaryWriter
33         writer, MessageVersion
34         messageVersion);
35     public void WriteHeader(XmlWriter writer,
36         MessageVersion messageVersion);
37     //其他 WriteHeader 方法
38
39     public override string Actor { get; }
40     public override bool IsReferenceParameter { get; }
```

```

37     public override bool MustUnderstand { get; }
38     public override bool Relay { get; }
39 }

```

除了 MessageHeader，WCF 还提供一个非常有价值的泛型类：System.ServiceModel.MessageHeader<T>，泛型参数 T 表示报头内容对应的类型，MessageHeader<T>为我们提供了强类型的报头创建方式。由于 Message 的 Headers 属性是一个 MessageHeaderInfo 的集合，MessageHeader<T>并不能直接作为 Message 对象的消息报头。GetUntypedHeader 方法提供了从 MessageHeader<T>对象到 MessageHeader 对象的转换。MessageHeader<T>定义如下：

```

40 public class MessageHeader<T>
41 {
42     public MessageHeader();
43     public MessageHeader(T content);
44     public MessageHeader(T content, bool
mustUnderstand, string actor, bool
45         relay);
46     public MessageHeader GetUntypedHeader
(string name, string ns);
47
48     public string Actor { get; set; }
49     public T Content { get; set; }
50     public bool MustUnderstand { get; set; }
51     public bool Relay { get; set; }
52 }

```

接下来，通过一个简单的例子演示如何为一个 Message 对象添加报头。假设在一个 WCF 应用中，我们需要在客户端和服务端之间传递一些上下文的信息（Contextual Information），比如当前用户的相关信息。为此定义一个 ApplicationContext 类，这是一个集合数据契约。ApplicationContext 是一个字典，为了简单起见，key 和 value 均使用字符串。ApplicationContext 不能被创建（构造函数被私有化），只能通过静态只读属性 Current 得到。当前 ApplicationContext 存入 CallContext 从而实现了在线程范围内共享的目的。在 ApplicationContext 中定义了两个属性 UserName 和 Department，表示用户名称和所在部门。3个常量分别表示 ApplicationContext 存储于 CallContext 的 Key，以及置于 MessageHeader 后对应的名称和命名空间。

```

53 [CollectionDataContract(Namespace = "
http://www.artech.com/", ItemName =
54     "Context", KeyName = "Key", ValueName
= "Value")]
55 public class ApplicationContext :
Dictionary<string, string>
56 {

```

```

57     private const string callContextKey = "__applicationContext";
58     public const string HeaderLocalName = "ApplicationContext";
59     public const string HeaderNamespace = "http://www.artech.com/";
60
61     private ApplicationContext()
62     { }
63
64     public static ApplicationContext Current
65     {
66         get
67         {
68             if (CallContext.GetData(callContextKey)
69 == null)
70             {
71                 CallContext.SetData(callContextKey,
72 new ApplicationContext());
73             }
74         }
75
76         public string UserName
77         {
78             get
79             {
80                 if (!this.ContainsKey("__username"))
81                 {
82                     return string.Empty;
83                 }
84                 return this["__username"];
85             }
86             set
87             {
88                 this["__username"] = value;
89             }
90         }
91
92         public string Department
93         {
94             get
95             {
96                 if (!this.ContainsKey("__department"))
97                 {

```



```

98         return string.Empty;
99     }
100     return this["__department"];
101 }
102 set
103 {
104     this["__department"] = value;
105 }
106 }
107 }

```

#### 6.2.4 消息报头集合 (2)

在下面代码中，首先对当前 `ApplicationContext` 进行相应的设置，然后创建 `MessageHeader<ApplicationContext>` 对象。通过调用 `GetUntypedHeader` 转换成 `MessageHeader` 对象之后，将其添加到 `Message` 的 `Headers` 属性集合中。后面是生成的 SOAP 消息。

```

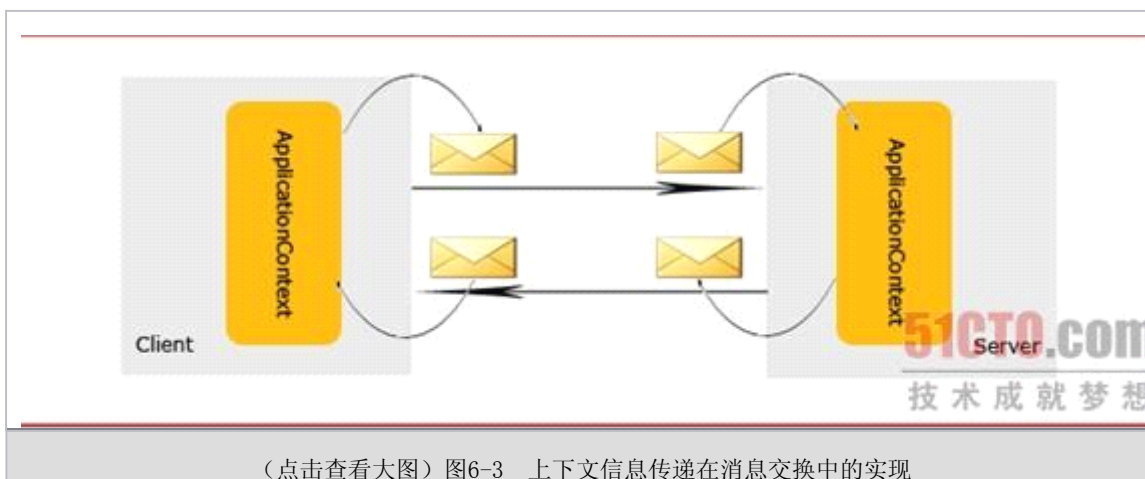
1  Message message = Message.CreateMessage(MessageVersion.Default,
2     "http://www.artech.com/myaction");
3  ApplicationContext.Current.UserName = "Foo";
4  ApplicationContext.Current.Department = "IT";
5  MessageHeader<ApplicationContext> header = new
6     MessageHeader<ApplicationContext>(ApplicationContext.Current);
7  message.Headers.Add(header.GetUntypedHeader(ApplicationContext.
8     HeaderLocalName, ApplicationContext.HeaderNamespace));
9  WriteMessage(message, @"e:\message.xml");
10 <s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
11     xmlns:s="http://www.w3.org/2003/05/soap-envelope">
12     <s:Header>
13         <a:Action s:mustUnderstand="1">http://www.artech.com/myaction
14         </a:Action>
15         <ApplicationContext xmlns:i="http://www.w3.org/2001/
16             XMLSchema-instance" xmlns="http://www.artech.com/">
17             <Context>
18                 <Key>__username</Key>
19                 <Value>Foo</Value>
20             </Context>
21             <Context>
22                 <Key>__department</Key>
23                 <Value>IT</Value>
24             </Context>
25         </ApplicationContext>
26     </s:Header>
27     <s:Body />

```

案例演示：通过消息报头传递上下文信息

在演示添加消息报头的例子中，创建了一个 `ApplicationContext`，这个类型将继续为本案服务。上面仅仅演示了如何为一个现成的 `Message` 对象添加相应的报头，在本案例中，我们将演示在一个具体的 WCF 应用中如何通过添加消息报头的方式从客户端向服务端传递一些上下文信息。

上面定义的 `ApplicationContext` 借助于 `CallContext` 实现了同一线程内数据的上下文消息的共享。由于 `CallContext` 的实现方式是将数据存储于当前线程的 TLS（Thread Local Storage）中，所以它仅仅在客户端或服务端执行的线程中有效。现在我们希望相同的上下文信息能够在客户端和服务端之间传递，毫无疑问，唯一的办法就是将信息存放在请求消息和回复消息中。图6-3大体上演示了具体的实现机制。



客户端的每次服务调用，会将当前 `ApplicationContext` 封装成 `MessageHeader`，存放到出栈消息（Outbound Message）的 SOAP 报头中；服务端在接收到入栈消息（InBound message）后，将其取出，作为服务端的当前 `ApplicationContext`。由此实现了客户端向服务端的上下文传递。从服务端向客户端上下文传递的实现与此类似：服务端将当前 `ApplicationContext` 植入出栈消息（Outbound Message）的 SOAP 报头中，接收到该消息的客户端将其取出，覆盖掉现有上下文的值。

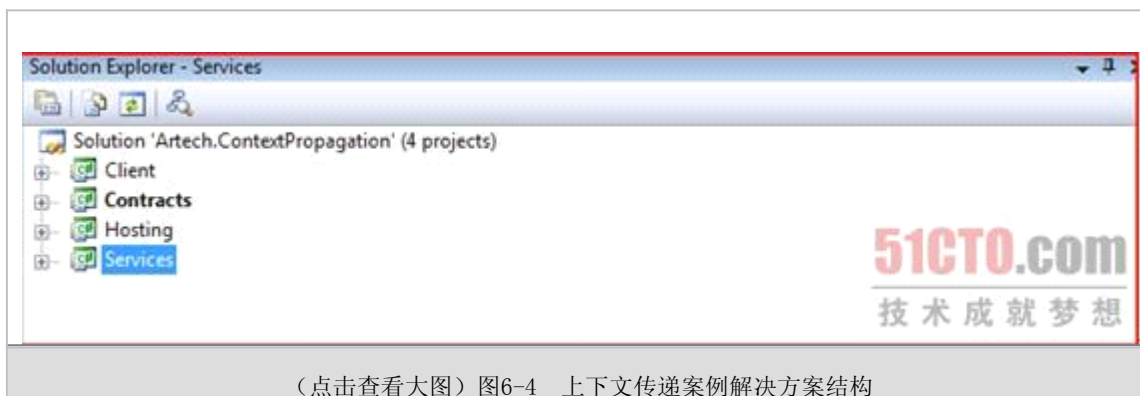
我们知道了如何实现消息报头的创建，现在需要解决的是如何将创建的消息报头植入到出栈和入栈消息报头集合中。可以借助 `System.ServiceModel.OperationContext` 实现这样的功能。`OperationContext` 代表当前操作执行的上下文，定义了一系列与当前操作执行有关的上下文属性，其中就包含出栈和入栈消息报头集合。对于一个请求-回复模式服务调用来讲，`IncomingMessageHeaders` 和 `OutgoingMessageHeaders` 对于客户端分别代表回复和请求消息的 SOAP 报头，而对于服务端则与此相反。

注： `OperationContext` 代表服务操作执行的上下文。通过 `OperationContext` 可以得

到出栈和入栈消息的 SOAP 报头列表、消息属性或 HTTP 报头。对于 Duplex 服务，在服务端可以通过 `OperationContext` 得到回调对象。此外通过 `OperationContext` 还可以得到基于当前执行的安全属性的其他相关信息。

```
29 public sealed class OperationContext :
    IExtensibleObject<OperationContext>
30 {
31     //其他成员
32     public MessageHeaders IncomingMessageHeaders { get; }
33     public MessageHeaders OutgoingMessageHeaders { get; }
34 }
```

有了上面这些铺垫，对于我们即将演示的案例就很好理解了。照例创建一个简单的计算器的例子，同样按照经典的4层结构处理，如图6-4所示。



先看看服务契约 (`ICalculator`) 和服务实现 (`CalculatorService`)。在 `Add` 操作的具体实现中，先通过 `OperationContext.Current.IncomingMessageHeaders` 根据预先定义在 `ApplicationContext` 中的报头名称和命名空间得到从客户端传入的 `ApplicationContext`，并将其输出。待运算结束后，修改服务端当前 `ApplicationContext` 的值，并将其封装成 `MessageHeader`，通过 `OperationContext.Current.OutgoingMessageHeaders` 植入到回复消息的 SOAP 报头中。

```
35 using System.ServiceModel;
36 namespace Artech.ContextPropagation.Contracts
37 {
38     [ServiceContract]
39     public interface ICalculator
40     {
41         [OperationContract]
42         double Add(double x, double y);
43     }
```

```

44 }
45 using System;
46 using Artech.ContextPropagation.Contracts;
47 using System.ServiceModel;
48 namespace Artech.ContextPropagation.Services
49 {
50     public class CalculatorService : ICalculator
51     {
52         public double Add(double x, double y)
53         {
54             //从请求消息报头中获取 ApplicationContext
55             ApplicationContext context = OperationContext.Current.
56                 IncomingMessageHeaders.GetHeader<ApplicationContext>
57                 (ApplicationContext.HeaderLocalName, ApplicationContext.
58                 HeaderNamespace);
59             ApplicationContext.Current.UserName = context.UserName;
60             ApplicationContext.Current.Department = context.Department;
61             Console.WriteLine("ApplicationContext.Current.UserName =
62                 \"{0}\"", ApplicationContext.Current.UserName);
63             Console.WriteLine("ApplicationContext.Current.Department =
64                 \"{0}\"", ApplicationContext.Current.Department);
65
66             double result = x + y;
67
68             // 将服务端当前 ApplicationContext 添加到回复消息报头集合
69             ApplicationContext.Current.UserName = "Bar";
70             ApplicationContext.Current.Department = "HR/Admin";
71             MessageHeader<ApplicationContext> header = new MessageHeader
72                 <ApplicationContext>(ApplicationContext.Current);
73
74             OperationContext.Current.OutgoingMessageHeaders.Add(header.
75                 GetUntypedHeader(ApplicationContext.HeaderLocalName,
76                 ApplicationContext.HeaderNamespace));
77
78             return result;
79         }
80     }
81 }

```

#### 6.2.4 消息报头集合 (3)

客户端的代码与服务端在消息报头的设置和获取正好相反。在服务调用代码中，先初始化当前 ApplicationContext，通过 ChannelFactory<ICalculator>创建服务代理对象。根据创建的服务代理对象创建 OperationContextScope 对象。在该 OperationContextScope 对象

的作用范围内（using 块中），将当前的 ApplicationContext 封装成 MessageHeader 并植入出栈消息的报头列表中，待正确返回执行结果后，获取服务端植入回复消息中返回的 ApplicationContext，并覆盖掉现有的 Context 相应的值。

注：同 Transaction 和 TransactionScope 一样，OperationContextScope 定义了当前 OperationContext 存活的范围。对于客户端来说，当前 OperationContext 的生命周期和 OperationContextScope 一样，一旦成功创建 OperationContextScope，就会创建当前的 OperationContext，当 OperationContextScope 的 Dispose 方法被执行，当前的 OperationContext 对象也相应被回收。

```
1  using System;
2  using Artech.ContextPropagation.Contracts;
3  using System.ServiceModel;
4  using System.ServiceModel.Channels;
5  namespace Artech.ContextPropagation
6  {
7      class Program
8      {
9          static void Main(string[] args)
10         {
11             ApplicationContext.Current.UserName = "Foo";
12             ApplicationContext.Current.Department = "IT";
13             using (ChannelFactory<ICalculator> channelFactory = new
14                 ChannelFactory<ICalculator>("CalculatorService"))
15             {
16                 ICalculator calculator = channelFactory.CreateChannel();
17                 using (calculator as IDisposable)
18                 {
19                     using (OperationContextScope contextScope = new
20                         OperationContextScope
21                         (calculator as IContextChannel))
22                     {
23                         //将客户端当前 ApplicationContext 添加到请求消息报头集合
24                         MessageHeader<ApplicationContext> header = new
25                             MessageHeader<ApplicationContext>
26                             (ApplicationContext.Current);
27                         OutgoingMessageHeaders.Add
28                             (header.GetUntypedHeader(ApplicationContext.
29                                 HeaderLocalName, ApplicationContext.
30                                 HeaderNamespace));
31                         Console.WriteLine("x + y = {2}
32                         when x = {0} and y =
```

```

31             {1}", 1, 2, calculator.Add(1, 2));
32             //从回复消息报头中获取 ApplicationContext
33             ApplicationContext context =
OperationContext.Current.
34
IncomingMessageHeaders.GetHeader<ApplicationContext>
35             (ApplicationContext.HeaderLocalName,
36             ApplicationContext.HeaderNamespace);
37             ApplicationContext.Current.UserName
= context.UserName;
38             ApplicationContext.Current.Department =
39             context.Department;
40         }
41     }
42 }
43 Console.WriteLine("ApplicationContext.Current.UserName =
44     \"{0}\"", ApplicationContext.Current.UserName);
45 Console.WriteLine("ApplicationContext.Current.Department =
46     \"{0}\"", ApplicationContext.Current.Department);
47
48 Console.Read();
49     }
50 }
51 }

```

下面的两段代码分别代表服务端（Hosting）和客户端的输出结果，从中可以很清晰地看出，AppContext 实现了在客户端和服务端之间的双向传递。

```

52 ApplicationContext.Current.UserName = "Foo"
53 ApplicationContext.Current.Department = "IT"
54
55 x + y = 3 when x = 1 and y = 2
56 ApplicationContext.Current.UserName = "Bar"
57 ApplicationContext.Current.Department = "HR/Admiin"

```

### 6.3 消息契约（Message Contract）

在本节中，我们将讨论 WCF 4大契约之一的消息契约（Message Contract），消息契约是本书继服务契约和数据契约后的第3个契约。服务契约关注于对服务操作的描述，数据契约关注于对数据结构和格式的描述，而消息契约关注的是类型成员与消息的匹配关系。

我们知道只有可序列化的对象才能通过服务调用在客户端和服务端之间进行传递。到目前为止，我们知道的可序列化类型有两种：一种是应用了 SerializableAttribute 特性或实现了 ISerializable 接口的类型；另一种是数据契约对象。对于基于这两种类型的服务操作，

客户端通过 ClientMessageFormatter 将输入参数格式化成为请求消息，输入参数的全部内容都会作为有效负载置于消息的主体中；同样地，服务操作的执行结果被 DispatchMessageFormatter 序列化后作为回复消息的主体。

在一些情况下，会有这样的要求：当序列化一个对象并生成消息的时候，希望将部分数据成员作为 SOAP 的报头，部分作为消息的主体。比如说，我们有一个服务操作采用流的方式进行文件的上载，除了以流的方式传输以二进制表示的文件内容外，还需要传输一个额外的基于文件属性的信息，比如文件格式、文件大小等。一般的做法是将传输文件的内容流作为 SOAP 的主体，将其属性内容作为 SOAP 的报头进行传递。这样的功能，可以通过定义消息契约来实现。

### 6.3.1 消息契约的定义

消息契约和数据契约一样，都是定义在数据（而不是功能）类型上。不过数据契约旨在定义数据的结构（将数据类型与 XSD 进行匹配），而消息契约则更多地关注于数据的成员具体在 SOAP 消息中的表示。消息契约通过以下3个特性进行定义：MessageContractAttribute、MessageHeaderAttribute、MessageBodyMemberAttribute。MessageContractAttribute 应用于类型上，MessageHeaderAttribute 和 MessageBodyMemberAttribute 则应用于属性或字段成员上，以表明相应的数据成员是一个基于 SOAP 报头的成员还是 SOAP 主体的成员。先来简单介绍一下这3个特性。

#### MessageContractAttribute

通过在一个类或结构（Struct）上应用 MessageContractAttribute 使之成为一个消息契约。从 MessageContractAttribute 的定义来看，MessageContractAttribute 大体上具有以下两种类型的属性成员：

ProtectionLevel 和 HasProtectionLevel：表示保护级别，在服务契约中已经对保护级别作了简单的介绍，WCF 中通过 System.Net.Security.ProtectionLevel 枚举定义消息的保护级别。一般有3种可选的保护级别：None、Sign 和 EncryptAndSign。在下一卷关于 WCF 安全的章节中还将对此进行详细介绍。

IsWrapped、WrapperName、WrapperNamespace：IsWrapped 表述的含义为是否为定义的主体成员（一个或者多个）添加一个额外的根节点。WrapperName 和 WrapperNamespace 则表述该根节点的名称和命名空间。IsWrapped、WrapperName、WrapperNamespace 默认分别为 true、类型名称和 <http://tempuri.org/>。

```
1  [AttributeUsage(AttributeTargets.Struct |
AttributeTargets.Class, AllowMultiple = false)]
2  public sealed class MessageContractAttribute : Attribute
```

```

3  {
4      //其他成员
5      public bool          HasProtectionLevel { get; }
6      public ProtectionLevel ProtectionLevel { get; set; }
7
8      public bool          IsWrapped { get; set; }
9      public string  WrapperName { get; set; }
10     public string    WrapperNamespace { get; set; }
11 }

```

在下面的代码中通过应用 MessageContractAttribute 使 Customer 类型成为一个消息契约。ID 和 Name 属性通过应用 MessageHeaderAttribute 定义成消息报头成员，而 Address 属性则通过 MessageBodyMemberAttribute 定义成消息主体成员。后面的 XML 体现的是 Customer 对象在 SOAP 消息中的表现形式。

```

12 [MessageContract]
13 public class Customer
14 {
15     [MessageHeader(Name = "CustomerNo", Namespace =
16         "http://www.artech.com/")]
17     public Guid ID
18     { get; set; }
19
20     [MessageHeader(Name = "CustomerName", Namespace =
21         "http://www.artech.com/")]
22     public string Name
23     { get; set; }
24
25     [MessageBodyMember(Namespace = "http://www.artech.com/")]
26     public string Address
27     { get; set; }
28 }
29 <s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
30     xmlns:s="http://www.w3.org/2003/05/soap-envelope">
31     <s:Header>
32                                     <a:Action
33 s:mustUnderstand="1">http://tempuri.org/IOrderManager/
34     ProcessOrder</a:Action>
35     <h:CustomerName xmlns:h="
36 http://www.artech.com/">Foo
37     </h:CustomerName>
38     <h:CustomerNo xmlns:h="http://
39 www.artech.com/">2f62405b-a472-4d1c-
40 8c03-b888f9bd0df9</h:CustomerNo>

```



```

38     </s:Header>
39     <s:Body>
40         <Customer xmlns="http://tempuri.org/">
41             <Address xmlns="http://
www.artech.com/">#328, Airport Rd,
42                 Industrial Park, Suzhou
Jiangsu Province</Address>
43         </Customer>
44     </s:Body>
45 </s:Envelope>

```

如果我们将 IsWrapped 的属性设为 false，那么套在 Address 节点外的 Customer 节点将会从 SOAP 消息中去除。

```

46 [MessageContract(IsWrapped = false)]
47 public class Customer
48 {
49     //省略成员
50 }
51 <s:Envelope xmlns:a="http://www.w3.org/2005/08/
addressing" xmlns:s="http://www.w3.org/2003/05/soap-envelope">
52     .....
53     <s:Body>
54         <Address xmlns="http://
www.artech.com/">#328, Airport Rd, Industrial
55             Park, Suzhou Jiangsu Province</Address>
56     </s:Body>
57 </s:Envelope>

```

同样可以自定义这个主体封套（Wrapper）的命名和命名空间。下面我们将 MessageContractAttribute 的 WrapperName 和 WrapperNamespace 属性设为 Cust 和 <http://www.artech.com/>。

```

58 [MessageContract(IsWrapped = true, WrapperName
= "Cust", WrapperNamespace =
59     "http://www.artech.com/")]
60 public class Customer
61 {
62     //省略成员
63 }
64 <s:Envelope xmlns:a="http://www.w3.org/
2005/08/addressing" xmlns:s=
65     "http://www.w3.org/2003/05/soap-envelope">
66     .....
67     <s:Body>

```

```

68         <Cust xmlns="http://www.artech.com/">
69             <Address>#328, Airport Rd,
Industrial Park, Suzhou Jiangsu
70             Province</Address>
71         </Cust>
72     </s:Body>
73 </s:Envelope>

```

### MessageHeaderAttribute

MessageHeaderAttribute 和 MessageBodyMemberAttribute 分别用于定义消息报头成员和消息主体成员，它们都有一个共同的基类：System.ServiceModel.MessageContractMemberAttribute。MessageContractMemberAttribute 定义了以下属性成员：HasProtectionLevel、ProtectionLevel、Name 和 Namespace。

```

74 public abstract class MessageContractMemberAttribute : Attribute
75 {
76     public bool HasProtectionLevel { get; }
77     public ProtectionLevel ProtectionLevel { get; set; }
78
79     public string Name { get; set; }
80     public string Namespace { get; set; }
81 }

```

通过在属性或字段成员上应用 MessageHeaderAttribute 使之成为一个消息报头成员。MessageHeaderAttribute 定义了以下3个属性，阅读了本章6.1节的内容或对于 SOAP 规范有一定了解的读者，相信对它们不会陌生。

**Actor:** 表示处理该报头的目标节点 (SOAP Node)，SOAP 1.1中对应的属性 (Attribute) 为 actor，SOAP 1.2中就是我们介绍的 role 属性。

**MustUnderstand:** 表述 Actor (SOAP 1.1) 或 Role (SOAP 1.2) 定义的 SOAP 节点是否必须理解并处理该节点。对应的 SOAP 报头属性为 mustUnderstand。

**Relay:** 对应的 SOAP 报头属性为 relay，表明该报头是否需要传递到下一个 SOAP 节点。

```

82 [AttributeUsage(AttributeTargets.Field | AttributeTargets.Property,
83     AllowMultiple = false, Inherited = false)]
84 public class MessageHeaderAttribute : MessageContractMemberAttribute
85 {
86     public string Actor { get; set; }
87     public bool MustUnderstand { get; set; }
88     public bool Relay { get; set; }
89 }

```

同样使用上面定义的 Customer 消息契约，现在我们相应地修改了 ID 属性上的 MessageHeaderAttribute 设置：MustUnderstand = true, Relay=true, Actor=<http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver>。实际上是将相应的 SOAP 报头的目标 SOAP 节点定义成最终的消息接收者。由于<http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver>是 SOAP 1.2的预定义属性，所以这个消息契约之后在基于 SOAP 1.2的消息版本中有效。后面给出的是对应的 SOAP 消息。

```
90 [MessageContract(IsWrapped =true, WrapperNamespace="http://
91   www.artech.com/")]public class Customer
92 {
93     //其他成员
94     [MessageHeader(Name="CustomerNo", Namespace = "http://
95       www.artech.com/" ,MustUnderstand =true, Relay=true, Actor=
96       "http://www.w3.org/2003/05/soap-
97       envelope/role/ultimateReceiver" )]
98     public Guid ID
99     { get; set; }
100 }
101 <s:Envelope xmlns:a="http://www.w3.org/2005/08/
102   addressing" xmlns:s=
103     "http://www.w3.org/2003/05/soap-envelope">
104     <s:Header>
105         .....
106         <h:CustomerNo s:role="http://www.w3.
107           org/2003/05/soap-envelope/role/
108             ultimateReceiver" s:mustUnderstand="1" s:relay="1" xmlns:h=
109               "http://www.artech.com/">5330c91a-
110               7fd7-4bf5-ae3e-4ba9bfef3d4d
111             </h:CustomerNo>
112           </s:Header>
113           .....
114     </s:Envelope>
```

<http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver>在 SOAP 1.1中对应的表示为：“<http://schemas.xmlsoap.org/soap/actor/ultimateReceiver>”。如果在 SOAP 1.1下，ID 成员对应的 MessageHeaderAttribute 应该做如下的改动。从对应的 SOAP 消息来看，在 SOAP 1.2中的 role 属性变成了 actor 属性。

```
112 [MessageContract(IsWrapped =true, WrapperNamespace="http://
113   www.artech.com/")]public class Customer
114 {
```

```

115     //其他成员
116     [MessageHeader(Name="CustomerNo", Namespace =
117         "http://www.artech.com/" ,MustUnderstand
118         = true, Relay=true, Actor=
119         "http://schemas.xmlsoap.org/soap/
120         actor/ultimateReceiver" )]
121     public Guid ID
122     { get; set; }
123 }
124 <s:Envelope xmlns:a="http://www.w3.org/2005/08/
125     addressing" xmlns:s=
126     "http://schemas.xmlsoap.org/soap/envelope/">
127     <s:Header>
128         .....
129         <h:CustomerNo s:actor="http://schemas.
130             xmlsoap.org/soap/actor/
131                 ultimateReceiver" s:mustUnderstand="1"
132                 xmlns:h="http://www.artech.
133                     com/">e48a8897-c644-49f8-b5e7-
134                     cd16be4c75b7</h:CustomerNo>
135     </s:Header>
136     .....
137 </s:Envelope>

```

### MessageBodyMemberAttribute

MessageBodyMemberAttribute 应用于属性或字段成员，应用了该特性的属性或字段的内容将会出现在 SOAP 的主体部分。MessageBodyMemberAttribute 的定义显得尤为简单，仅仅具有一个 Order 对象，用于控制成员在 SOAP 消息主体中出现的位置。默认的排序规则是基于字母排序的。

可能细心的读者会问，为什么 MessageHeaderAttribute 中没有这样的 Order 属性呢？原因很简单，MessageHeaderAttribute 定义的是单个 SOAP 报头，在 6.1 节介绍 SOAP 和 WS-Addressing 时，我提到过 SOAP 消息报头集合中的每个报头元素与次序无关。而 MessageBodyMemberAttribute 则是定义 SOAP 主体的某个元素，主体成员之间的次序是契约的一个重要组成部分。所以 MessageHeaderAttribute 不叫 MessageHeaderMemberAttribute。

```

132 [AttributeUsage(AttributeTargets.Field |
133     AttributeTargets.Property, Inherited = false)]
134 public class MessageBodyMemberAttribute :
135     MessageContractMemberAttribute
136 {
137     public int Order { get; set; }

```

### 6.3.1 消息契约的定义

消息契约和数据契约一样，都是定义在数据（而不是功能）类型上。不过数据契约旨在定义数据的结构（将数据类型与 XSD 进行匹配），而消息契约则更多地关注于数据的成员具体在 SOAP 消息中的表示。消息契约通过以下3个特性进行定义：MessageContractAttribute、MessageHeaderAttribute、MessageBodyMemberAttribute。MessageContractAttribute 应用于类型上，MessageHeaderAttribute 和 MessageBodyMemberAttribute 则应用于属性或字段成员上，以表明相应的数据成员是一个基于 SOAP 报头的成员还是 SOAP 主体的成员。先来简单介绍一下这3个特性。

#### MessageContractAttribute

通过在一个类或结构（Struct）上应用 MessageContractAttribute 使之成为一个消息契约。从 MessageContractAttribute 的定义来看，MessageContractAttribute 大体上具有以下两种类型的属性成员：

ProtectionLevel 和 HasProtectionLevel：表示保护级别，在服务契约中已经对保护级别作了简单的介绍，WCF 中通过 System.Net.Security.ProtectionLevel 枚举定义消息的保护级别。一般有3种可选的保护级别：None、Sign 和 EncryptAndSign。在下一卷关于 WCF 安全的章节中还将对此进行详细介绍。

IsWrapped、WrapperName、WrapperNamespace：IsWrapped 表述的含义为是否为定义的主体成员（一个或者多个）添加一个额外的根节点。WrapperName 和 WrapperNamespace 则表述该根节点的名称和命名空间。IsWrapped、WrapperName、WrapperNamespace 默认分别为 true、类型名称和 <http://tempuri.org/>。

```
1  [AttributeUsage(AttributeTargets.Struct |
AttributeTargets.Class, AllowMultiple = false)]
2  public sealed class MessageContractAttribute : Attribute
3  {
4      //其他成员
5      public bool          HasProtectionLevel { get; }
6      public ProtectionLevel ProtectionLevel { get; set; }
7
8      public bool          IsWrapped { get; set; }
9      public string        WrapperName { get; set; }
10     public string        WrapperNamespace { get; set; }
11 }
```

在下面的代码中通过应用 MessageContractAttribute 使 Customer 类型成为一个消息契约。ID 和 Name 属性通过应用 MessageHeaderAttribute 定义成消息报头成员，而 Address 属性则通过 MessageBodyMemberAttribute 定义成消息主体成员。后面的 XML 体现的是 Customer 对象在 SOAP 消息中的表现形式。

```
12 [MessageContract]
13 public class Customer
14 {
15     [MessageHeader(Name = "CustomerNo", Namespace =
16         "http://www.artech.com/")]
17     public Guid ID
18     { get; set; }
19
20     [MessageHeader(Name = "CustomerName", Namespace =
21         "http://www.artech.com/")]
22     public string Name
23     { get; set; }
24
25     [MessageBodyMember(Namespace = "http://www.artech.com/")]
26     public string Address
27     { get; set; }
28 }
29 <s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
30     xmlns:s="http://www.w3.org/2003/05/soap-envelope">
31     <s:Header>
32
33                                     <a:Action
s:mustUnderstand="1">http://tempuri.org/IOrderManager/
34         ProcessOrder</a:Action>
35     <h:CustomerName xmlns:h="
http://www.artech.com/">Foo
36     </h:CustomerName>
37     <h:CustomerNo xmlns:h="http://
www.artech.com/">2f62405b-a472-4d1c-
38     8c03-b888f9bd0df9</h:CustomerNo>
39     </s:Header>
40     <s:Body>
41         <Customer xmlns="http://tempuri.org/">
42             <Address xmlns="http://
www.artech.com/">#328, Airport Rd,
43             Industrial Park, Suzhou
44             Jiangsu Province</Address>
45         </Customer>
46     </s:Body>
47 </s:Envelope>
```

如果我们将 IsWrapped 的属性设为 false，那么套在 Address 节点外的 Customer 节点将会从 SOAP 消息中去除。

```
46 [MessageContract(IsWrapped = false)]
47 public class Customer
48 {
49     //省略成员
50 }
51 <s:Envelope xmlns:a="http://www.w3.org/2005/08/
addressing" xmlns:s="http://www.w3.org/2003/05/soap-envelope">
52     .....
53     <s:Body>
54         <Address xmlns="http://
www.artech.com/">#328, Airport Rd, Industrial
55             Park, Suzhou Jiangsu Province</Address>
56     </s:Body>
57 </s:Envelope>
```

同样可以自定义这个主体封套（Wrapper）的命名和命名空间。下面我们将 MessageContractAttribute 的 WrapperName 和 WrapperNamespace 属性设为 Cust 和 <http://www.artech.com/>。

```
58 [MessageContract(IsWrapped = true, WrapperName
= "Cust", WrapperNamespace =
59     "http://www.artech.com/")]
60 public class Customer
61 {
62     //省略成员
63 }
64 <s:Envelope xmlns:a="http://www.w3.org/
2005/08/addressing" xmlns:s=
65     "http://www.w3.org/2003/05/soap-envelope">
66     .....
67     <s:Body>
68         <Cust xmlns="http://www.artech.com/">
69             <Address>#328, Airport Rd,
Industrial Park, Suzhou Jiangsu
70                 Province</Address>
71         </Cust>
72     </s:Body>
73 </s:Envelope>
```

### MessageHeaderAttribute

MessageHeaderAttribute 和 MessageBodyMemberAttribute 分别用于定义消

息报头成员和消息主体成员，它们都有一个共同的基类：`System.ServiceModel.MessageContractMemberAttribute`。`MessageContractMemberAttribute` 定义了以下属性成员：`HasProtectionLevel`、`ProtectionLevel`、`Name` 和 `Namespace`。

```
74 public abstract class MessageContractMemberAttribute : Attribute
75 {
76     public bool HasProtectionLevel { get; }
77     public ProtectionLevel ProtectionLevel { get; set; }
78
79     public string Name { get; set; }
80     public string Namespace { get; set; }
81 }
```

通过在属性或字段成员上应用 `MessageHeaderAttribute` 使之成为一个消息报头成员。`MessageHeaderAttribute` 定义了以下3个属性，阅读了本章6.1节的内容或对于 SOAP 规范有一定了解的读者，相信对它们不会陌生。

**Actor:** 表示处理该报头的目标节点 (SOAP Node)，SOAP 1.1中对应的属性 (Attribute) 为 `actor`，SOAP 1.2中就是我们介绍的 `role` 属性。

**MustUnderstand:** 表述 Actor (SOAP 1.1) 或 Role (SOAP 1.2) 定义的 SOAP 节点是否必须理解并处理该节点。对应的 SOAP 报头属性为 `mustUnderstand`。

**Relay:** 对应的 SOAP 报头属性为 `relay`，表明该报头是否需要传递到下一个 SOAP 节点。

```
82 [AttributeUsage(AttributeTargets.Field | AttributeTargets.Property,
83     AllowMultiple = false, Inherited = false)]
84 public class MessageHeaderAttribute : MessageContractMemberAttribute
85 {
86     public string Actor { get; set; }
87     public bool MustUnderstand { get; set; }
88     public bool Relay { get; set; }
89 }
```

同样使用上面定义的 Customer 消息契约，现在我们相应地修改了 ID 属性上的 `MessageHeaderAttribute` 设置：`MustUnderstand = true`，`Relay=true`，`Actor=http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver`。实际上是将相应的 SOAP 报头的目标 SOAP 节点定义成最终的消息接收者。由于 <http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver> 是 SOAP 1.2的预定义属性，所以这个消息契约之后在基于 SOAP 1.2的消息版本中有效。后面给出的是对应的 SOAP 消息。

```
90 [MessageContract(IsWrapped =true, WrapperNamespace="http://
```



```

91     www.artech.com/")]public class Customer
92 {
93     //其他成员
94     [MessageHeader(Name="CustomerNo", Namespace = "http://
95         www.artech.com/" ,MustUnderstand =true, Relay=true, Actor=
96         "http://www.w3.org/2003/05/soap-
envelope/role/ultimateReceiver" )]
97     public Guid ID
98     { get; set; }
99
100 }
101 <s:Envelope xmlns:a="http://www.w3.org/2005/08/
addressing" xmlns:s=
102     "http://www.w3.org/2003/05/soap-envelope">
103     <s:Header>
104         .....
105         <h:CustomerNo s:role="http://www.w3.
org/2003/05/soap-envelope/role/
106             ultimateReceiver" s:mustUnderstand="1" s:relay="1" xmlns:h=
107             "http://www.artech.com/">5330c91a-
7fd7-4bf5-ae3e-4ba9bfef3d4d
108         </h:CustomerNo>
109     </s:Header>
110     .....
111 </s:Envelope>

```

<http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver> 在 SOAP 1.1 中对应的表示为: "<http://schemas.xmlsoap.org/soap/actor/ultimateReceiver>。如果在 SOAP 1.1 下, ID 成员对应的 MessageHeaderAttribute 应该做如下的改动。从对应的 SOAP 消息来看, 在 SOAP 1.2 中的 role 属性变成了 actor 属性。

```

112 [MessageContract(IsWrapped =true, WrapperNamespace="http://
113     www.artech.com/")]public class Customer
114 {
115     //其他成员
116     [MessageHeader(Name="CustomerNo", Namespace =
117         "http://www.artech.com/" ,MustUnderstand
= true, Relay=true, Actor=
118         "http://schemas.xmlsoap.org/soap/
actor/ultimateReceiver" )]
119     public Guid ID
120     { get; set; }
121 }
122 <s:Envelope xmlns:a="http://www.w3.org/2005/08/

```

```

addressing" xmlns:s=
123  "http://schemas.xmlsoap.org/soap/envelope/">
124    <s:Header>
125      .....
126      <h:CustomerNo s:actor="http://schemas.
xmlsoap.org/soap/actor/
127                               ultimateReceiver"    s:mustUnderstand="1"
xmlns:h="http://www.artech.
128                               com/">e48a8897-c644-49f8-b5e7-
cd16be4c75b7</h:CustomerNo>
129    </s:Header>
130    .....
131  </s:Envelope>

```

### MessageBodyMemberAttribute

MessageBodyMemberAttribute 应用于属性或字段成员，应用了该特性的属性或字段的内容将会出现在 SOAP 的主体部分。MessageBodyMemberAttribute 的定义显得尤为简单，仅仅具有一个 Order 对象，用于控制成员在 SOAP 消息主体中出现的位置。默认的排序规则是基于字母排序的。

可能细心的读者会问，为什么 MessageHeaderAttribute 中没有这样的 Order 属性呢？原因很简单，MessageHeaderAttribute 定义的是单个 SOAP 报头，在 6.1 节介绍 SOAP 和 WS-Addressing 时，我提到过 SOAP 消息报头集中的每个报头元素与次序无关。而 MessageBodyMemberAttribute 则是定义 SOAP 主体的某个元素，主体成员之间的次序是契约的一个重要组成部分。所以 MessageHeaderAttribute 不叫 MessageHeaderMemberAttribute。

```

132 [AttributeUsage(AttributeTargets.Field |
AttributeTargets.Property, Inherited = false)]
133 public class MessageBodyMemberAttribute :
MessageContractMemberAttribute
134 {
135     public int Order { get; set; }
136 }

```

### 6.3.2 案例演示：基于消息契约的方法调用是如何格式化成消息的？（1）

在上面一章中，介绍了 MessageFormatter 在 WCF 体系中所起的作用：ClientMessageFormatter 和 DispatchMessageFormatter 分别在客户端和服务端，根据操作的描述（Operation Description），借助于相应的序列化器（Serializer）实现了方法调用与消息之间的转换。接下来，将通过一个实实在在的案例程序为大家演示如何通过 ClientMessageFormatter 将输入参数转换为基于当前服务操作的 Message。由于本节的主题

是消息契约，所以在这里我们将转换对象限定为消息契约。不过，不论是消息参数还是一般的可序列化对象，其转换过程都是一样的。

### 步骤一 创建消息契约

本案例模拟一个订单处理的 WCF 应用，我们首先定义如下一个 Order 类型。Order 是一个消息契约，属性 OrderID 和 Date 通过 MessageHeaderAttribute 定义成消息报头，作为主体的 Details 的类型 OrderDetails 被定义成集合数据契约。OrderDetails 的元素类型是数据契约 OrderDetail，代表订单中每笔产品明细。

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.Serialization;
4  using System.ServiceModel;
5  namespace Artech.TypedMessage
6  {
7      [MessageContract]
8      public class Order
9      {
10         [MessageHeader(Namespace = "http://www.artech.com/")]
11         public Guid OrderID
12         { get; set; }
13
14         [MessageHeader(Namespace = "http://www.artech.com/")]
15         public DateTime Date
16         { get; set; }
17
18         [MessageBodyMember]
19         public OrderDetails Details
20         { get; set; }
21
22         public override string ToString()
23         {
24             return string.Format("Oder ID:
25 {0}\nDate: {1}\nDetail Count:
26 {2}", this.OrderID,this.Date.
27 ToShortDateString(),this.Details.
28 Count);
29
30         }
31     }
32
33     [CollectionDataContract(ItemName = "Detail",Namespace
34     ="http://www.artech.com/")]
35     public class OrderDetails : List<OrderDetail>
```

```

33     { }
34
35     [DataContract(Namespace ="http://www.artech.com/")]
36     public class OrderDetail
37     {
38         [DataMember]
39         public Guid ProductID
40         { get; set; }
41
42         [DataMember]
43         public int Quantity
44         { get; set; }
45     }
46 }

```

## 步骤二 创建 **MessageFormatter**

本案例的目的在于重现WCF如何通过ClientMessageFormatter实现将输入参数序列化请求消息，以及通过DispatchMessageFormatter实现将请求消息反序列化成输入参数。根据使用的序列化器的不同，WCF中定义了两种典型的MessageFormatter：一种是基于DataContractSerializer的DataContractSerializerOperationFormatter；另一种则是基于XmlSerializer的XmlSerializerOperationFormatter。由于DataContractSerializerOperationFormatter是默认的消息Formatter，所以我们案例就采用DataContractSerializerOperationFormatter。

我们的任务就是创建这个DataContractSerializerOperationFormatter。由于这是一个定义在System.ServiceModel.Dispatcher命名空间下的内部（Internal）类型，所以只能通过反射的机制调用构造函数来创建这个对象。DataContractSerializerOperationFormatter定义了一个构造函数，3个输入参数类型分别为：OperationDescription、DataContractFormatAttribute和DataContractSerializerOperationBehavior。

```

47 internal class DataContractSerializerOperationFormatter :
    OperationFormatter
48 {
49     //其他成员
50     public DataContractSerializerOperationFormatter
    (OperationDescription
51         description, DataContractFormatAttribute
    dataContractFormatAttribute,

```

```

52     DataContractSerializerOperationBehavior
serializerFactory);
53 }

```

为此我们定义下面一个辅助方法 `CreateMessageFormatter<TFormatter, TContract>`。`TFormatter` 代表 `MessageFormatter` 的两个接口：`IClientMessageFormatter` 和 `IDispatchMessageFormatter` (`DataContractSerializerOperationFormatter` 同时实现了这两个接口)，`TContract` 则是服务契约的类型。参数 `operationName` 为当前操作的名称。代码不算复杂，主要的流程如下：通过服务契约类型创建 `ContractDescription`，根据操作名称得到 `OperationDescription` 对象。通过反射机制调用 `DataContractSerializerOperationFormatter` 的构造函数创建该对象。

```

54 static TFormatter CreateMessageFormatter<TFormatter,
DataContract>(string operationName)
55 {
56     ContractDescription contractDesc = ContractDescription.GetContract
57         (typeof(TContract));
58     var operationDescs = contractDesc.
Operations.Where(op => op.Name ==
59         operationName);
60     if(operationDescs.Count() == 0)
61     {
62         throw new ArgumentException("
operationName","Invalid operation
63             name.");
64     }
65     OperationDescription operationDesc = operationDescs.ToArray()[0];
66     string formatterTypeName = "System.
ServiceModel.Dispatcher.
67         DataContractSerializerOperation
Formatter,System.ServiceModel,
68             Version=3.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089";
69     Type formatterType = Type.GetType
(formatterTypeName);
70     ConstructorInfo constructor = formatterType.GetConstructor(new
Type[]
71         { typeof(OperationDescription),
typeof(DataContractFormatAttribute),
72         typeof(DataContractSerializerOperationBehavior) });
73     return (TFormatter)constructor.Invoke
(new object[] { operationDesc,
74         new DataContractFormatAttribute(), null });
75 }

```

MessageFormatter 已经创建出来了，序列化与反序列化的问题也就很简单了。为此我定义了以下两个辅助方法：SerializeRequest<TContract> 和 DeserializeRequest<TContract>，具体实现就是调用创建出来的 MessageFormatter 的同名方法。

```
76 static Message SerializeRequest<TContract>(MessageVersion
messageVersion,
77 string operationName, params object[] values)
78 {
79     IClientMessageFormatter formatter = CreateMessageFormatter
80         <IClientMessageFormatter, TContract>(operationName);
81     return formatter.SerializeRequest(messageVersion, values);
82 }
83
84 static void DeserializeRequest<TContract>(Message message, string
85 operationName, object[] parameters)
86 {
87     IDispatchMessageFormatter formatter = CreateMessageFormatter
88         <IDispatchMessageFormatter, TContract>(operationName);
89     formatter.DeserializeRequest(message, parameters);
90 }
```

### 6.3.2 案例演示：基于消息契约的方法调用是如何格式化消息的？（2）

#### 步骤三 通过 MessageFormatter 实现消息的格式化

现在我们通过一个简单的例子来演示通过上面创建的 MessageFormatter 实现对消息的格式化。由于 MessageFormatter 进行序列化和反序列化依赖于操作的描述（消息的结构本来就是由操作决定的），为此我们定义了一个服务契约 IOrderManager。操作 ProcessOrder 将消息契约 Order 作为唯一的参数。

```
1 using System.ServiceModel;
2 namespace Artech.TypedMessage
3 {
4     [ServiceContract]
5     public interface IOrderManager
6     {
7         [OperationContract]
8         void ProcessOrder(Order order);
9     }
10 }
```

在下面的代码中，先调用 SerializeRequest<IOrderManager>方法将 Order 对象进行序列化并生成 Message 对象，该过程实际上体现了 WCF 的客户端框架是如何通过

ClientMessageFormatter 将操作方法调用连同输入参数转换成请求消息的。随后，调用 DeserializeRequest<IOrderManager>方法将 Message 对象反序列化成 Order 对象，该过程则代表 WCF 的服务端框架是如何通过 DispatchMessageFormatter 将请求消息反序列化成输入参数的。

```
11 OrderDetail detail1 = new OrderDetail
12 {
13     ProductID = Guid.NewGuid(),
14     Quantity = 666
15 };
16
17 OrderDetail detail2 = new OrderDetail
18 {
19     ProductID = Guid.NewGuid(),
20     Quantity = 999
21 };
22
23 Order order = new Order
24 {
25     OrderID = Guid.NewGuid(),
26     Date = DateTime.Today,
27     Details = new OrderDetails { detail1, detail2 }
28 };
29 //模拟 WCF 客户端的序列化
30 Message message =
SerializeRequest<IOrderManager>(MessageVersion.Default,
31     "ProcessOrder", order);
32 MessageBuffer buffer = message.
CreateBufferedCopy(int.MaxValue);
33 WriteMessage(buffer.CreateMessage(), "message.xml");
34
35 //模拟 WCF 服务端的反序列化
36 object[] DeserializedOrder = new object[] { null };
37 DeserializeRequest<IOrderManager>(buffer.
CreateMessage(), "ProcessOrder",
38     DeserializedOrder);
39 Console.WriteLine(DeserializedOrder[0]);
```

下面的 XML 表示调用 SerializeRequest<IOrderManager>生成的 SOAP 消息。程序最终的输出结果也表明了反序列化的成功执行。

```
40 <s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
41     xmlns:s="http://www.w3.org/2003/05/soap-envelope">
42     <s:Header>
```

```

43                                     <a:Action
s:mustUnderstand="1">http://tempuri.org/IOrderManager/
44             ProcessOrder</a:Action>
45             <h>Date xmlns:h="http://www.artech.com/">2008-12-
21T00:00:00+08:00
46         </h>Date>
47         <h:OrderID xmlns:h="http://www.artech.com/">cd94a6f0-7e21-4ace-
83f7-
48             2ddf061cfbbe</h:OrderID>
49     </s:Header>
50     <s:Body>
51         <Order xmlns="http://tempuri.org/"
52             <Details xmlns:d4p1="http://www.artech.com/" xmlns:i=
53                 "http://www.w3.org/2001/XMLSchema-instance">
54             <d4p1:Detail>
55                 <d4p1:ProductID>bc2a186d-
56                 569a-4146-9b97-3693248104c0
57                 </d4p1:ProductID>
58                 <d4p1:Quantity>666</d4p1:Quantity>
59             </d4p1:Detail>
60             <d4p1:Detail>
61                 <d4p1:ProductID>72687c23-c2b2-
62                 4451-b6c3-da6d040587fc
63                 </d4p1:ProductID>
64                 <d4p1:Quantity>999</d4p1:Quantity>
65             </d4p1:Detail>
66             </Details>
67         </Order>
68     </s:Body>
69 </s:Envelope>
70
71 Oder ID: cd94a6f0-7e21-4ace-83f7-2ddf061cfbbe
72 Date: 12/21/2008
73 Detail Count: 2

```

## 6.4 消息编码 (Message Encoding)

消息作为 WCF 进行通信的唯一媒介，最终需要通过写入传输层进行传递。而消息进行传输的一个前提或是一项必不可少的工作是对消息进行相应的编码。WCF 提供了一系列可供选择的编码方式，它们分别在互操作和性能方面各具优势。在本节我们将对各种编码方式进行消息的讨论。

### 6.4.1 序列化 (反序列化) 和编码 (解码)

有些 WCF 的开发人员，甚至包括一些比较资深的开发人员，往往对序列化和编码分得不



是很清楚，经常把序列化和编码混为一谈。对于 WCF 来说，序列化和编码具有明确的界限：序列化关注于如何将一个可序列化对象（数据契约和应用的 `SerializableAttribute` 特性或者实现了 `ISerializable` 接口）转换为 XML InfoSet，反序列化则是通过解析 XML Info 生成相应托管对象的过程。通过序列化得到的 XML InfoSet 必须通过相应的编码（Encoding）才能被写入传输层，基于某种传输协议在网络中传递，而当经过编码的信息抵达接收方，接收方需要通过相应的解码方式对二进制信息进行解码（Decoding），解码生成的 XML InfoSet 被交付给相应的序列化器进行反序列化。

从序列化和编码在 WCF 整个框架体系所处的层次来看，序列化（反序列化）通过相应的 `MessageFormatter`（`ClientMessageFormatter` 和 `DispatchMessage`）实现，而 `MessageFormatter` 最终又通过相应的 `Serializer`（`DataContractSerializer`、`XmlContractSerializer` 等）完成真正的序列化（反序列化）工作。`MessageFormatter` 处于 WCF 的服务模型层（`ServiceModel Layer`），而编码则是在发生在信道层（`Channel Layer`）。在第3章介绍绑定的时候，我们说绑定是信道层的缔造者，组成绑定的绑定元素创建各自的信道（`Channel`）并串联成一个用于处理消息的管道：信道栈（`Channel Stack`）。对于任何一种绑定类型，消息编码绑定元素（`Message Encoding Binding Element`）和传输绑定元素（`Transport Binding Element`）是不可或缺的，一般来说前者位于后者之上。在 WCF 真正的执行框架中，消息编码绑定元素用于创建相应的编码器（`Encoder`），发送端的传输信道在消息写入传输层之前通过该编码器对消息进行编码，而接收端的传输信道通过此编码器对接收的二进制数据进行解码。

从互操作性的角度来看，编码方法很大程度上决定了跨平台支持的能力。有的编码方式是平台无关的，有的则仅限于某种特定的平台。WCF 提供了3种典型的编码方式：`Binary`、`Text` 和 `MTOM`。`Binary` 以二进制的方式进行消息的编码，但是仅限于 .NET 平台之间的通信；`Text` 则提供与平台无关的基于文本的编码方式。`MTOM` 编码基于 `WS-MTOM` 规范，对于改善大规模二进制数据在 SOAP 消息的传输性能具有重大的意义，既然该编码方式遵循相应的规范，无疑这也是一种跨平台的编码方式。

#### 6.4.2 `XmlDictionary`、`XmlDictionaryWriter` 和 `XmlDictionaryReader` (1)

在正式介绍 WCF 消息编码之前，很有必要了解如下几个实现编码的核心对象：`XmlDictionary`、`XmlDictionary` 和 `XmlDictionaryWriter`。

##### **`XmlDictionary`**

`XmlDictionary`，顾名思义，它是一个字典，它是从事编码和解码的双方共享的一份“词汇表”。这样的说法可能有点抽象，我们不妨做一个类比。比如我说“WCF 是 .NET 平台下基于 SOA 的消息通信框架”，对于各位读者来说，这句话很好理解。如果我向另一个对计算机一窍不通的人说这句话，毫无疑问，对方是无论如何不能理解的。读者和我之所以能够通过

这样的语言进行交流，是因为我们之间具有相似的知识背景，在我们之间共享着相同的词汇表，对每个单词的含义具有一致的理解。而别人不能理解，是在于我和他之间的信息不对称，如果要让他理解，我必须用他所能理解的方式进行交流，在这种情形之下，我可能要花很多文字对这句话的一些术语进行详细的解释，比如什么是.NET 平台，什么是 SOA，什么又是通信框架。所以，交流的前提是双方具有相同的“词汇表”，双方就某个主题共享越多的相同“词汇”，交流就越容易，说的话将越简洁。

数据的编码也像我们日常的沟通和交流一样，编码的一方是“说”的一方，解码的一方是“听”的一方。说的一方按照它所掌握的“词汇表”对信息进行编码，对方只有具有相同的“词汇表”才能正常地解码。如果这个“词汇表”越详尽，编码后的内容容量就越小。内容的浓缩意味着什么？意味着网络流量的减少，意味着为你节省更多的带宽。而 XmlDictionary 就是这样的一个词汇表。

XmlDictionary 定义在 System.Xml 命名空间下，它是 System.Xml.XmlDictionaryString 的集合。XmlDictionaryString 相当于一个 KeyValuePair<int, string> 对象，它是一个键-值对，键和值的类型为 int 和 string。下面是 XmlDictionaryString 和 XmlDictionary 的定义。

```
1  public class XmlDictionaryString
2  {
3      //其他成员
4      public XmlDictionaryString(IXmlDictionary
dictionary, string value, int
5          key);
6      public IXmlDictionary Dictionary { get; }
7      public static XmlDictionaryString Empty { get; }
8      public int Key { get; }
9      public string Value { get; }
10
11 }
12 public class XmlDictionary : IXmlDictionary
13 {
14
15     //其他成员
16     public XmlDictionary();
17     public XmlDictionary(int capacity);
18     public virtual XmlDictionaryString
Add(string value);
19
20     public virtual bool TryLookup(int key,
out XmlDictionaryString result);
21     public virtual bool TryLookup(string value,
```

```

out XmlDictionaryString
22     result);
23     public virtual bool TryLookup(XmlDictionaryString value, out
24

```

通过下面的代码，创建了一个 `XmlDictionary` 对象，通过 `Add` 方法添加了3个 `XmlDictionaryString`。严格说来 `XmlDictionary` 并不是一个集合对象，因为它没有实现 `IEnumerable` 接口。通过 `Add` 方法只能指定 `XmlDictionaryString` 的 `Value`，`Key` 的值会以自增长的方式自动赋上。所以 `Customer`、`Name`、`Company` 3个元素的 `Key` 分别为0、1、2，这可以从最终输出结果中看出来。

```

25  IList<XmlDictionaryString> dictionaryStringList
= new List<XmlDictionaryString>();
26  XmlDictionary dictionary = new XmlDictionary();
27  dictionaryStringList.Add(dictionary.Add("Customer"));
28  dictionaryStringList.Add(dictionary.Add("Name"));
29  dictionaryStringList.Add(dictionary.Add("Company"));
30  foreach (XmlDictionaryString dictionaryString
in dictionaryStringList)
31  {
32      Console.WriteLine("Key:{0}\tValue:{1}",
dictionaryString.Key, dictionaryString.Value);
33  }
34  Key: 0    Value: Customer
35  Key: 1    Value: Name
36  Key: 2    Value: Company
37
38  XmlDictionaryWriter

```

`XmlDictionaryWriter` 和后面介绍的 `XmlDictionaryReader`，在 WCF 编码（解码）过程中具有举足轻重的地位，因为最终的编码和解码工作分别落在这个两个类上面。`XmlDictionaryWriter` 将 XML `InfoSet` 进行编码写入到流中，`XmlDictionaryReader` 将数据从流中读出并进行解码，生成相应的 XML `InfoSet`。

`XmlDictionaryWriter` 是一个继承自 `XmlWriter` 的抽象类，WCF 中定义了一系列具体的 `XmlDictionaryWriter`，它们直接或间接地继承自 `XmlDictionaryWriter`，为编码和解码提供了不同的实现。典型的 `XmlDictionaryWriter` 包括以下3个：

`XmlUTF8TextWriter`：提供基于文本的编码和解码实现。

`XmlBinaryWriter`：提供基于二进制的编码和解码实现。

`XmlMtomWriter`：提供基于 MTOM（Message Transmission Optimized Mechanism）的编码和解码实现。

上面3个类型定义在 System.Runtime.Serialization 程序集的 internal 类型，所以不能直接使用。XmlDictionaryWriter 定义了一系列的工厂方法以方便开发者创建这些对象。其中上面3种类型 XmlDictionaryWriter 对应的工厂方法分别为：CreateTextWriter、CreateBinaryWriter 和 CreateMtomWriter。

```
39 public abstract class XmlDictionaryWriter : XmlWriter
40 {
41     //其他成员
42     public static XmlDictionaryWriter
CreateBinaryWriter(Stream stream);
43     public static XmlDictionaryWriter
CreateBinaryWriter(Stream stream,
44         XmlDictionary dictionary);
45     public static XmlDictionaryWriter
CreateBinaryWriter(Stream stream,
46         XmlDictionary dictionary,
XmlBinaryWriterSession session);
47     public static XmlDictionaryWriter
CreateBinaryWriter(Stream stream,
48         XmlDictionary dictionary,
XmlBinaryWriterSession session, bool
49         ownsStream);
50
51     public static XmlDictionaryWriter
CreateDictionaryWriter(XmlWriter
52         writer);
53
54     public static XmlDictionaryWriter
CreateMtomWriter(Stream stream,
55         Encoding encoding, int maxSizeInBytes,
string startInfo);
56
57     public static XmlDictionaryWriter
CreateMtomWriter(Stream stream,
58         Encoding encoding, int maxSizeInBytes,
string startInfo, string boundary,
59         string startUri, bool writeMessageHeaders,
bool ownsStream);
60     public static XmlDictionaryWriter
CreateTextWriter(Stream stream);
61     public static XmlDictionaryWriter
CreateTextWriter(Stream stream,
62         Encoding encoding);
63     public static XmlDictionaryWriter
```

```

CreateTextWriter(Stream stream,
64     Encoding encoding, bool ownsStream);
65 }

```

这3种类型的 XmlDictionaryWriter 代表了 WCF 目前支持的3种典型的消息编码方式：Text、Binary 和 MTOM。接下来，我们将通过一个个具体的例子，来比较这3种不同的 XmlDictionaryWriter 经过编码后，产生的内容到底有何不同。

#### 6.4.2 XmlDictionary、XmlDictionaryWriter 和 XmlDictionaryReader (2)

##### XmlUTF8TextWriter (CreateTextWriter)

由于基于纯文本的编码与平台无关，所以它能为不同的厂商所支持，这和 SOA 跨平台互操作的主张一致，所以基于文本的编码是最为常用的编码方式。WCF 的 BasicHttpBinding、WsHttpBinding 及 WsDualHttpBinding 都会采用基于文本的编码。在 WCF 中，所有基于文本的编码工作最终会落在 XmlUTF8TextWriter 上面，由于该类是一个内部类型，我们只能通过 XmlDictionaryWriter 提供的 3 个静态工厂方法 CreateTextWriter 来创建 XmlUTF8TextWriter 对象。CreateTextWriter 方法的参数 stream 便是经过编码的二进制数组需要写入的流；encoding 表明采用的字符编码方式，在这里只有两种类型的字符编码支持：UTF8和 Unicode，这从 XmlUTF8TextWriter 的命名就可以看出来；至于 ownsStream，表明 XmlUTF8TextWriter 对象是否拥有对应的 stream 对象，如果是 true，则表明 XmlUTF8TextWriter 是 stream 的拥有者，XmlUTF8TextWriter 的关闭将伴随着 stream 的关闭，默认为 true。

```

1  public abstract class XmlDictionaryWriter : XmlWriter
2  {
3      //其他成员
4      public static XmlDictionaryWriter
CreateTextWriter(Stream stream);
5      public static XmlDictionaryWriter
CreateTextWriter(Stream stream,
6          Encoding encoding);
7      public static XmlDictionaryWriter
CreateTextWriter(Stream stream,
8          Encoding encoding, bool ownsStream);
9  }

```

下面是一个简单地使用 XmlUTF8TextWriter 进行编码的例子。在这里使用 XmlDictionary 的 CreateTextWriter 方法创建 XmlUTF8TextWriter 对象，对一个简单的 XML 文档（文档中仅仅具有一个 XML 元素）进行编码，然后输出经过编码后的字节长度、二进制表示和以文本显示的文档内容。代码后面是真实的输出。

```

10 MemoryStream stream = new MemoryStream();

```

```

11 using (XmlDictionaryWriter writer = XmlDictionaryWriter.
12     CreateTextWriter(stream, Encoding.UTF8))
13 {
14     writer.WriteStartDocument();
15     writer.WriteElementString("Customer",
16 "http://www.artech.com/", "Foo");
17     writer.Flush();
18
19     long count = stream.Position;
20     byte[] bytes = stream.ToArray();
21     StreamReader reader = new StreamReader(stream);
22     stream.Position = 0;
23     string content = reader.ReadToEnd();
24
25     Console.WriteLine("字节数为: {0}", count);
26     Console.WriteLine("编码后的二进制表示为: \n{0}",
27         BitConverter.ToString(bytes));
28     Console.WriteLine("编码后的文本表示为: \n{0}", content);
29 }
29 字节数为: 93
30 编码后的二进制表示为:
31 3C-3F-78-6D-6C-20-76-65-72-73-69-6F-6E-3D-22-31-
32 2E-30-22-20-65-6E-63-6F-64-69-6E-67-3D-22-75-
33 74-66-2D-38-22-3F-3E-3C-43-75-73-74-6F-6D-65-
34 72-20-78-6D-6C-6E-73-3D-22-68-74-74-70-3A-2F-
35 2F-77-77-77-2E-61-72-74-65-63-68-2E-63-6F-6D-
36 2F-22-3E-46-6F-6F-3C-2F-43-75-73-74-6F-6D-65-72-3E
37
38 编码后的文本表示为:
39 <?xml version="1.0" encoding="utf-8"?><Customer xmlns=
40 "http://www.artech.com/">Foo</Customer>
41
42 XmlBinaryWriter(CreateBinaryWriter)

```

XmlBinaryWriter 通过二进制的方式进行编码，所以它能够极大地减少编码后字节的大小，在进行网络传输的时候能够极大地节约网络带宽，获得最好的传输性能。但是，这种形式的编码并不具备跨平台的特性，仅限于客户端和服务端采用 WCF 的应用场景。

为了演示通过 XmlBinaryWriter 进行编码，我将上面的代码略加改动：通过调用 CreateBinaryWriter 创建 XmlBinaryWriter 对象。从最终的输出结果可以看出来，较之通过 TextUTF8TextWriter，通过 XmlBinary 编码后的字节数得到了极大的压缩（从原来的93变成了39），压缩率超过了50%。

```

36 MemoryStream stream = new MemoryStream();
37 using (XmlDictionaryWriter writer = XmlDictionaryWriter.
38     CreateBinaryWriter(stream))

```

```

39 {
40     //省略成员
41 }
42 字节数为: 39
43 编码后的二进制表示为:
44 40-08-43-75-73-74-6F-6D-65-72-08-16-68-
45 74-74-70-3A-2F-2F-77-77-77-2E-61-
46 72-74-65-63-68-2E-63-6F-6D-2F-99-03-46-6F-6F
47 编码后的文本表示为:
48 [省略不可读的编码内容]

```

如果我们查看 `XmlDictionaryWriter` 的 `WriteElementString` 方法, 会发现其具有5个重载, 其中3个是从 `XmlWriter` 中继承下来的(我们的代码使用的就是 `XmlWriter` 定义的方法), 其余两个是 `XmlDictionaryWriter` 自定义成员。与 `XmlWriter` 中继承下来的方法不同的是, 元素名称和命名空间通过 `XmlDictionaryString` 类型表示。实际上 `XmlDictionaryWriter` 的很多方法都同时提供以字符串和 `XmlDictionaryString` 表示的XML元素或属性名称和命名空间。在本节的开始我们就说了, `XmlDictionary` 是编码和解码双方共享的“词汇表”, 通过在编码过程中有效地使用它, 可以在很大程度上压缩编码后的字节数。

```

48 public abstract class XmlDictionaryWriter : XmlWriter
49 {
50     //其他成员
51     public void WriteElementString(XmlDictionaryString localName,
52         XmlDictionaryString namespaceUri, string value);
53     public void WriteElementString(string prefix, XmlDictionaryString
54         localName, XmlDictionaryString namespaceUri, string value);
55 }

```

相应地, `XmlDictionary` 也反映在 `CreateBinaryWriter` 静态方法上面。`CreateBinaryWriter` 方法比 `CreateTextWriter` 多了一些重载, 其中多了一个 `IXmlDictionary` 接口类型的参数 `dictionary`。

```

56 public abstract class XmlDictionaryWriter : XmlWriter
57 {
58     //其他成员
59     public static XmlDictionaryWriter
60     CreateBinaryWriter(Stream stream);
61     public static XmlDictionaryWriter
62     CreateBinaryWriter(Stream stream,
63         IXmlDictionary dictionary);
64     public static XmlDictionaryWriter
65     CreateBinaryWriter(Stream stream,
66         IXmlDictionary dictionary,
67         XmlBinaryWriterSession session);

```

```

64
65     public static XmlDictionaryWriter
CreateBinaryWriter(Stream stream,
66         XmlDictionary dictionary,
XmlBinaryWriterSession session, bool
67         ownsStream);
68 }

```

在现有代码的基础上，我做了一些修正，先创建 XmlDictionary 对象，将后面使用到的 XML 元素名称（Customer）和命名空间（<http://www.artech.com/>）定义成相应的 XmlDictionaryString，并添加到 XmlDictionary 中。在调用 CreateBinaryWriter 的时候指定该 XmlDictionary，并在调用 WriteElementString 方法的时候以 DictionaryString 的形式制定元素命名和命名空间。如果看了最终的输出结果，你可能会不敢相信自己的眼睛，字节长度变成了9（93=>39=>9）。之所以使用了 XmlDictionary 后的编码能够得到如此高的压缩率，就在于元素的名称和命名空间通过 Key-Value 的形式表示在了 XmlDictionary 中，在编码的时候会将 XML 中相应的 Value 内容替换成 int 型的 Key，这样做当然能够使得压缩率得到极大的提升了。

```

69 XmlDictionary dictionary = new XmlDictionary();
70 IList<XmlDictionaryString> dictionaryStrings = new
71     List<XmlDictionaryString>();
72 dictionaryStrings.Add(dictionary.Add("Customer"));
73 dictionaryStrings.Add(dictionary.Add("http://www.artech.com/"));
74
75 MemoryStream stream = new MemoryStream();
76 using (XmlDictionaryWriter writer = XmlDictionaryWriter.
77     CreateBinaryWriter(stream, dictionary))
78 {
79     writer.WriteStartDocument();
80     writer.WriteElementString(dictionaryStrings[0],
dictionaryStrings[1],
81         "Foo");
82     writer.Flush();
83     //其他操作
84 }
85 字节数为： 9
86 编码后的二进制表示为：
87 42-00-0A-02-99-03-46-6F-6F
88 编码后的文本表示为：
89 [省略不可读的编码内容]

```

#### **XmlMtomWriter (CreateMtomWriter)**

在很多分布式应用场景中，我们会通过 SOAP 消息传输一些大规模的二进制数据，比如



我们上传文件、图片、MP3甚至是视频。对于这些大块的二进制内容，如果采用 Binary 的编码方式，固然能够获得最好的编码压缩率，保证数据的快速传输，但是却不能获得跨平台的能力。如果采用纯文本的编码方式，基于 Base64 的编码方式会使编码后的内容显得非常冗余，而且这些冗余的数据会直接置于 SOAP 消息的主体中，使得 SOAP 消息十分庞大，从而影响 SOAP 消息正常的传输。为了解决这样的问题，MTOM (Message Transmission Optimization Mechanism) 应运而生。MTOM 兼具文本编码的跨平台能力（因为 MTOM 是 W3C 制定一个规范），又具有 Binary 编码高压缩率的优势。要想深入了解 MTOM 的消息传输优化机制，读者可以访问 W3C 的官方网站下载相关的文档。在这里，我仅仅是对该机制的实现作一个简单的介绍。

首先，二进制的內容仍然按照 Base64 的方式进行编码，然后对包含 `<xs:base64binary>` 的元素进行传输优化 (Transmission Optimization)。我们可以视这种优化为通过一种标准的、高压缩率的格式对其进行编码，这种格式是基于 XOP (XML-binary Optimized Packaging)。SOAP 消息在被传输的时候，通过一种称为 MIME Multipart/Related XOP Package 的形式发送。MIME Multipart/Related XOP Package，XOP 是经过对 `<xs:base64binary>` 元素进行优化编码后的数据包，Multipart/Related XOP 就是多个关联的 XOP，每个 XOP 数据包和 SOAP 封套 (SOAP Envelope) 是分开的，XOP 并不内嵌于 SOAP 封套中，它作为其附件 (Attachment) 单独传送，SOAP 封套保留一份 XOP 数据包的引用。

在 WCF 中，所有关于 MTOM 编码与解码相关的功能都通过 `XmlMtomWriter` 来完成，`XmlMtomWriter` 通过 `XmlDictionaryWriter` 的 `CreateMtomWriter` 静态方法创建。当我们通过 `XmlMtomWriter` 对于一个 XML Infoset 执行写操作时，最终生成的是一个具有报头 (Header) 和主体 (Body) 的 MIME Multipart/Related XOP Package，XML Infoset 的内容经过编码被放到主体部分。参数 `startInfo` 表示该 XML Infoset 对应 Content-Type 的 type 属性，对于 SOAP 自然就是 "Application/soap+xml"，而 `boundary` 则表示分隔符，`startUri` 作为 Content-ID，而 `writeMessageHeaders` 参数则表示是否写入 MIME Multipart/Related XOP Package 的报头内容。

```
90 public abstract class XmlDictionaryWriter : XmlWriter
91 {
92     //其他成员
93     public static XmlDictionaryWriter
CreateMtomWriter(Stream stream,
94         Encoding encoding, int maxSizeInBytes,
string startInfo);
95     public static XmlDictionaryWriter
CreateMtomWriter(Stream stream,
96         Encoding encoding, int maxSizeInBytes,
```

```

string startInfo, string boundary,
97     string startUri, bool writeMessageHeaders,
bool ownsStream);
98 }

```

接下来我通过一个简单的例子演示相同的XML元素通过XmlMtomWriter编码后又具有怎样的格式。在现有演示代码的基础上，通过调用 CreateMtomWriter 方法创建 XmlMtomWriter，并将 startInfo、boundary 和 startUri 分别指定为 "Application/soap+xml"、<http://www.artech.com/binary> 和 <http://www.artech.com/contentid>。从最后的结果我们可以看到：整个数据包包含两个部分：报头和主体，报头的主要作用在于指定整个数据包的 MIME 版本和 Content-Type。在 Content-Type 中 multipart/related;type="application/xop+xml" 是基于整个数据包，而 boundary="<http://www.artech.com/binary>";start="<http://www.artech.com/contentid>";start-info="Application/soap+xml" 则针对主体部分。

```

99 MemoryStream stream = new MemoryStream();
100 using (XmlDictionaryWriter writer =
    XmlDictionaryWriter.CreateMtomWriter
101 (stream, Encoding.UTF8, int.MaxValue,
    "Application/soap+xml", "http://
102 www.artech.com/binary", "http://
    www.artech.com/contentid", true, true))
103 {
104 //省略操作
105 }
106 字节数为: 517
107 编码后的二进制表示为:
108 4D-49-4D-45-2D-56-65-72-73-69-6F-6E-3A-20-31-(...省略...) 0A
109 编码后的文本表示为:
110 MIME-Version: 1.0
111 Content-Type: multipart/related;type="
    application/xop+xml";boundary="http://www.artech.
112 com/binary";start="<http://www.artech.com/contentid>";start-info=
113 "Application/soap+xml"
114 --http://www.artech.com/binary
115 Content-ID: <http://www.artech.com/contentid>
116 Content-Transfer-Encoding: 8bit
117 Content-Type: application/xop+xml;charset=utf-8;type=
118 "Application/soap+xml"
119
120 <?xml version="1.0" encoding="utf-8"?>
121 <Customer xmlns="http://www.artech.com/">Foo</Customer>
122 --http://www.artech.com/binary-

```

由于 MTOM 只有在针对大规模的二进制数据的传输时才能显示出优化的能力,对于文本内容反而因为多了很多必须的结构化描述信息,使得最终编码后的数据包都基于纯文本编码方式而冗余。MOTM 对于二进制数据的编码,我会在后续的部分为读者作演示。

### **XmlDictionaryReader**

有 XmlDictionaryWriter 就 必然 有 XmlDictionaryReader , XmlDictionaryWriter 对 XML Infoset 进行编码并将编码后的字节写入流中,而 XmlDictionaryReader 则读取二进制流并对其解码生成相应的 XML Infoset。WCF 同样 定义 了 3 个 具体 的 XmlDictionaryReader : XmlUTF8TextReader 、 XmlBinaryReader 和 XmlMtomReader,他们通过定义在 XmlDictionaryReader 的静态方法 CreateTextReader、CreateBinaryReader 和 CreateMtomReader 进行创建。

## **6.5 消息编码在 WCF 框架中的实现**

通过上面的介绍,我们知道了 WCF 所有与编码与解码相关的功能都实现在相应的 XmlDictioanryWriter 和 XmlDictionaryReader 中。但是在真正的 WCF 处理框架中,却并不直接使用 XmlDictioanryWriter 和 XmlDictionaryReader 对象,而通过相应的消息编码器 (MessageEncoder) 对其进行进一步封装,专门用于消息的编码和解码。

### **6.5.1 消息编码器 (MessageEncoder)**

消息编码器通过类型 MessageEncoder 表示, MessageEncoder 是定义在 System.ServiceModel.Channels 命名空间下的一个抽象类。从下面的定义中可以看出, MessageEncoder 主要包含两种类型的操作:读消息和写消息,分别通过 ReadMessage 和 WriteMessage 方法实现。此外,两个额外的方法,GetProperty<T>用于获取 MessageEncoder 相关的一些属性,IsContentTypeSupported 用于判断 MessageEncoder 是否支持某种类型的 MIME 类型。

```
1 public abstract class MessageEncoder
2 {
3     //其他成员
4     public virtual T GetProperty<T>() where T : class;
5     public virtual bool IsContentTypeSupported(string contentType);
6
7     public Message ReadMessage(ArraySegment<byte>
buffer, BufferManager
8         bufferManager);
9     public Message ReadMessage(Stream stream,
int maxSizeOfHeaders);
10     public abstract Message ReadMessage
```

```

(ArraySegment<byte> buffer,
11     BufferManager bufferManager, string contentType);
12     public abstract Message ReadMessage
(Array stream, int maxSizeOfHeaders,
13         string contentType);
14
15     public abstract void WriteMessage
(Message message, Stream stream);
16     public ArraySegment<byte> WriteMessage
(Message message, int
17         maxMessageSize, BufferManager bufferManager);
18     public abstract ArraySegment<byte>
WriteMessage(Message message, int
19         maxMessageSize, BufferManager
bufferManager, int messageOffset);
20
21     public abstract string ContentType { get; }
22     public abstract string MediaType { get; }
23     public abstract MessageVersion MessageVersion { get; }
24 }

```

与上面介绍的3种类型的 XmlDictionaryWriter/XmlDictionaryReader 相对应，WCF 同样定义了 MessageEncoder：TextMessageEncoder、BinaryMessageEncoder 和 MtomMessageEncoder 三种 MessageEncoder，它们分别封装了 XmlUTF8TextWriter/XmlUTF8TextReader、XmlBinaryWriter/XmlBinaryReader 和 XmlMtomWriter/XmlMtomReader。WCF 定义了3个相应的工厂类：TextMessageEncoderFactory、BinaryMessageEncoderFactory 和 MtomMessageEncoderFactory 用于创建相应的 MessageEncoder。它们共同继承一个抽象类：MessageEncoderFactory。通过只读属性 Encoder 得到相应的 MessageEncoder。

```

25 public abstract class MessageEncoderFactory
26 {
27     //其他成员
28     public abstract MessageEncoder Encoder { get; }
29 }

```

### 6.5.2 案例演示：通过 MessageCoder 对消息进行编码

接下来，我们来演示本章的第3个案例：如何通过 MessageCoder 对一个具体的 Message 对象进行编码。本案例主要演示 TextMessageCoder 和 MtomMessageEncoder 编码方式的对比。此外，为了演示 MTOM 对二进制数据的编码优化，我们创建一个基于二进制内容的 Message 对象，并将一个位图作为消息的主体。

我们先创建如下一个静态辅助方法 WriteMessage, 该方法通过 MessageEncoderFactory 得到的 MessageEncoder 对象将 Message 对象写入一个文件中。

```
1  static void WriteMessage(MessageEncoderFactory
2  encoderFactory, Message message, string fileName)
3  {
4      using (FileStream stream = new FileStream
5      (fileName, FileMode.Create,
6          FileAccess.Write, FileShare.Write))
7      {
8          encoderFactory.Encoder.WriteMessage(message, stream);
9      }
10 }
```

如果调用上面的方法, 首先需要创建 MessageEncoderFactory 对象。由于 TextMessageEncoderFactory 和 MtomMessageEncoderFactory 是一个内部类型, 不能直接实例化, 所以只能通过反射的机制创建两个 MessageEncoder。下面是 TextMessageEncoder 和 MtomMessageEncoderFactory 构造函数的定义。

```
9  internal class TextMessageEncoderFactory : MessageEncoderFactory
10 {
11     //其他成员
12     public TextMessageEncoderFactory(MessageVersion version, Encoding
13         writeEncoding, int maxReadPoolSize, int maxWritePoolSize,
14         XmlDictionaryReaderQuotas quotas);
15 }
16 internal class MtomMessageEncoderFactory : MessageEncoderFactory
17 {
18     //其他成员
19     public MtomMessageEncoderFactory(MessageVersion version, Encoding
20         writeEncoding, int maxReadPoolSize, int maxWritePoolSize, int
21         maxBufferSize, XmlDictionaryReaderQuotas quotas);
22 }
```

在下面的代码中, 先通过 Message 的静态方法 CreateMessage 创建 Message 对象, 需要注意的第 3 个参数是一个表示位图的 Bitmap 对象。然后通过反射创建 TextMessageEncoderFactory 和 MtomMessageEncoderFactory 对象, 并调用上面定义的辅助方法 WriteMessage。

```
23 Message message = Message.CreateMessage(MessageVersion.Default,
24     "http://www.artech.com/myaction", new Bitmap
25     (@\"C:\Users\Jinnan\Pictures\photo.jpg\"));
26
27 MessageBuffer buffer = message.CreateBufferedCopy(int.MaxValue);
28
```

```

27 //通过反射创建 TextMessageEncoderFactory
28 string encoderFactoryType = "System.ServiceModel.Channels.
29 TextMessageEncoderFactory, System.ServiceModel, Version=3.0.0.0,
30 Culture=neutral, PublicKeyToken=b77a5c561934e089";
31 MessageEncoderFactory encoderFactory = (MessageEncoderFactory)
32 Activator.CreateInstance(Type.GetType(encoderFactoryType),
33 MessageVersion.Default, Encoding.UTF8,
34 int.MaxValue, int.MaxValue, new
35 XmlDictionaryReaderQuotas());
36 WriteMessage(encoderFactory, buffer.CreateMessage(),
37 @"E:\message.text.xml");
38
39 //通过反射创建 MtomMessageEncoderFactory
40 encoderFactoryType = "System.ServiceModel.Channels.
41 MtomMessageEncoderFactory, System.ServiceModel, Version=3.0.0.0,
42 Culture=neutral, PublicKeyToken=b77a5c561934e089";
43 encoderFactory = (MessageEncoderFactory)Activator.CreateInstance(Type.
44 GetType(encoderFactoryType), MessageVersion.Default, Encoding.UTF8,
45 int.MaxValue, int.MaxValue, int.MaxValue, new
46 XmlDictionaryReaderQuotas());
47
48 WriteMessage(encoderFactory, buffer.CreateMessage(),
49 @"E:\message.mtom.xml");

```

下面给出的两段文字分别是通过TextMessageEncoder和MtomMessageEncoder对相同的Message对象进行编码后的结果。从中我们可以清晰地看出，TextMessageEncoder将位图进行Base64编码，编码后的内容以内联（Inline）的方式包含在SOAP主体中。而MtomMessageEncoder会生成一个MIME Multipart/Related XOP Package，SOAP封套作为其主体。编码后的字节和SOAP封套是分离的，SOAP的主体部分并不包含位图的内容，仅仅是通过Context-ID对分离的内容进行引用。

```

50 <s:Envelope xmlns:s="http://
www.w3.org/2003/05/soap-envelope"
51 xmlns:a="http://www.w3.org/2005/08/
addressing"><s:Header><a:Action
52 s:mustUnderstand="1">http://
www.artech.com/myaction</a:Action></
53 s:Header><s:Body><Bitmap xmlns="
http://schemas.datacontract.org/2004/07/
54 System.Drawing" xmlns:i="http://
www.w3.org/2001/XMLSchema-instance"
55 xmlns:x="http://www.w3.org/2001/XMLSchema"><Data
i:type="x:base64Binary"

```

```

56  xmlns="">/9j/4AAQSkZJRgABAQAAQABAAD/ (...省略...)
ZIz7V3gVcR/KPu+1UNWVfs3Qf6309jTkk47DW5/9k=</Data></Bitmap></s:Body></
57  s:Envelope>
58  MIME-Version: 1.0
59  Content-Type: multipart/related;type="
application/xop+xml";boundary=
60  "06a0ac15-70c6-47e9-8837-
ebc04a9ac1c2+id=1";start="<http://
61  tempuri.org/0/633655837835941838>";start-info="application/soap+xml"
62
63  --06a0ac15-70c6-47e9-8837-ebc04a9ac1c2+id=1
64  Content-ID: <http://tempuri.org/0/633655837835941838>
65  Content-Transfer-Encoding: 8bit
66  Content-Type: application/xop+xml;
67
68
69
70
71
72
73
74
75
76
77
78
79
80  charset=utf-8;type="application/soap+xml"
81
82  <s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
83  xmlns:a="http://www.w3.org/2005/08/
addressing"><s:Header><a:Action
84  s:mustUnderstand="1">http://
www.artech.com/myaction</a:Action></
85  s:Header><s:Body><Bitmap xmlns="
http://schemas.datacontract.org/2004/07/
86  System.Drawing" xmlns:i="http://
www.w3.org/2001/XMLSchema-instance"
87  xmlns:x="http://www.w3.org/2001/XMLSchema"><Data
i:type="x:base64Binary"
88  xmlns=""><xop:Include href=
89  "cid:http%3A%2F%2Ftempuri.org%2F1%2F633655837836161838"
90  xmlns:xop="http://www.w3.org/2004/08/xop/include"/></Data></Bitmap>
91  </s:Body></s:Envelope>

```

```

92 --06a0ac15-70c6-47e9-8837-ebc04a9ac1c2+id=1
93 Content-ID: <http://tempuri.org/1/633655837836161838>
94 Content-Transfer-Encoding: binary
95 Content-Type: application/octet-stream
96
97 [省略不可读的编码内容]
98 --06a0ac15-70c6-47e9-8837-ebc04a9ac1c2+id=1--

```

### 6.5.3 WCF 体系下的编码机制实现

在本章最后的部分，我们来介绍 WCF 体系下是如何对消息进行编码的。在客户端，以方法调用形式体现的服务访问通过 ClientMessageFormatter 生成请求消息。该请求消息最终通过绑定对象从服务模型层转到信道层。我们说绑定是绑定元素的有序组合，对于所有类型的绑定来说，有两个绑定类型是必不可少的：MessageEncodingBindingElement 和 TransportBindingElement。而消息的编码由这两个绑定元素共同完成。

上面我们介绍了 3 种编码方式：Text、Binary 和 MTOM；对应 3 种不同的 XmlDictionaryWriter/XmlDictionaryReader：XmlUTF8TextWriter/ XmlUTF8TextReader、XmlBinaryWriter/XmlBinaryReader 和 XmlMtomWriter/XmlMtomReader；3 种 XmlDictionaryWriter/XmlDictionaryReader 又对应着 3 种 MessageEncoder：TextMessageEncoder、BinaryMessageEncoder 和 MtomMessageEncoder；这 3 种不同的 MessageEncoder 又具有它们各自的 MessageEncoderFactory：TextMessageEncoderFactory、BinaryMessageEncoderFactory 和 MtomMessageEncoderFactory。最终这 3 种 MessageEncoderFactory 被 3 种相应的 MessageEncodingBindingElement 用于进行具体的编码。MessageEncodingBindingElement 通过 CreateMessageEncoderFactory 得到相应的 MessageEncoderFactory。

```

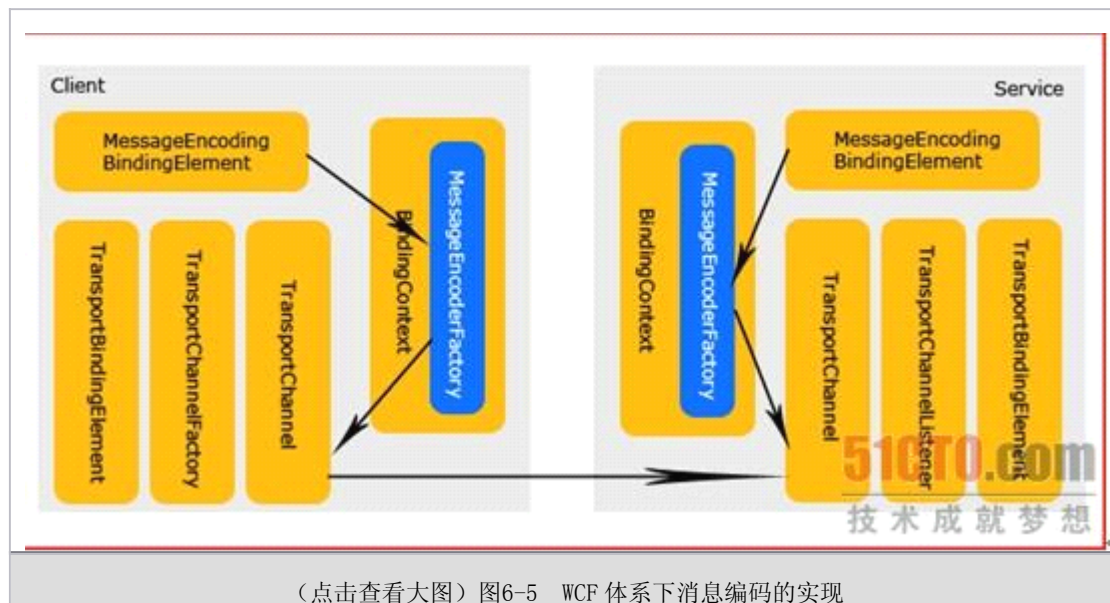
1  public abstract class MessageEncodingBindingElement :
BindingElement
2  {
3      //其他成员
4      public abstract MessageEncoderFactory
CreateMessageEncoderFactory();
5      public override T GetProperty<T>(BindingContext context) where T:
class;
6      public abstract MessageVersion
MessageVersion { get; set; }
7  }

```

对应着 3 种不同的 MessageEncoderFactory，WCF 定义了 3 种不同的 MessageEncodingBindingElement，它们分别是：TextMessageEncodingBindingElement、BinaryMessageEncodingBindingElement 和 MtomMessageEncodingBindingElement。



在介绍绑定的时候，我们说 BindingElement 创建相应的 ChannelFactory/ChannelListener，而 ChannelFactory/ChannelListener 最终创建相应的 Channel 进行消息的处理。这种说法是不准确的，并不是所有的 BindingElement 都会创建 Channel，实际上没有专门用于编码的 Channel，具体的编码工作是 TransportChannel 完成的。图6-5解释了 WCF 进行消息编码的本质。



当通过绑定对象创建信道栈的时候，MessageEncodingBindingElement 的 BuildChannelFactory/BuildChannelListener 方法首先被调用，MessageEncodingBindingElement 会创建相应的 MessageEncoderFactory 对象，将其置于当前的 BindingContext 中。然后 TransportBindingElement 的 BuildChannelFactory/BuildChannelListener 方法被调用，并创建 TransportChannelFactory/TransportChannelListener 对象，TransportChannelListener 和 TransportChannelFactory 创建 TransportChannel 用于请求监听和消息发送，与此同时 TransportChannel 会将 MessageEncoderFactory 从 BindingContext 获取下来用于消息的解码和编码。

## 第7章 服务寄宿

(Service Hosting)

任何一个程序都需要运行于一个确定的进程中，进程是一个容器，其中包含程序实例运行所需的资源。同理，一个 WCF 服务的监听与执行同样需要通过一个进程来承载。我们将为 WCF 服务创建或指定一个进程的方式称为服务寄宿 (Service Hosting)。服务寄宿的本质是通过某种方式创建或指定一个进程，用以监听服务的请求和执行服务操作，为服务提供一个运行环境。

服务寄宿的方式大体分两种：一种是为一组 WCF 服务创建一个托管的应用程序，通过手

工启动程序的方式对服务进行寄宿。所有托管的应用程序均可作为 WCF 服务的宿主，比如 Console 应用、Windows Forms 应用和 ASP.NET 应用等，我们把这种服务寄宿方式称为自我寄宿 (Self Hosting)；另一种则是通过操作系统现有的进程激活方式为 WCF 服务提供宿主，Windows 下的进程激活手段包括 IIS、Windows Service 和 WAS (Windows Process Activation Service) 等。

服务寄宿的手段是为一个 WCF 服务类型创建一个 ServiceHost 对象（或者任何继承自 ServiceHostBase 的对象）。无论采用哪种寄宿方式，在为某个服务创建 ServiceHost 的过程中，WCF 框架内部会执行一系列的操作，其中最重要的步骤就是为服务创建服务描述。我们先来介绍服务描述。

## 7.1 服务描述 (Service Description)

WCF 服务描述通过类型 System.ServiceModel.Description.ServiceDescription 表示，ServiceDescription 对象是 WCF 服务运行时的描述。除了包含 WCF 服务的一些基本信息，比如服务的名称、命名空间和 CLR 类型等，ServiceDescription 还包含服务所有终结点和服务行为的描述。

### 7.1.1 ServiceDescription 与 ServiceBehavior

从下面 ServiceDescription 的定义可以看出，ServiceDescription 中定义了一系列属性，它们的含义如下：

Behaviors: 服务行为 (Service Behavior) 的集合。

ConfigurationName: 服务在配置文件中的名称，默认为服务类型的全名（命名空间+类型名称）。

Name: 服务的名称，默认为服务类型名称（不包含命名空间）。

Namespace: 服务的命名空间，默认为“<http://tempuri.org/>”。

ServiceType: 服务的 CLR 类型。

```
1 public class ServiceDescription
2 {
3     //其他成员
4     public KeyedByTypeCollection<IServiceBehavior> Behaviors { get; }
5     public string ConfigurationName { get; set; }
6     public ServiceEndpointCollection Endpoints { get; }
7     public string Name { get; set; }
8     public string Namespace { get; set; }
9     public Type ServiceType { get; set; }
```

## Name 与 Namespace

ServiceDescription 的 Name 和 Namespace 分别表示服务的名称和命名空间，这两个属性同样体现在服务发布的 WSDL 中。可以通过 ServiceBehaviorAttribute 的 Name 和 Namespace 属性进行设定。ServiceDescription 的 Name 和 Namespace 的默认值分别为服务类型名称和<http://tempuri.org/>，所以下面两种定义是等效的。

```

11 [ServiceBehavior]
12 public class CalculatorService : ICalculator
13 {
14     //省略成员
15 }
16 [ServiceBehavior(Name      =      "CalculatorService",      Namespace      =
"http://tempuri.org/")]
17 public class CalculatorService : ICalculator
18 {
19     //省略成员
20 }
```

而 ServiceDescription 的 Namespace 映射 WSDL 的目标命名空间 (targetNamespace), Name 则直接对应 <wsdl:service> 节点的 Name 属性。在下面的服务定义中，通过 ServiceBehaviorAttribute 将 Name 和 Namespace 设置为 “CalcService” 和 <http://www.artech.com/>，后面的 XML 体现了服务在 WSDL 中的表示方式。

```

21 [ServiceBehavior(Name      =      "CalcService",      Namespace      =
"http://www.artech.com/")]
22 public class CalculatorService : ICalculator
23 {
24     //省略成员
25 }
26 <?xml version="1.0" encoding="utf-8"?>
27 <wsdl:definitions      name="CalcService"      targetNamespace=
http://www.artech.com/
28     ...>
29     .....
30     <wsdl:service name="CalcService">
31         .....
32     </wsdl:service>
33 </wsdl:definitions>
34 ConfigurationName
```

ServiceDescription 的 ConfigurationName 表示服务的配置名称，同样可以通过 ServiceBehaviorAttribute 特性的同名属性进行设定。在默认情况下，ConfigurationName

的值为服务类型的全名（命名空间 + 类型名称），下面两种服务的定义是等效的。

```
35 namespace Artech.ServiceDescriptionDemos
36 {
37     [ServiceBehavior]
38     public class CalculatorService : ICalculator
39     {
40         //省略成员
41     }
42 }
43 namespace Artech.ServiceDescriptionDemos
44 {
45     [ServiceBehavior(ConfigurationName =
46         "Artech.ServiceDescriptionDemos.
47         CalculatorService")]
48     public class CalculatorService : ICalculator
49     {
50         //省略成员
51     }
52 }
```

如果配置文件中<service>的 Name 属性更改了，在服务定义中需要通过 ServiceBehaviorAttribute 特性对 ConfigurationName 进行相应的修正，如下面的代码所示。

```
52 namespace Artech.ServiceDescriptionDemos
53 {
54     [ServiceBehavior(ConfigurationName = "CalculatorService")]
55     public class CalculatorService : ICalculator
56     {
57         //省略成员
58     }
59 }
60 <?xml version="1.0" encoding="utf-8" ?>
61 <configuration>
62     <system.serviceModel>
63         <services>
64             <service name="CalculatorService">
65                 .....
66             </service>
67         </services>
68     </system.serviceModel>
69 </configuration>
```

服务行为 (Service Behavior)

如果说契约 (Contract) 是涉及双边的描述 (契约是服务的提供者和服务消费者进行交互的依据), 那么行为 (Behavior) 就是基于单边的描述。客户端行为体现的是 WCF 进行服务调用的方式, 而服务端行为则体现了 WCF 的请求分发方式。行为是对 WCF 进行扩展的最为重要的方式, 按照行为作用域的不同, WCF 的行为大体包含以下4种:

服务行为 (Service Behavior): 基于服务本身的行为, 实现了接口 `System.ServiceModel.Description.IServiceBehavior`, 可以通过自定义特性 (Attribute) 或配置的方式进行指定。

终结点行为 (Endpoint Behavior): 基于某个服务终结点 (客户端或者服务端) 的行为, 实现了接口 `System.ServiceModel.Description.IEndpointBehavior`, 可以通过配置的方式进行指定。

契约行为 (Contract Behavior): 基于某个服务契约的行为, 作用于实现了该契约的所有服务 (服务端行为) 和基于该契约进行服务调用的服务代理 (客户端行为), 实现了接口 `System.ServiceModel.Description.IContractBehavior`, 可以通过自定义特性 (Attribute) 的方式进行指定。

操作行为 (Operation Behavior): 基于服务中的某个操作, 可以通过自定义特性的方式将操作行为应用于服务契约的某一个操作契约上, 也可以直接应用于服务的操作方法上。

在 `ServiceDescription` 中, 类型为 `KeyedByTypeCollection<IServiceBehavior>` 的 `Behaviors` 属性表示服务所有的服务行为集合。所有的服务行为都实现了 `System.ServiceModel.Description.IServiceBehavior` 接口。`IServiceBehavior` 的定义如下, 从中可以看出 `IServiceBehavior` 定义了如下3个方法。

注: `KeyedByTypeCollection<T>` 可以看成是以 `T` 实例为 Value, Value 对象真实类型为 Key 的 Dictionary, 可以通过类型定位并获取相应的成员对象。

`AddBindingParameters`: 为某个自定义绑定元素 (Custom Binding Element) 添加绑定参数, 以指导或确保绑定元素的正常操作, 比如通过设置的绑定参数创建和初始化相应的信道。

`ApplyDispatchBehavior`: 通过改变 WCF 服务端分发系统的属性, 或者添加/替换分发系统中用以实现某种分发操作的可扩展对象, 进而改变服务分发的行为。

`Validate`: 通过检验服务描述, 用以保证当前服务设置的正确性及后续工作的正常执行。

```
70 public interface IServiceBehavior
71 {
72     void AddBindingParameters(ServiceDescription serviceDescription,
73         ServiceHostBase serviceHostBase, Collection<ServiceEndpoint>
74         endpoints, BindingParameterCollection bindingParameters);
```

```

75     void ApplyDispatchBehavior(ServiceDescription serviceDescription,
76         ServiceHostBase serviceHostBase);
77         void Validate(ServiceDescription serviceDescription,
ServiceHostBase
78         serviceHostBase);
79     }

```

WCF 为我们预定义了一系列的 ServiceBehavior, ServiceBehaviorAttribute 就是其中之一。ServiceBehaviorAttribute 不仅仅是一个自定义特性 (Custom Attribute), 实际上它本身就是一个实现了 IServiceBehavior 的服务行为。

```

80 [AttributeUsage(AttributeTargets.Class)]
81 public sealed class ServiceBehaviorAttribute :
Attribute, IServiceBehavior
82 {
83     //省略成员
84 }

```

对于其他一些预定义服务行为, 比如用于实现与 ASP.NET 兼容的 `AspNetCompatibilityRequirementsAttribute`, 用于进行限流控制的 `ServiceThrottlingBehavior`, 用于进行服务授权的 `ServiceAuthorizationBehavior` 等, 可以通过应用特性或配置的方式应用于某个 WCF 服务。

对于 `ServiceDescription` 来说, 最重要的属 `ServiceEndpointCollection` 类型的 `Endpoints` 属性。该属性表示为服务添加的所有在终结点集合, 集合的每个元素为 `System.ServiceModel.Description.ServiceEndpoint` 对象, 接下来我们就来着重讨论 `ServiceEndpoint`。

### 7.1.2 ServiceEndpoint 与 EndpointBehavior

`ServiceEndpoint` 对象是对终结点的运行时描述, 终结点的三要素 (ABC: Address、Binding、Contract) 分别由同名的属性表示。`ListenUri` 和 `ListenUriMode` 表示终结点真正的监听地址和监听模式, `Address` 和 `ListenUri` 又被称为逻辑地址和物理地址 (关于逻辑地址和物理地址, 本书的第2章有详细的介绍)。

```

1 public class ServiceEndpoint
2 {
3     //其他成员
4     public EndpointAddress Address { get; set; }
5     public KeyedByTypeCollection<IEndpointBehavior> Behaviors { get; }
6     public Binding Binding { get; set; }
7     public ContractDescription Contract { get; }
8     public Uri ListenUri { get; set; }

```

```

9      public ListenUriMode ListenUriMode { get; set; }
10     public string Name { get; set; }
11 }

```

在 `ServiceEndpoint` 中，类型为 `KeyedByTypeCollection<IEndpointBehavior>` 的 `Behaviors` 属性表示绑定到该终结点的终结点行为（Endpoint Behavior）集合。集合的成员为实现了 `IEndpointBehavior` 接口的终结点行为对象。`IEndpointBehavior` 的定义如下，`AddBindingParameters`、`ApplyDispatchBehavior` 和 `Validate` 与 `IServiceBehavior` 同名方法语义类似。不同的是，`IEndpointBehavior` 所有方法的作用域仅限于当前终结点，并且 `IEndpointBehavior` 既可以作用于服务端，也可以用于客户端。为此，增加了一个新的方法：`ApplyClientBehavior`。`ApplyClientBehavior` 方法与 `ApplyDispatchBehavior` 相对，通过修改客户端运行时（Client Runtime）的属性，或者添加/替换客户端运行时某些可扩展对象，进而实现控制客户端行为的目的。

```

12 public interface IEndpointBehavior
13 {
14     void AddBindingParameters(ServiceEndpoint endpoint,
15         BindingParameterCollection bindingParameters);
16     void ApplyClientBehavior(ServiceEndpoint
17         endpoint, ClientRuntime
18         clientRuntime);
19     void ApplyDispatchBehavior(ServiceEndpoint
20         endpoint, EndpointDispatcher
21         endpointDispatcher);
22     void Validate(ServiceEndpoint endpoint);
23 }

```

终结点的契约通过 `ContractDescription` 对象表示，在第5章介绍服务契约的时候，已经对 `ContractDescription` 进行了简单的介绍，为了让大家对服务描述有一个系统的认识，接下来继续介绍 `ContractDescription`。

### 7.1.3 ContractDescription 和 ContractBehavior

`ContractDescription` 定义了以下一些属性用于描述服务契约。由于服务契约通过 `ServiceContractAttribute` 特性定义，所以大部分的属性都和 `ServiceContractAttribute` 的属性相匹配。由于在第5章介绍服务契约的时候，已经对大部分成员进行了介绍，在这里就不再作重复的介绍了。

```

1  public class ContractDescription
2  {
3      //其他成员
4      public KeyedByTypeCollection<IContractBehavior> Behaviors { get; }
5      public OperationDescriptionCollection

```

```

Operations { get; }
6 }

```

在 ContractDescription 中，类型为 KeyedByTypeCollection<IContractBehavior> 的属性 Behaviors 代表基于服务契约的契约行为（Contract Behavior）集合，集合成员为实现了接口 IContractBehavior 的契约行为对象。IContractBehavior 具有与 IEndpointBehavior 一样的方法成员，但是契约行为作用于实现了该服务契约的所有服务（服务端行为），基于该服务契约进行服务调用的服务代理（客户端行为）。

```

7 public interface IContractBehavior
8 {
9     void AddBindingParameters(ContractDescription contractDescription,
10         ServiceEndpoint endpoint, BindingParameterCollection
11         bindingParameters);
12     void ApplyClientBehavior(ContractDescription contractDescription,
13         ServiceEndpoint endpoint, ClientRuntime clientRuntime);
14     void ApplyDispatchBehavior(ContractDescription contractDescription,
15         ServiceEndpoint endpoint, DispatchRuntime dispatchRuntime);
16
17     void Validate(ContractDescription contractDescription,
18         ServiceEndpoint endpoint);
19 }

```

ContractDescription 的 Operations 属性表示服务契约所有操作的描述，类型为 OperationDescriptionCollection，表示一个 OperationDescription 对象的集合。接下来，我们来介绍 OperationDescription。

#### 7.1.4 OperationDescription 和 OperationBehavior

OperationDescription 定义了一系列的属性，用以描述定义在服务契约中的操作契约。由于操作契约通过 OperationContractAttribute 定义，所以 OperationDescription 的大部分属性与 OperationContractAttribute 属性一一匹配。同样地，在第5章介绍服务契约的时候已经对大部分成员进行了介绍，在这里就不再重复介绍。

```

1 public class OperationDescription
2 {
3     //其他成员
4     public KeyedByTypeCollection<IOperationBehavior> Behaviors { get; }
5 }

```

上面不止一次地提出客户端操作（Client Operation）和服务端操作（Dispatch Operation）的概念，这是由于在运行时，基于相同的 OperationDescription 创建的操作对



象在客户端和服务端是不同的，服务端操作称为分发操作（DispatchOperation），通过类型 System.ServiceModel.Dispatcher.DispatchOperation 表示，客户端操作通过类型 System.ServiceModel.Dispatcher.ClientOperation 表示。

在 OperationDescription 中，类型为 KeyedByTypeCollection<IOperationBehavior> 的 Behaviors 属性表示基于操作的所有操作行为（Operation Behavior）集合，集合成员为实现了 IOperationBehavior 接口的类型对象。IOperationBehavior 具有与 IEndpointBehavior、IContractBehavior 一样的方法成员。IOperationBehavior 的作用域仅限于当前的操作（客户端操作或服务端操作）。

```
6 public interface IOperationBehavior
7 {
8     void AddBindingParameters(OperationDescription
operationDescription,
9         BindingParameterCollection bindingParameters);
10    void ApplyClientBehavior(OperationDescription
operationDescription,
11        ClientOperation clientOperation);
12    void ApplyDispatchBehavior(OperationDescription
operationDescription,
13        DispatchOperation dispatchOperation);
14    void Validate(OperationDescription operationDescription);
15 }
```

## 7.2 服务寄宿详解

虽然直到本章我们才开始系统地介绍服务的寄宿，但是在前面的章节中，无论是服务的自我寄宿，还是基于 IIS 的寄宿，我们都进行过很多的演示，相信读者对服务的寄宿不会感到陌生。在 WCF 中，对服务进行寄宿的唯一途径就是为服务类型创建并开启 ServiceHostBase 对象。由于 WCF 已经为我们定义了一个功能强大的自定义 ServiceHostBase 对象：ServiceHost，所以在绝大部分情况下，我们都是直接使用 ServiceHost 进行服务的寄宿。

### 7.2.1 创建 ServiceHost

进行服务寄宿的第一个步骤就是为服务类型创建一个 ServiceHost 对象。ServiceHost 定义了如下两个构造函数，其中第1个构造函数只能用于 InstanceContextMode.Single 的服务，传入的对象为单例服务对象。我们现在主要通过第2个构造函数来介绍，当基于某个服务的 ServiceHost 被创建时，WCF 服务端框架为我们做了什么。

```
1 public class ServiceHost : ServiceHostBase
2 {
3     //其他成员
4     public ServiceHost(object singletonInstance,
```

```
params Uri[] baseAddresses);  
5     public ServiceHost(Type serviceType,  
params Uri[] baseAddresses);  
6 }
```

调用构造函数创建 `ServiceHost` 的最主要任务就是根据服务类型创建 `ServiceDescription` 对象，这个被创建的 `ServiceDescription` 对象最终被赋值给 `ServiceHost` 的 `Description` 属性。我们现在就来介绍一下这个 `ServiceDescription` 是如何一步一步被创建的：

将传入的 `serviceType` 作为 `ServiceDescription` 的 `ServiceType` 属性。通过反射得到应用在服务类型的 `ServiceBehaviorAttribute` 特性，如果该特性存在并且为服务设置了 `Name`、`Namespace` 和 `ConfigurationName` 属性，则用它们覆盖 `ServiceDescription` 同名属性的默认值；

通过反射得到通过自定义特性方式和配置方式应用到服务类型上的服务行为。这些服务行为和通过反射得到的 `ServiceBehaviorAttribute` 特性一起组成了 `ServiceDescription` 的服务行为集合；

通过解析服务的配置，得到为服务配置的所有终结点和相应的终结点行为，并创建相应的 `ServiceEndpoint` 对象。这些创建的 `ServiceEndpoint` 对象最终被添加到 `ServiceDescription` 的 `Endpoints` 集合属性中。对于自我寄宿，在成功创建了 `ServiceHost` 后，可以通过 `AddServiceEndpoint` 方法添加额外的终结点，这些终结点也会被添加到的 `Endpoints` 集合属性中；

对于每个终结点的 `Contract`，通过反射应用在 service 契约类型上的 `ServiceContractAttribute` 特性创建相应的 `ContractDescription`。然后通过反射获得以自定义特性方式应用在 service 契约上的契约行为，并将它们纳入契约的 `Behaviors` 集合属性；

对于 `ContractDescription` 中表示操作的每个 `OperationDescription`，通过反射应用在操作方法的 `OperationContractAttribute` 特性创建 `OperationDescription`，然后通过反射的机制获得以自定义特性的方式应用在方法上的操作行为。

### 7.2.2 开启 `ServiceHost` (1)

最简单的服务寄宿工作无外乎包含下面一些基本的操作：创建基于服务类型的 `ServiceHost` 对象；通过 `AddServiceEndpoint` 方法为服务添加一个或多个终结点；调用 `Open` 方法开启 `ServiceHost` 对象。创建 `ServiceHost` 对象的首要任务是创建用于详细描述当前服务的 `ServiceDescription` 对象，而通过调用 `Open` 方法开启 `ServiceHost` 的目的就是借助于创建的服务描述创建和初始化服务的运行时（Runtime）。由于整个过程比较复杂，接下来按照先后顺序介绍整个运行时的初始化过程。

### 验证服务描述

为了确保后续工作的正确执行，首要的步骤就是验证创建出来的 `ServiceDescription` 的正确性。而真正的验证规则定义在各种行为中，这样的行为包括：服务行为、终结点行为、契约行为和操作行为。当 `ServiceHost` 的 `Open` 方法被执行的时候，WCF 会通过 `ServiceHost` 的 `Description` 属性得到当前的服务描述，加载所有的行为对象，通过调用行为对象 `Validate` 方法在各自的作用域范围内验证服务描述的有效性。具体的步骤如下：

通过 `ServiceDescription` 的 `Behaviors` 属性得到所有的服务行为集合，遍历该集合并调用 `Validate` 方法；

通过 `ServiceDescription` 的 `Endpoints` 属性获得服务所有的终结点集合，遍历所有的终结点并获取相应的终结点行为，调用终结点行为的 `Validate` 方法；

对于每个终结点，通过 `Contract` 属性获取终结点的契约描述，遍历契约对应的所有契约行为，调用它们的 `Validate` 方法；

对于每个服务契约，通过 `Operations` 获取所有的操作描述，遍历每个操作并获取对应的所有操作行为，调用它们的 `Validate` 方法。

按照 `ListenUri` 和 `ListenUriMode` 分组所有终结点

对于每一个用于描述终结点的 `EndpointDescription` 对象来说，除了具有一个类型 `EndpointAddress` 的 `Address` 属性用以表示终结点的逻辑地址外，还具有一个类型 `Uri` 的 `ListenUri` 属性用以表示终结点的监听地址或物理地址。不同的终结点可以共享相同的 `ListenUri`。而 `ListenUriMode` 枚举用以表示是否须要确保 `ListenUri` 的唯一性（关于 `ListenUri` 和 `ListenUriMode`，请参阅本书第2章）

WCF 开启 `ServiceHost` 的第2个主要步骤就是将服务所有的终结点按照 `ListenUri` 和 `ListenUriMode` 进行重新分组。比如，`CalculatorService` 具有3个终结点，表7-1列出了3个终结点各自的 `Address`、`ListenUri` 和 `ListenUriMode`，后面是相应的配置。

表7-1

	Address	ListenUri	ListenUriMode
Endpoint1	http://127.0.0.1:7777/CalculatorService	http://127.0.0.1:6666/CalculatorService	Explicit
Endpoint2	http://127.0.0.1:8888/CalculatorService	http://127.0.0.1:6666/CalculatorService	Explicit

	e	e	
Endpoint3	http://127.0.0.1:9999/CalculatorService	http://127.0.0.1:6666/CalculatorService	Unique

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <configuration>
3      <system.serviceModel>
4          <services>
5              <service name="Artech.ServiceDescriptionDemos.
6                  CalculatorService">
7                  <endpoint name="endpoint1"
8                      address="http://
127.0.0.1:7777/CalculatorService"
9                      binding="wsHttpBinding"
10                     contract="Artech.
ServiceDescriptionDemos.ICalculator"
11                     listenUri="http://
127.0.0.1:6666/CalculatorService"
12                     listenUriMode="Explicit"/>
13                  <endpoint name="endpoint2"
14                      address=http://
127.0.0.1:8888/CalculatorService
15                      binding="wsHttpBinding"
16                      contract="Artech.
ServiceDescriptionDemos.ICalculator"
17                      listenUri=http://
127.0.0.1:6666/CalculatorService
18                      listenUriMode="Explicit"/>
19                  <endpoint name="endpoint3"
20                      address=http://
127.0.0.1:9999/CalculatorService
21                      binding="wsHttpBinding"
22                      contract="Artech.
ServiceDescriptionDemos.ICalculator"
23                      listenUri=http://
127.0.0.1:6666/CalculatorService
24                      listenUriMode="Unique"/>
25              </service>
26          </services>
27      </system.serviceModel>
28  </configuration>

```

## 7.2.2 开启 ServiceHost (2)

CalculatorService 的 3 个终结点共享相同的监听地址 (<http://127.0.0.1:6666/CalculatorService>),但是具有不同的监听地址模式 (Endpoint1 和 Endpoint2为 Explicit, Endpoint3为 Unique)。按照监听地址和监听模式的不同组合, 3 个终结点将分为如下两组, 见表7-2。

表7-2

第 一 组	<div>Endpoint1</div> <div>Address: http://127.0.0.1:7777/CalculatorService</div> <div>ListenUri : http://127.0.0.1:6666/CalculatorService</div> <div>ListenUriMode: Explicit</div> <div>Endpoint2</div> <div>Address: http://127.0.0.1:8888/CalculatorService</div> <div>ListenUri : http://127.0.0.1:6666/CalculatorService</div> <div>ListenUriMode: Explicit</div>
第 二 组	<div>Endpoint3</div> <div>Address: http://127.0.0.1:9999/CalculatorService</div> <div>ListenUri : http://127.0.0.1:6666/CalculatorService</div> <div>ListenUriMode: Unique</div>

添加绑定参数

WCF 的4种典型的行为（服务行为、终结点行为、契约行为和操作行为）都定义了一个 AddBindingParameters 方法,用以向某个自定义的绑定元素添加相应的绑定参数,以指导或保证该绑定元素正常执行。比如,通过 AddBindingParameters 添加必要的信息指导绑定元素创建和初始化相应的信道。AddBindingParameters 方法的执行方式和 Validate 方法类似,具体的步骤如下:

通过 ServiceDescription 的 Behaviors 属性获取服务所有的服务行为集合,遍历集合调用 AddBindingParameters 方法;

通过 ServiceDescription 的 Endpoints 属性获得服务所有的终结点描述,通过 EndpointDescription 的 Behaviors 得到所有终结点行为集合,遍历集合调用每个终结点行为的 AddBindingParameters 方法;

对于每个终结点描述,通过 Contract 得到服务对应的契约描述,通过

ContractDescription 的 Behaviors 属性获得所有契约行为集合，遍历集合调用每个契约行为的 AddBindingParameters 方法；

对于每个契约描述，通过 Operations 得到定义在服务契约中的所有操作描述，借助 OperationDescription 的 Behaviors 属性得到所有操作行为集合，遍历集合调用每个操作行为的 AddBindingParameters 方法。

创建信道分发器（ChannelDispatcher）和终结点分发器（EndpointDispatcher）

在 WCF 中，服务寄宿的本质就在于创建一个请求监听和操作执行的运行环境，而请求的监听和操作的执行最终落在两类重要的对象上面：信道分发器（ChannelDispatcher）和终结点分发器（EndpointDispatcher）。信道分发器负责进行请求的监听，终结点分发器负责操作的执行，信道分发器将请求消息分发给相应的终结点分发器。

信道分发器（ChannelDispatcher）

在 WCF 中，信道分发器通过类型 ChannelDispatcher 表示。ChannelDispatcher 继承自抽象基类 ChannelDispatcherBase，其定义如下。每个信道分发器通过各自的信道监听器进行请求的监听，信道监听器是一个实现了接口 IChannelListener 类型的对象，通过只读属性 Listener 可以获得这个对象。此外 ChannelDispatcherBase 的另一个只读属性 Host 便是当前的 ServiceHostBase 对象。

```
1 public abstract class ChannelDispatcherBase : CommunicationObject
2 {
3     // 其他成员
4     public abstract ServiceHostBase Host { get; }
5     public abstract IChannelListener Listener { get; }
6 }
```

在 ChannelDispatcher 中定义了一系列额外的属性和方法，其中最重要的就是 Endpoints 属性，类型为 SynchronizedCollection<EndpointDispatcher>，代表一个 EndpointDispatcher 的集合。由于 ChannelDispatcher 对应一个 IChannelListener，而一个 IChannelListener 绑定到一个确定的 URI，所以每一个监听地址都对应着一个 ChannelDispatcher。而多个终结点可以共享同一个监听地址，一个终结点和一个 EndpointDispatcher 对应，所以一个 ChannelDispatcher 对象对应着一个或多个 EndpointDispatcher 对象。

```
7 public class ChannelDispatcher : ChannelDispatcherBase
8 {
9     //其他成员
10    public SynchronizedCollection<EndpointDispatcher> Endpoints { get; }
11    public override ServiceHostBase Host { get; }
```

```

12     public override IChannelListener Listener { get; }
13 }

```

#### 终结点分发器 (EndpointDispatcher)

EndpointDispatcher 对应着服务的一个终结点，它从 ChannelDispatcher 接收请求消息，在进行必要的反序列化和其他额外的辅助操作后，执行相应的操作，并将结果序列化或成回复消息返回给 ChannelDispatcher。从下面 EndpointDispatcher 的定义来看其提供了一系列的属性和方法，我们着重介绍下面几个属性：

ChannelDispatcher: EndpointDispatcher 对应的 ChannelDispatcher 对象。

DispatchRuntime: 表示当前终结点分发器在分发运行时，通过该属性可以获取当前运行时状态，以及添加、修改或删除相应的运行时对象，从而实现对整个分发行为的定制。

### 7.2.2 开启 ServiceHost (3)

AddressFilter、ContractFilter 和 FilterPriority: 当 ChannelDispatcher 成功接收请求消息，并试图将其转发给相应的 EndpointDispatcher 的时候，通过这3个属性进行 EndpointDispatcher 的选择（关于基于 MessageFilter 和 FilterPriority 对 EndpointDispatcher 的选择，在第2章中有详细介绍）。

```

1  public class EndpointDispatcher
2  {
3      //其他成员
4      public ChannelDispatcher      ChannelDispatcher { get; }
5      public DispatchRuntime        DispatchRuntime { get; }
6
7      public MessageFilter
8      AddressFilter { get; set; }
9      public MessageFilter
10     ContractFilter { get; set; }
11     public int
12     FilterPriority { get; set; }
13 }

```

#### 如何创建 ChannelDispatcher 和 EndpointDispatcher

当基于 ListenUri 和 ListenUriMode 对所有的终结点进行分组时，由于每一组具有不同的 ListenUri 和 ListenUriMode，而 ListenUriMode 又决定了最终的监听地址（关于 ListenUriMode 采用怎样的机制决定最终的监听地址，可以参见本书第3章），所以每一组终结点实际上对应着一个监听地址。

WCF 会为每一组创建一个信道分发器。而信道分发器通过信道监听器进行请求的监听，所以需要为它创建信道监听器对象。WCF 通过终结点绑定对象的 BuildChannelListener 方法创

建 `IChannelListener` 对象。最后，WCF 为所有与该 `ChannelDispatcher` 共享相同监听地址的终结点创建终结点分发器，并将其纳入该信道分发器的 `EndpointDispatcher` 列表之中。

以上面例子中给出的终结点配置为例，我们可以通过下面的代码查看当 `ServiceHost` 被成功开启之后，会有怎样的信道分发器和终结点分发器被创建出来，它们之间都会有怎样的关系。

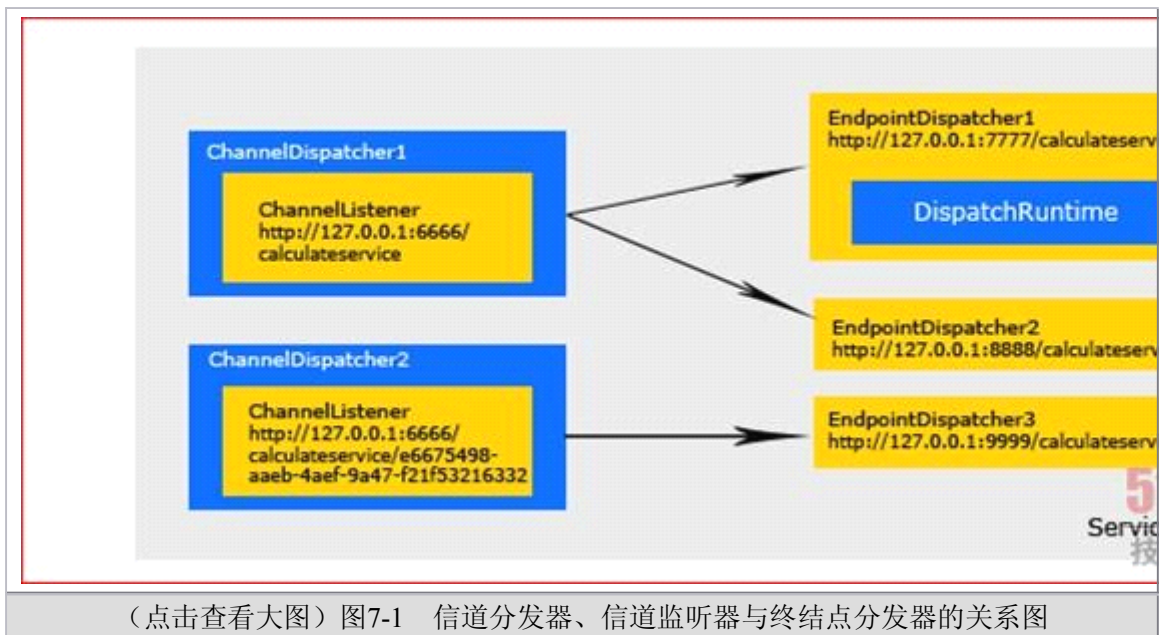
```
14 using (ServiceHost host = new ServiceHost
    (typeof(CalculatorService)))
15 {
16     host.Open();
17     for (int i = 0; i < host.ChannelDispatchers.Count; i++)
18     {
19         ChannelDispatcher dispatcher = host.
ChannelDispatchers[i] as ChannelDispatcher;
20         Console.WriteLine("ChannelDispatcher {0}
:{1}", i + 1, dispatcher.Listener.Uri);
21         for (int j = 0; j < dispatcher.Endpoints.Count; j++)
22         {
23             Console.WriteLine("\tEndpointDispatcher {0} :{1}", j + 1,
dispatcher.Endpoints[j].EndpointAddress.Uri);
24         }
25     }
26 }
27 }
```

输出结果：

```
28 ChannelDispatcher 1: http://127.0.0.1:6666/CalculatorService
29     EndpointDispatcher 1: http://127.0.0.1:7777/CalculatorService
30     EndpointDispatcher 2: http://127.0.0.1:8888/CalculatorService
31     ChannelDispatcher 2: http://127.0.0.1:6666/CalculatorService/e6675498-
32     aaeb-4aef-9a47- f21f53216332
```

输出结果反映了 `ServiceHost`、`ChannelDispatcher` 和 `EndpointDispatcher` 之间的关系，这种关系可以通过图7-1表示。





## 应用分发行为

当服务所有的 ChannelDispatcher 和 EndpointDispatcher 被成功创建时，该服务的整个监听与执行的环境已经基本搭建起来。接下来须要做的就是将分发行为（Dispatching Behavior）应用到服务的运行环境中。这个步骤和验证服务描述、添加绑定参数类似：获取当前服务所有的行为，包括服务行为、终结点行为、契约行为和操作行为，调用 ApplyDispatchBehavior 方法，具体的步骤如下：

通过 ServiceDescription 的 Behaviors 属性获取服务所有的服务行为集合，遍历集合调用 ApplyDispatchBehavior 方法；

通过 ServiceDescription 的 Endpoints 属性获得服务所有的终结点描述，通过 EndpointDescription 的 Behaviors 得到所有终结点行为集合，遍历集合调用每一个终结点行为的 ApplyDispatchBehavior 方法；

对于上面的每个终结点描述，通过 Contract 得到服务对应的契约描述，通过 ContractDescription 的 Behaviors 属性获得所有契约行为集合，遍历集合调用每一个契约行为的 ApplyDispatchBehavior 方法；

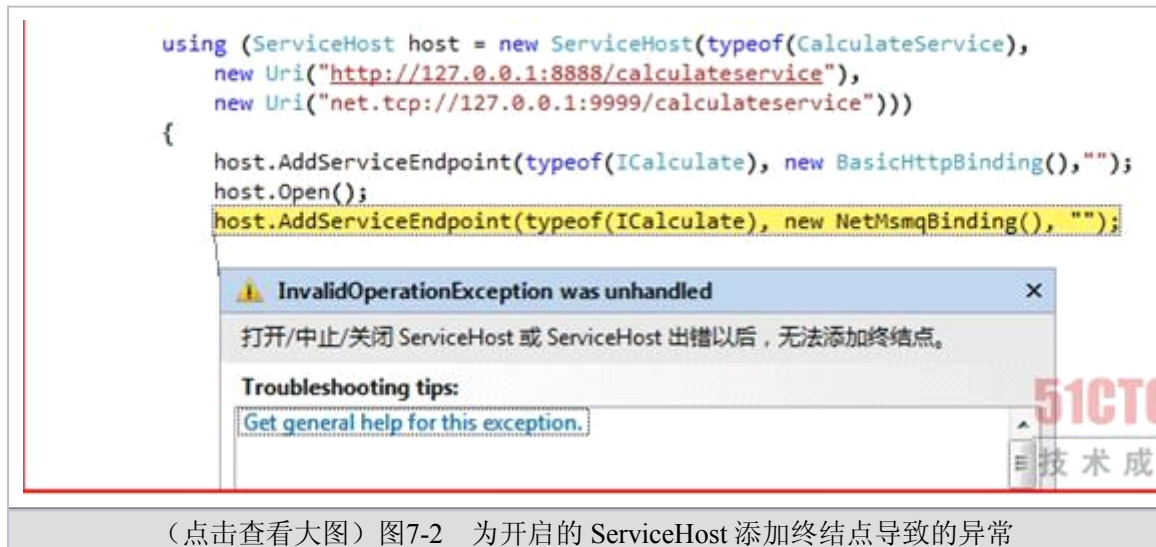
对于上面的契约描述，通过 Operations 得到定义在服务契约中的所有操作描述，借助 OperationDescription 的 Behaviors 属性得到所有操作行为集合，遍历集合调用每一个操作行为的 ApplyDispatchBehavior 方法。

## 7.2.2 开启 ServiceHost (4)

### 补充说明

服务的描述随着 ServiceHost 的实例化而被创建，而在调用 ServiceHostBase 的 Open 方法后，

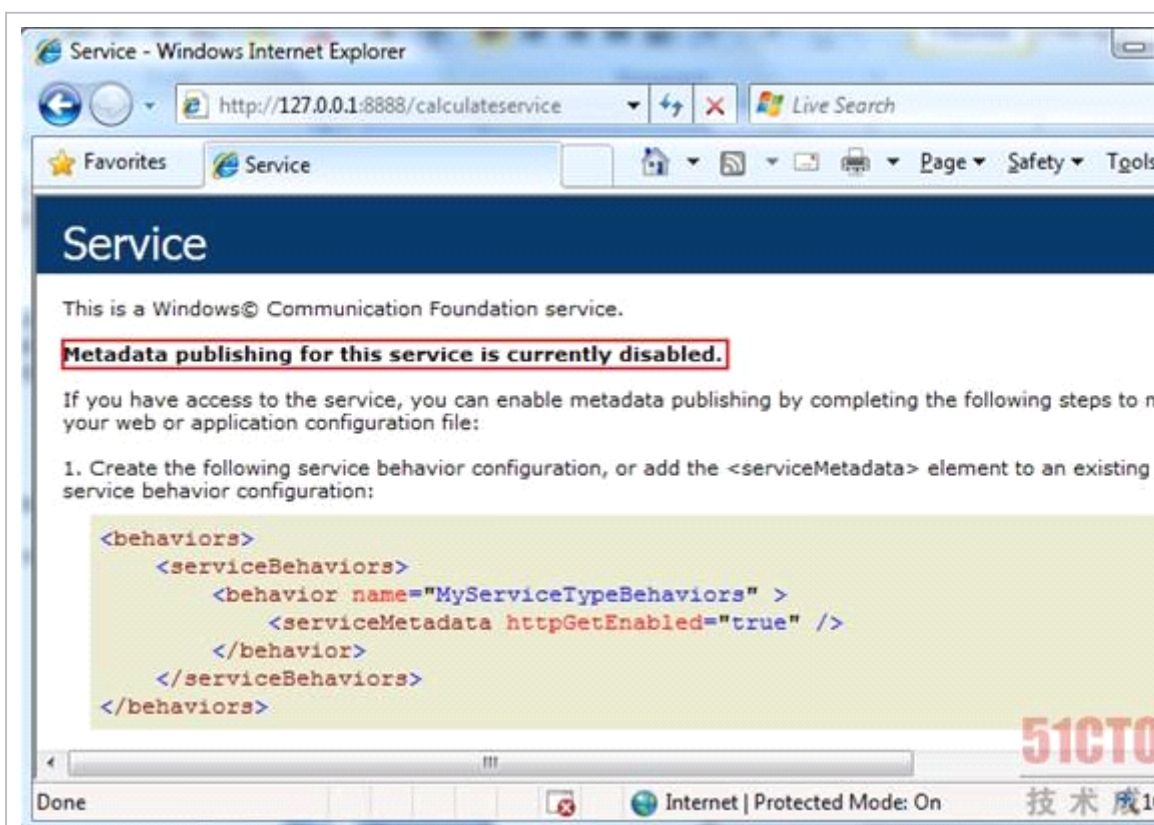
WCF 会根据服务描述创建出服务监听和执行的环境。所以我们通过编程的方式为服务添加终结点只有在 ServiceHostBase 被开启之前才有意义，而实际上，AddServiceEndpoint 方法只有在 ServiceHostBase 的 State 为 Created 时才是有效的，如果 AddServiceEndpoint 方法放在 Open 方法执行之后，会抛出如图7-2所示的 InvalidOperationException 异常。



（点击查看大图）图7-2 为开启的 ServiceHost 添加终结点导致的异常

同理，通过 ServiceHostBase 的 Description 属性对服务描述的修改，也只有在 Open 方法执行之前才有意义。比如，在下面的代码中，在成功执行 Open 方法后，为服务添加一个 ServiceMetadataBehavior 服务行为。但是，当我们通过 IE 访问该服务时（见图7-3），却提示元数据发布禁用（Metadata publishing for this service is currently disabled）。

```
1 using (ServiceHost host = new ServiceHost
2   (typeof(CalculatorService),
3     new Uri("http://127.0.0.1:8888/CalculatorService")))
4 {
5     host.AddServiceEndpoint(typeof(ICalculator),
6       new BasicHttpBinding(), "");
7     host.Open();
8
9     ServiceMetadataBehavior behavior = host.Description.Behaviors.
10       Find<ServiceMetadataBehavior>();
11     if (behavior == null)
12     {
13         behavior = new ServiceMetadataBehavior();
14         host.Description.Behaviors.Add(behavior);
15     }
16     behavior.HttpGetEnabled = true;
17     behavior.HttpGetUrl = new Uri("http://127.0.0.1:8888/
18       CalculatorService/mex");
19     Console.Read();
20 }
```



(点击查看大图) 图7-3 为开启的 ServiceHost 应用 ServiceMetadataBehavior 行为

### 7.3 WCF 服务的自我寄宿 (Self-Hosting)

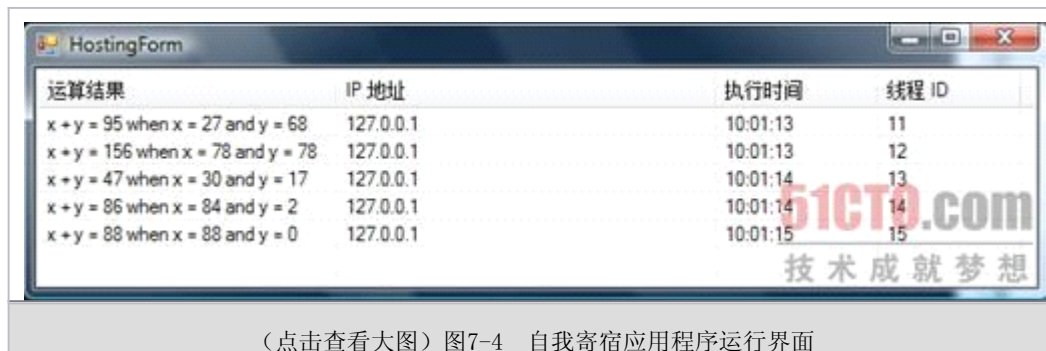
服务寄宿的本质在于为服务创建或指定一个进程,并为之创建一个监听和执行的运行环境。我们可以自行为服务建立一个托管应用程序,应用的类型可以是 Console 应用、Windows Form 应用, WPF 应用和 ASP.NET 应用等,也可以借助于 Windows 提供的原生进程激活方式,比如 IIS、Windows Service 和 WAS (Windows Process Activation Service) 等。本节着重介绍第1种方式,下一节会对后一种寄宿方式进行详细介绍。

WCF 服务的自我寄宿意味着我们需要以手工的方式控制服务寄宿涉及的所有方面,手工开启和关闭寄宿进程,以手工的方式控制服务的生命周期(服务的生命周期可以理解为 ServiceHostBase 的生命周期)。通过自我寄宿,我们还可以按照需求自由地设计管理界面。在前面的章节中,多次演示了基于 Console 应用的寄宿方式,在这里我们通过一个简单的案例演示如何通过 Windows 应用来进行服务的寄宿。

#### 7.3.1 案例演示: 如何通过 Windows 应用进行服务寄宿 (1)

接下来,我们将通过一个案例演示如何通过一个 Windows Form 应用来寄宿 WCF 服务。在这个案例中,还会利用一个 Windows Form 创建一个简易的控制界面,在这个界面中会实

时显示出每个服务操作执行的结果、开始执行时间、客户的 IP 地址和操作执行的线程 ID, 图7-4为寄宿 WCF 服务的 Windows Form 应用的运行界面, 其中的表格是一个 ListView 控件。



本案例不仅会演示基于 Windows Form 应用进行服务寄宿的整个流程, 还会介绍案例涉及的一些额外的知识。比如, 如何解决采用 GUI 应用寄宿服务引发的并发问题; 如何获取访问者的 IP 地址等。

本案例依然采用一贯的4层结构, Contract 和 Services 为 Class Library 项目, 用于定义服务契约和实现契约的服务; Windows 应用 Hosting 用于寄宿服务; Console 应用 Client 模拟客户端。

#### 步骤一 定义服务契约: ICalculator

我们依然沿用熟悉的计算服务的例子, 服务契约 ICalculator 接口定义如下。

```
1 using System.ServiceModel;
2 namespace Artech.WindowsFormHostingDemo.Contracts
3 {
4     [ServiceContract(Namespace="http://www.artech.com/")]
5     public interface ICalculator
6     {
7         [OperationContract]
8         double Add(double x, double y);
9     }
10 }
```

#### 步骤二 实现服务: CalculatorService

下面是整个服务的实现代码, 其中静态属性 DisplayPanel 表示服务寄宿应用界面上的 ListView 控件, 用于实时地将当前执行信息传输出来。由于在 GUI 程序中, 对控件的操作只能执行于创建控件的线程中, 所以我们借助于 SynchronizationContext 对象得到 UI 线程的同步上下文, 实现在非 UI 线程中向 ListView 添加相应条目。

在操作方法 Add 中, 我们需要实时输出运算结果、执行时间、线程 ID, 以及客户端 IP

地址。在服务端，可以从请求消息的消息属性集合提取服务访问者的 IP 地址。每个消息都有一个消息属性的集合，该集合可以被看作一个字典集合，消息属性以键-值对的形式作为该集合的元素。我们可以动态地为某一个消息添加消息属性，并可以根据名称从消息中提取消息属性的值。在 WCF 中，最为实用的就是通过消息属性的方式添加或提取 HTTP 请求或回复的报头。

客户端在进行服务调用的时候，一般地，自身的 IP 地址会以消息属性的形式添加到请求消息中。这个特殊的消息属性被称为“远程终结点消息属性（RemoteEndpointMessageProperty）”，它在消息属性集合的 Key 可以通过静态只读属性 RemoteEndpointMessageProperty.Name（其值实际上是一个字符串：System.ServiceModel.Channels.RemoteEndpointMessageProperty）获得。RemoteEndpointMessageProperty 的 Address 属性即为 IP 地址。将所有这些信息封装成 ListViewItem 对象，添加到 ListView 的显示列表中。为了有效地模拟并发，特意在 Add 方法返回前休眠5秒。

```
11 using System;
12 using Artech.WindowsFormHostingDemo.Contracts;
13 using System.Threading;
14 using System.Windows.Forms;
15 using System.ServiceModel;
16 using System.ServiceModel.Channels;
17 namespace Artech.WindowsFormHostingDemo.Services
18 {
19     public class CalculatorService : ICalculator
20     {
21         public static ListView DisplayPanel
22         { get; set; }
23
24         public static SynchronizationContext SynchronizationContext
25         { get; set; }
26
27         public double Add(double x, double y)
28         {
29             DateTime calculationTime = DateTime.Now;
30             string threadID = Thread.CurrentThread.
ManagedThreadId.ToString();
31             double result = x + y;
32             object remoteProperty;
33             OperationContext.Current.Incoming
MessageProperties.TryGetValue
34             (RemoteEndpointMessageProperty.Name,
out remoteProperty);
```

```

35         string ipAddress = (remoteProperty as
36             RemoteEndpointMessageProperty).Address;
37         SynchronizationContext.Post(delegate
38             {
39             ListViewItem item = new ListViewItem(
40                 new string[] {string.Format
41                     ("x + y = {2} when x = {0} and y =
42                     {1}", x, y, result),
43                     ipAddress,
44                     calculationTime.ToString("hh:MM:ss"),
45                     threadID});
46
47             DispalyPanel.Items.Add(item);
48         }, null);
49         Thread.Sleep(5000);
50         return x + y;
51     }
52 }

```

### 7.3.1 案例演示：如何通过 Windows 应用进行服务寄宿（2）

#### 步骤三 创建寄宿应用：Hosting

Hosting 是一个简单的 Windows 应用，这个应用仅仅包含上一个 Windows 窗体。如图7-4所示，该 Windows 窗体的所有内容仅仅是一个以 Detail 模式显示的 ListView。服务寄宿放在 Form 的 Load 事件中，在 Closed 事件中将 ServiceHost 关闭。在 Load 事件中，除了进行服务寄宿外，还需要通过为 CalculatorService 初始化两个静态变量：DisplayPanel 和 SynchronizationContext，相关代码和配置如下所示：

```

1  using System;
2  using System.Windows.Forms;
3  using Artech.WindowsFormHostingDemo.Services;
4  using System.ServiceModel;
5  using System.Threading;
6  namespace Artech.WindowsFormHostingDemo.Hosting
7  {
8      public partial class HostingForm : Form
9      {
10         ServiceHost _serviceHost = null;
11         //其他成员
12         private void HostingForm_Load(object
13             sender, EventArgs e)
14         {
15             CalculatorService.DispalyPanel = this.listViewDispalyPanel;

```

```

15         CalculatorService.SynchronizationContext =
16             SynchronizationContext.Current;
17         this._serviceHost = new ServiceHost
18             (typeof(CalculatorService));
19         this._serviceHost.Open();
20     }
21     private void HostingForm_FormClosed(object
22     sender, FormClosedEventArgs e)
23     {
24         IDisposable disposable = this._serviceHost as IDisposable;
25         if (disposable != null)
26         {
27             disposable.Dispose();
28         }
29     }
30 }
31 <?xml version="1.0" encoding="utf-8" ?>
32 <configuration>
33     <system.serviceModel>
34         <services>
35             <service name="Artech.Windows
36                 FormHostingDemo.Services.
37                 CalculatorService">
38                 <endpoint address="net.tcp://127.0.0.1:9999/
39                     CalculatorService"
40                         binding="netTcpBinding" bindingConfiguration=""
41                         contract=
42                             "Artech.WindowsForm
43                             HostingDemo.Contracts.ICalculator" />
44             </service>
45         </services>
46     </system.serviceModel>
47 </configuration>

```

#### 步骤四 创建客户端: Client

Client 是一个简单的 Console 应用。为了模拟用客户端并发对同一个服务进行服务调用，我们将服务代理的创建、服务调用和回收放到线程池中，以确保在 for 循环中的5次服务访问是并发的，下面是所有代码和配置。

```

45 using System;
46 using Artech.WindowsFormHostingDemo.Contracts;
47 using System.ServiceModel;

```

```

48 using System.Threading;
49 namespace Artech.WindowsFormHostingDemo.Client
50 {
51     class Program
52     {
53         static void Main(string[] args)
54         {
55             Random random = new Random();
56             using (ChannelFactory<ICalculator> channelFactory = new
57                 ChannelFactory<ICalculator>("CalculatorService"))
58             {
59                 for (int i = 0; i < 5; i++)
60                 {
61                     ThreadPool.QueueUserWorkItem(
62                         delegate
63                         {
64                             ICalculator calculator =
65                                 channelFactory.CreateChannel();
66                             using (calculator as IDisposable)
67                             {
68                                 int op1 = (int)(100 *
random.NextDouble());
69                                 int op2 = (int)(100 *
random.NextDouble());
70                                 calculator.Add(op1, op2);
71                             }
72                         });
73                 }
74
75                 Console.Read();
76             }
77         }
78     }
79 }
80 <?xml version="1.0" encoding="utf-8" ?>
81 <configuration>
82     <system.serviceModel>
83         <client>
84             <endpoint address="net.tcp://
127.0.0.1:9999/CalculatorService"
85                 binding="netTcpBinding" bindingConfiguration=""
contract=
86                 "Artech.WindowsFormHostingDemo.
Contracts.ICalculator"

```

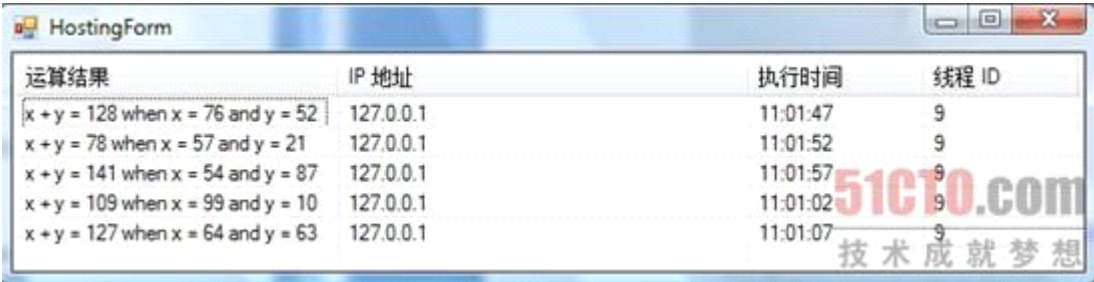


```

87         name="CalculatorService" />
88     </client>
89 </system.serviceModel>
90 </configuration>

```

通过以上4个步骤，整个应用创建完毕，图7-5为运行时的界面。从中我们会发现一个很严重的问题：虽然在客户端，5次服务访问是并发执行的，但是从显示出来的执行记录可以看出，服务操作的执行却是同步的，因为它们执行的时间依次相差5秒。操作的同步执行的特性从线程 ID 也可以看出：所有的线程 ID 都是9（该线程实际上是 UI 的主线程）。



运算结果	IP 地址	执行时间	线程 ID
x + y = 128 when x = 76 and y = 52	127.0.0.1	11:01:47	9
x + y = 78 when x = 57 and y = 21	127.0.0.1	11:01:52	9
x + y = 141 when x = 54 and y = 87	127.0.0.1	11:01:57	9
x + y = 109 when x = 99 and y = 10	127.0.0.1	11:01:02	9
x + y = 127 when x = 64 and y = 63	127.0.0.1	11:01:07	9

（点击查看大图）图7-5 自我寄宿应用程序执行结果（A）

这样的问题在真正的项目应用中是绝对不能被接受的，读者可以想像：在这个例子中，有5个并发的服务访问，那么有一个将至少等待25秒，如果10个并发访问就是50秒。倘若客户端采用的默认超时时限为1分钟，那么该应用最理想的状态只能处理12个并发访问。

#### 步骤五 解决并发服务执行的问题

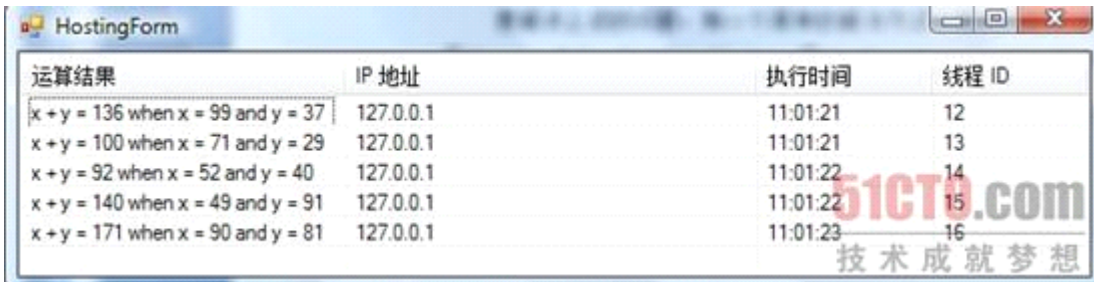
要解决上述问题，有一个简单的解决方式，我们只须在服务 CalculatorService 上面应用 ServiceBehaviorAttribute，将 UseSynchronizationContext 属性设为 false 就可以了。

```

91 [ServiceBehavior(UseSynchronizationContext = false)]
92 public class CalculatorService : ICalculator
93 {
94     //省略实现
95 }

```

再次运行我们的应用程序，你将会得到如下的输出。无论从执行时间还是执行的线程 ID，都可以确定操作现在已经变成并发执行的了。在下一卷关于并发的章节中，会对 [ServiceBehavior(UseSynchronizationContext = false)] 背后的原理进行详细介绍。



运算结果	IP 地址	执行时间	线程 ID
x + y = 136 when x = 99 and y = 37	127.0.0.1	11:01:21	12
x + y = 100 when x = 71 and y = 29	127.0.0.1	11:01:21	13
x + y = 92 when x = 52 and y = 40	127.0.0.1	11:01:22	14
x + y = 140 when x = 49 and y = 91	127.0.0.1	11:01:22	15
x + y = 171 when x = 90 and y = 81	127.0.0.1	11:01:23	16

## 7.4 通过 IIS 进行服务寄宿

WCF 的服务寄宿, 究其本质, 就是创建或激活一个进程, 为 WCF 服务构建一个运行环境。IIS 本身可以看作一个自动化的进程激活工具。以 ASP.NET 应用为例: 我们通过 HTTP 协议请求寄宿于目的 Web 服务器 (IIS) 的某个 ASP.NET 资源, 比如 .aspx 页面, 或者 .asmx Web 服务等, IIS 会将该请求转发给相应的工作进程 (Worker Process), 如果该进程不存在, 则激活或创建一个新的工作进程来处理该请求。同理, 也可以采用相似的进程激活机制来处理基于 WCF 的服务请求。较之手工创建托管应用程序对 WCF 服务进行自我寄宿, 基于 IIS 的服务寄宿方式具有如下的优点。

自动化的进程激活与关闭 (Process Activation & Idle Shutdown): IIS 除了具有处理上述根据请求自动激活工作进程的能力之外, 还可以通过相应的配置使工作进程空闲到一定的时候自动将其关闭, 以节约服务器的资源。

自动化的进程回收 (Process Recycling): 可以通过相应的配置使 IIS 定期重启以解决一些周期性的已知或未知的问题, 比如内存泄露。

自动化的进程健康监测 (Process Health Monitoring): 可以通过 WWW 服务 (World Wide Web Publishing Service) 周期性地访问工作进程以检查其健康状态, 如果监测出工作进程出现问题, 可以关闭该进程并创建新的工作进程。

ASP.NET 共享寄宿模式 (ASP.NET Shared Hosting Mode): 可以充分利用类似于 ASP.NET Web Form 应用或 XML Web Service 的共享寄宿模式, 将多个应用寄宿于同一个工作进程, 以改善服务器密度和可扩展性 (Server Density and Scalability), 甚至可以将一个 ASP.NET Web Form 应用和 WCF 服务运行在一个相同的 AppDomain 中以获得最好的性能。

ASP.NET 动态编译 (Dynamic Compilation): 将 WCF 服务寄宿于 IIS 下可以利用 ASP.NET 应用程序的动态编译功能, 以方便开发。

借助于 IIS 寄宿 WCF 服务是最常见的服务寄宿方式, 能够适应真正的企业级服务部署的要求。基于 IIS 的 WCF 服务寄宿使用于各种典型的 Windows 平台和不同的 IIS 版本, 下面列出了比较典型的操作系统和 IIS 版本的组合:

Windows XP (SP2) + IIS 5.1

Windows 2003 + IIS 6.0

Windows Vista + IIS 7.0

Windows Server 2008 + IIS 7.0

在详细介绍基于 IIS 进行服务寄宿的原理之前，我们先通过一个简单的案例了解进行 IIS 服务寄宿的大体步骤。

#### 7.4.1 案例演示：如何通过 IIS 进行服务寄宿（1）

在本案例中，将以最简洁的代码演示基于 IIS 服务寄宿的必要步骤，图7-8演示的是整个解决方案的3层结构。



（点击查看大图）图7-8 基于 IIS 服务寄宿案例应用结构

Contracts: 定义服务契约，并被其他两个项目引用。

<http://localhost/IISHostingDemo/>: 一个简单的 IIS Web 站点，即是服务的寄宿站点，服务的实现类置于其中。

Client: 一个 Console 应用，模拟调用服务的客户端。

步骤一 定义服务契约: ICalculator

我们依然沿用运算服务的例子，所以将经常使用的 ICalculator 接口定义成服务契约。

```
1 using System.ServiceModel;
2 namespace Artech.IISHostingDemo.Contracts
3 {
4     [ServiceContract(Namespace = "http://www.artech.com/")]
5     public interface ICalculator
6     {
7         [OperationContract]
8         double Add(double x, double y);
9     }
10 }
```

步骤二 实现服务: CalculatorService

在解决方法目录下创建子目录（假设为 Services），将此目录映射为 IIS 的虚拟目录，

名称为 IISHostingDemo。然后通过 VS 以添加现有 Web 站点的形式开启该虚拟目录 (<http://localhost/IISHostingDemo>)。在此 Web 站点中添加新的类文件，定义如下服务 CalculatorService:

```
11 using Artech.IISHostingDemo.Contracts;
12 public class CalculatorService : ICalculator
13 {
14     public double Add(double x, double y)
15     {
16         return x + y;
17     }
18 }
```

如同 XML Web Service 需要 .asmx 文件一样, 寄宿于 IIS 的 WCF 服务需要一个 .svc 文件。虽然在 VS 的“添加新项目”对话框中有“WCF Service”选项, 但是当我们选择该选项的时候, 它除了创建 .svc 文件之外, 还会在同一个项目下“自作主张”地创建代表服务契约的接口和实现契约的服务类。由于 .svc 文件仅仅是一个包含“%ServiceHost%”指令 (Directive) 的文本文件, 我们可以添加一个文本文件, 并将扩展名改为 .svc。在这里我们在 Web 站点根目录下创建一个 CalculatorService.svc 的文件, 内容包含如下的“%ServiceHost%”指令 (Directive)。

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

“%ServiceHost%”指令 (Directive) 具有5个属性 (Attribute), 它们分别具有如下的含义:

Service: 实现了服务契约的 WCF 服务类型的有效名称 (Qualified Name), 比如 “Artech.WcfServices.Services.CalculatorService”, 或者 “Artech.WcfServices.Services.CalculatorService”, Artech.WcfServices.Services, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null”。

Debug: “true”或“false”, 表示 WCF 服务是否以 Debug 的模式进行编译。

CodeBehind: 源文件 (.cs 或者 .vb 文件) 的路径。

Factory: 由于 WCF 服务的寄宿工作最终都落在一个 ServiceHostBase 对象上, 我们可以使用系统定义的 System.ServiceModel.ServiceHost 类型, 也可以自定义继承 ServiceHostBase 的 ServiceHost。每一个 ServiceHost 类型都具有与之匹配的 ServiceHostFactory 类型, 用于创建相应的 ServiceHost 对象。所有的 ServiceHostFactory 都继承自 ServiceHostBaseFactory, System.ServiceModel.ServiceHost 类型对应的 ServiceHostFactory 为 System.ServiceModel.ServiceHostFactory。Factory 属性用于指定 ServiceHostFactory 类型的有效名称, 可以缺省, 默认为 “System.ServiceModel.

ServiceHostFactory”。

Language: 同.aspx 页面一样, 后台代码 (VB.NET 或者 C#) 既可以以 CodeBehind 的形式存储于单独的文件内, 也可以以内联 (Inline) 的方式和“%ServiceHost%”指令一起保存, 下面的示例代码演示的就是内联的.svc。Language 表示源代码对应的编程语言, 有效值为“VB”或“C#”。

```
19 <%@ ServiceHost Language="C#" Service="CalculatorService" %>
20 public class CalculatorService: ICalculator
21     {
22         #region ICalcualte Members
23
24         public double Add(double x, double y)
25         {
26             return x + y;
27         }
28
29         #endregion
30     }
```

#### 7.4.1 案例演示：如何通过 IIS 进行服务寄宿 (2)

完成了 WCF 服务和相应.svc 文件的定义之后, 需要在 web.config 中对服务进行相应的配置。基于 IIS 服务寄宿的配置与自我寄宿方式基本一致, 唯一不同的地方在于: 不必指定服务的地址, 因为.svc 文件的地址即为服务的地址。

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <configuration>
3      <system.serviceModel>
4          <services>
5              <service name="CalculatorService">
6                  <endpoint binding="wsHttpBinding"
contract="Artech.IISHostingDemo.
7                      Contracts.ICalculator"/>
8              </service>
9          </services>
10     </system.serviceModel>
11 </configuration>
```

#### 步骤三 创建客户端: Client

Client 是一个简单的 Console 应用, 下面是服务调用程序和配置, 其中终结点的地址为.svc 的网络路径。

```

12 using System;
13 using System.ServiceModel;
14 using Artech.IISHostingDemo.Contracts;
15 namespace Artech.IISHostingDemo.Client
16 {
17     class Program
18     {
19         static void Main(string[] args)
20         {
21             using (ChannelFactory<ICalculator> channelFactory = new
22                 ChannelFactory<ICalculator>("CalculatorService"))
23             {
24                 ICalculator calculator = channelFactory.CreateChannel();
25                 using (calculator as IDisposable)
26                 {
27                     Console.WriteLine("x + y = {2}
when x = {0} and y =
28                     {1}", 1, 2, calculator.Add(1, 2));
29                 }
30             }
31             Console.Read();
32         }
33     }
34 }
35 <?xml version="1.0" encoding="utf-8" ?>
36 <configuration>
37     <system.serviceModel>
38         <client>
39             <endpoint name="CalculatorService" address=
40                 "http://localhost/IISHostingDemo/
CalculatorService.svc"
41                 binding="wsHttpBinding" contract="
Artech.IISHostingDemo.
42                 Contracts.ICalculator"/>
43         </client>
44     </system.serviceModel>
45 </configuration>

```

#### 7.4.2 IIS 管道与 ASP.NET 架构 (1)

同 .asmx Web 服务一样，借助 IIS 对 WCF 进行寄宿，实际上还是利用 ASP.NET。如果读者希望对基于 IIS 服务寄宿的实现机制有一个深层次的理解，对于 IIS 和 ASP.NET 架构的理解是必须的。

说到 ASP.NET，我们大部分读者会很自然地想到基于 Web Form 的 Web 应用，或者是基

于.aspx的Web Service应用,但是ASP.NET远非如此。那么如何对ASP.NET下一个定义呢?我个人比较倾向于这样的说法:“ASP.NET是一个托管的Web请求处理引擎。”

从本质上讲,ASP.NET本身并不依赖于IIS或Web Server,它提供一个独立的处理Web请求的管道(Pipeline),我们完全可以将ASP.NET运行时(HttpRuntime)寄宿于一个Windows Form应用。但是,绝大部分基于ASP.NET的Web应用都部署于IIS下,IIS和ASP.NET协同工作处理Web请求。我们以一个ASP.NET Web Form应用为例,客户端通过浏览器对某一个.aspx页面发起请求,当该请求抵达目的服务器,IIS是第一道屏障。Web请求经过IIS管道的处理(比如认证),被转发给ASP.NET管道进行进一步的处理。

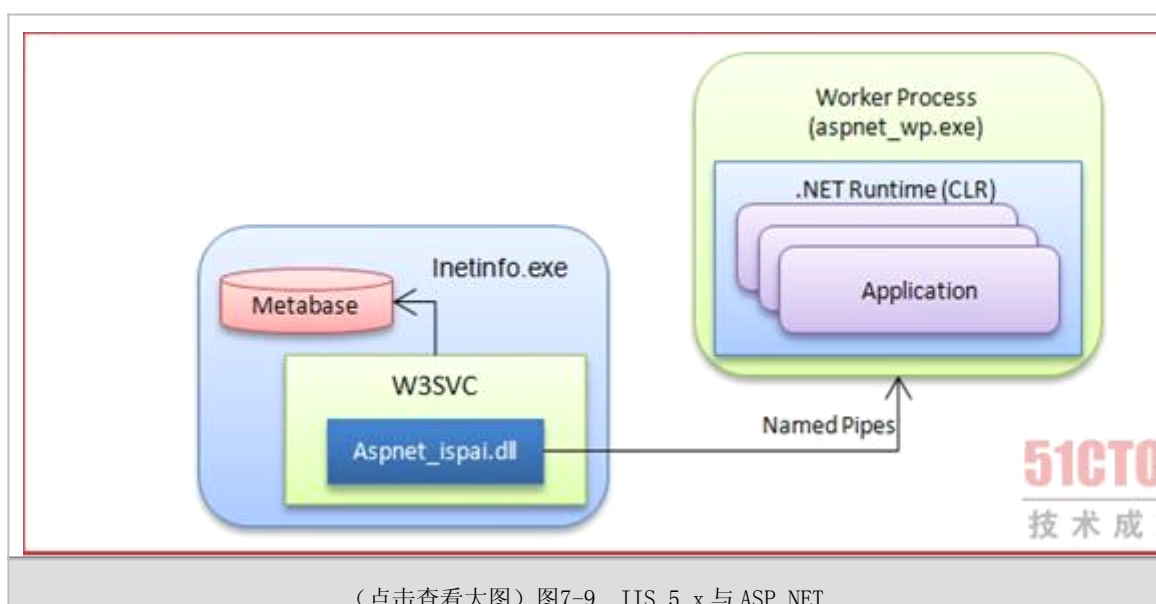
在本节中,我们将从一个较低的层次来剖析IIS管道和ASP.NET架构,以及Web请求从IIS到ASP.NET是如何过渡的。由于不同版本的IIS的处理方式具有很大的差异,接下来会介绍3个主要的IIS版本各自对Web请求的不同处理方式。

### IIS 5.x 与 ASP.NET

我们先来看看IIS 5.x是如何处理基于ASP.NET资源(比如.aspx、.asmx等)请求的,整个过程基本上可以通过图7-9体现。

IIS 5.x运行在进程InetInfo.exe中,在该进程中一个最重要的服务就是名为World Wide Web Publishing Service(简称W3SVC)的Windows Service。W3SVC的主要功能包括HTTP请求的监听、工作进程的管理及配置管理(通过从Metabase中加载相关配置信息)等。

当检测到某个HTTP Request时,先根据扩展名判断请求的是否是静态资源(比如.html、.img、.txt、.xml等),如果是则直接将文件内容以HTTP Response的形式返回。如果是动态资源(比如.aspx、.asp、.php等),则通过扩展名从IIS的脚本映射(Script Map)找到相应的ISAPI DLL。



ISAPI 是 Internet 服务器 API (Internet Server Application Programming Interface) 的缩写, 是一套本地的 (Native) Win32 API, 具有较高的执行性能, 是 IIS 和其他动态 Web 应用或平台之间的纽带。比如 ASP ISAPI 桥接 IIS 与 ASP, 而 ASP.NET ISAPI 则连接着 IIS 与 ASP.NET。ISAPI 定义在一个 Dll 中, ASP.NET ISAPI 对应的 Dll 为 `Aspnet_isapi.dll`, 你可以在目录 `%windir%\Microsoft.NET\Framework\{version no}\` 中找到该 Dll。

ISAPI 支持 ISAPI 扩展 (ISAPI Extension) 和 ISAPI 筛选 (ISAPI Filter), 前者是真正处理 HTTP 请求的接口, 后者则可以在 HTTP 请求真正被处理之前查看、修改、转发或拒绝请求, 比如 IIS 可以利用 ISAPI 筛选进行请求的验证 (Authentication)。

如果我们请求的是一个基于 ASP.NET 的资源类型, 比如: `.aspx` Web Page、`.asmx` Web Service 或 `.svc` WCF Service 等, `Aspnet_isapi.dll` 会被加载, ASP.NET ISAPI 扩展会创建 ASP.NET 的工作进程 (如果该进程尚未启动), 对于 IIS 5.x 来说, 该工作进程为 `aspnet.exe`。IIS 进程与工作进程之间通过命名管道 (Named Pipes) 进程通信, 以获得最好的性能。

在工作进程初始化过程中, .NET 运行时 (CLR) 被加载, 从而构建了一个托管的环境。对于某个 Web 应用的初次请求, CLR 会为其创建一个 AppDomain。在此 AppDomain 中, HTTP 运行时 (HTTP Runtime) 被加载并用以创建相应的应用。寄宿于 IIS 5.x 的所有 Web 应用都运行在同一个进程 (工作进程 `Aspnet_wp.exe`) 的不同 AppDomain 中。

## IIS 6.0与 ASP.NET

通过上面的介绍, 我们可以看出 IIS 5.x 至少存在着如下两个方面的不足:

ISAPI Dll 被加载到 `InetInfo.exe` 进程中, 它和工作进程之间是一种典型的跨进程通信方式, 尽管采用性能最好的命名管道, 但是仍然会带来性能的瓶颈。

所有的 ASP.NET 应用, 运行在相同进程 (`aspnet_wp.exe`) 中的不同的应用程序域 (AppDomain) 中, 基于应用程序域的隔离级别不能从根本上解决一个应用程序对另一个程序的影响, 在更多的时候, 我们需要不同的 Web 应用运行在不同的进程中。

在 IIS 6.0 中, 为了解决第一个问题, ISAPI.dll 被直接加载到工作进程中。为了解决第二个问题, 引入了应用程序池 (Application Pool) 的机制。我们可以为一个或多个 Web 应用创建应用程序池, 每一个应用程序池对应一个独立的工作进程, 从而为运行在不同应用程序池中的 Web 应用提供基于进程的隔离级别。IIS 6.0 的工作进程名称为 `w3wp.exe`。

当然, 除了上面两点改进之外, IIS 6.0 还有其他一些值得称道的地方, 其中最重要的一点就是创建了一个新的 HTTP 监听器: HTTP 协议栈 (HTTP Protocol Stack, HTTP.SYS)。HTTP.SYS 运行在 Windows 的内核模式 (Kernel Mode) 下, 作为驱动程序而存在。它是 Windows 2003 的 TCP/IP 网络子系统的一部分, 从结构上, 它属于 TCP 之上的一个网络驱动程序。严



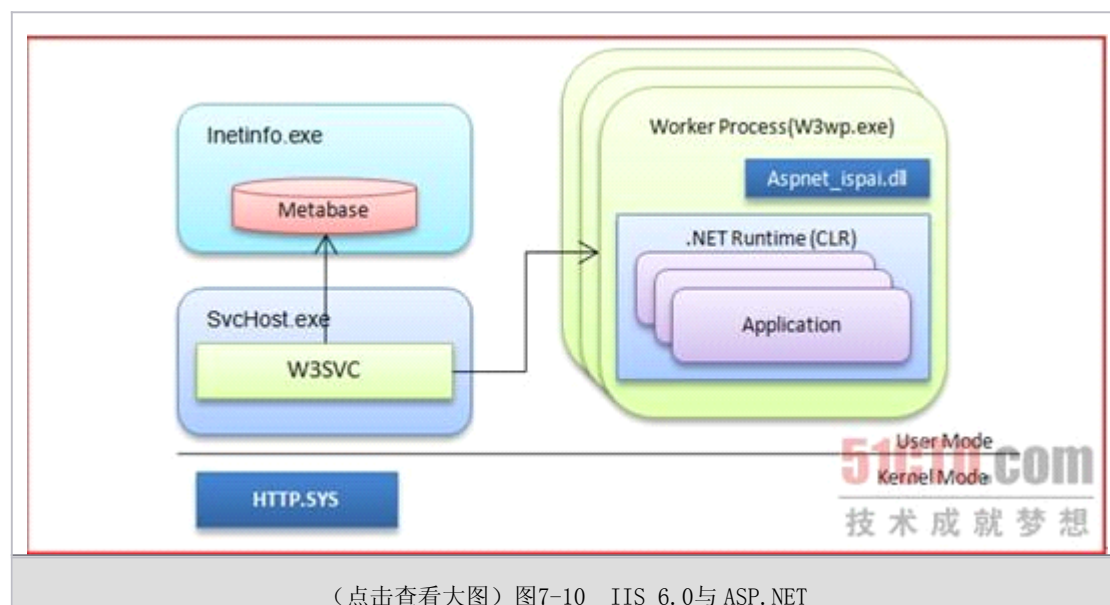
格地说, HTTP.SYS 已经不属于 IIS 的范畴了, 所以 HTTP.SYS 的配置信息并不保存在 IIS 的元数据库 (Metabase) 中, 而是定义在注册表中。HTTP.SYS 的注册表项位于下面的路径中: HKEY\_LOCAL\_MACHINE/SYSTEM/CurrentControlSet/Services/HTTP。HTTP.SYS 能够带来如下的好处。

**持续监听:** 由于 HTTP.SYS 是一个网络驱动程序, 始终处于运行状态, 对于用户的 HTTP 请求, 能够及时作出反应。

**更好的稳定性:** HTTP.SYS 运行在操作系统内核模式下, 并不执行任何用户代码, 所以其本身不会受到 Web 应用、工作进程和 IIS 进程的影响。

**内核模式下数据缓存:** 如果某个资源被频繁请求, HTTP.SYS 会把响应的内容进行缓存, 缓存的内容可以直接响应后续的请求。由于这是基于内核模式的缓存, 不存在内核模式和用户模式的切换, 响应速度将得到极大的改进。

图7-10体现了 IIS 的结构和处理 HTTP 请求的流程。从中可以看出, 与 IIS 5.x 不同, W3SVC 从 InetInfo.exe 进程脱离出来 (对于 IIS 6.0来说, InetInfo.exe 基本上可以看作单纯的 IIS 管理进程), 运行在另一个进程 SvcHost.exe 中。不过 W3SVC 的基本功能并没有发生变化, 只是在功能的实现上作了相应的改进。与 IIS 5.x 一样, 元数据库 (Metabase) 依然存在于 InetInfo.exe 进程中。



(点击查看大图) 图7-10 IIS 6.0与 ASP.NET

#### 7.4.2 IIS 管道与 ASP.NET 架构 (2)

当 HTTP.SYS 监听到用户的 HTTP 请求时, 将其分发给 W3SVC。W3SVC 解析出请求的 URL, 并根据从 Metabase 获取的 URL 与 Web 应用之间的映射关系得到目标应用, 并进一步得到目标应用运行的应用程序池或工作进程。如果工作进程不存在 (尚未创建或被回收), 则为该请求创建新的工作进程, 工作进程的这种创建方式被称为请求式创建。在工作进程的初始化

过程中，相应的 ISAPI.dll 被加载，对于 ASP.NET 应用来说，被加载的 ISAPI.dll 为 Aspnet\_ispai.dll。ASP.NET ISAPI 再负责进行 CLR 的加载、AppDomain 创建、Web Application 的初始化等。

## IIS 7.0与 ASP.NET

IIS 7.0在请求的监听和分发机制上又进行了革新性的改进，主要体现在对于 Windows 进程激活服务（Windows Process Activation Service, WAS）的引入，将原来（IIS 6.0）W3SVC 承载的部分功能分流给了 WAS。具体来说，通过上面的介绍，我们知道对于 IIS 6.0 来说，W3SVC 主要承载着3大功能：

HTTP 请求接收：接收 HTTP.SYS 监听到的 HTTP 请求。

配置管理：从元数据库（Metabase）中加载配置信息对相关组件进行配置。

进程管理：创建、回收、监控工作进程。

在 IIS 7.0, 后两组功能被移入 WAS 中，接收 HTTP 请求的任务依然落在 W3SVC 头上。WAS 的引入为 IIS 7.0一项前所未有的特性：同时处理 HTTP 和非 HTTP 请求。在 WAS 中，通过一个重要的接口：监听器适配器接口（Listener Adapter Interface）抽象出不同协议监听器监听到的请求。至于 IIS 下的监听器，除了基于网络驱动的 HTTP.SYS 提供 HTTP 请求监听功能外，WCF 提供了3种类型的监听器：TCP 监听器、命名管道（Named Pipes）监听器和 MSMQ 监听器，分别提供了基于 TCP、命名管道和 MSMQ 传输协议的监听功能。与此3种监听器相对的是3种监听器适配器（Adapter），它们提供监听器与监听器适配器接口之间的适配。从这个意义上讲，IIS 7.0中的 W3SVC 更多地为 HTTP.SYS 起着监听适配器的功能。WCF 提供的这3种监听器和监听适配器定义在程序集 SMHost.exe 中，你可以通过下面的目录找到该程序集：`%windir%\Microsoft.NET\Framework\v3.0\Windows Communication Foundatio`。

WCF 提供的这3种监听器和监听适配器最终以 Windows Service 的形式体现，虽然它们定义在一个程序集中，我们依然通过服务工作管理器（SCM, Service Control Manager）对其进行单独的启动、终止和配置。SMHost.exe 提供了4个重要的 Windows Service：

NetTcpPortSharing：为 WCF 提供 TCP 端口共享，关于端口共享，在本书的第二章有详细介绍。

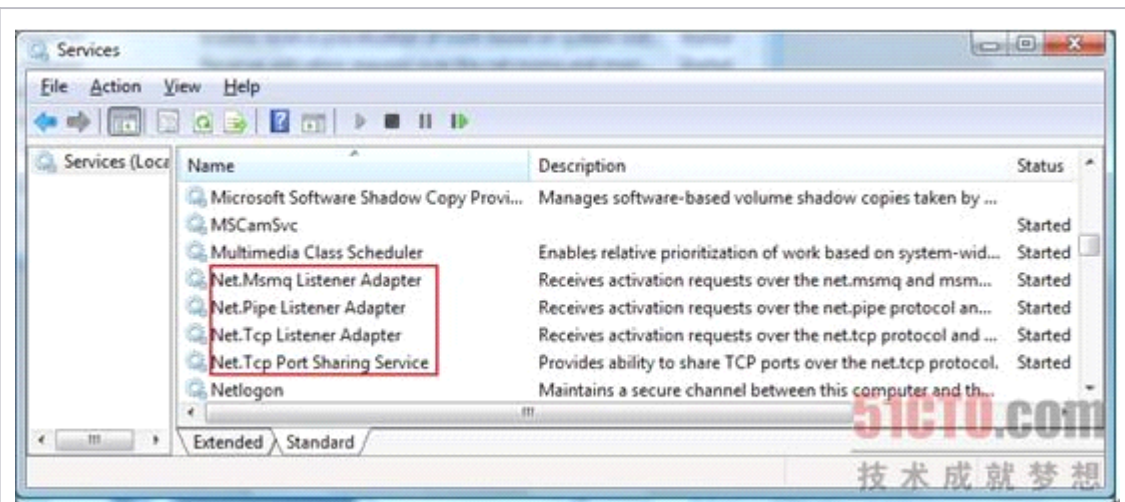
NetTcpActivator：为 WAS 提供基于 TCP 的激活请求，包含 TCP 监听器和对应的监听适配器。

NetPipeActivator：为 WAS 提供基于命名管道的激活请求，包含命名管道监听器和对应的监听适配器。

NetMsmqActivator：为 WAS 提供基于 MSMQ 的激活请求，包含 MSMQ 监听器和对应的监听

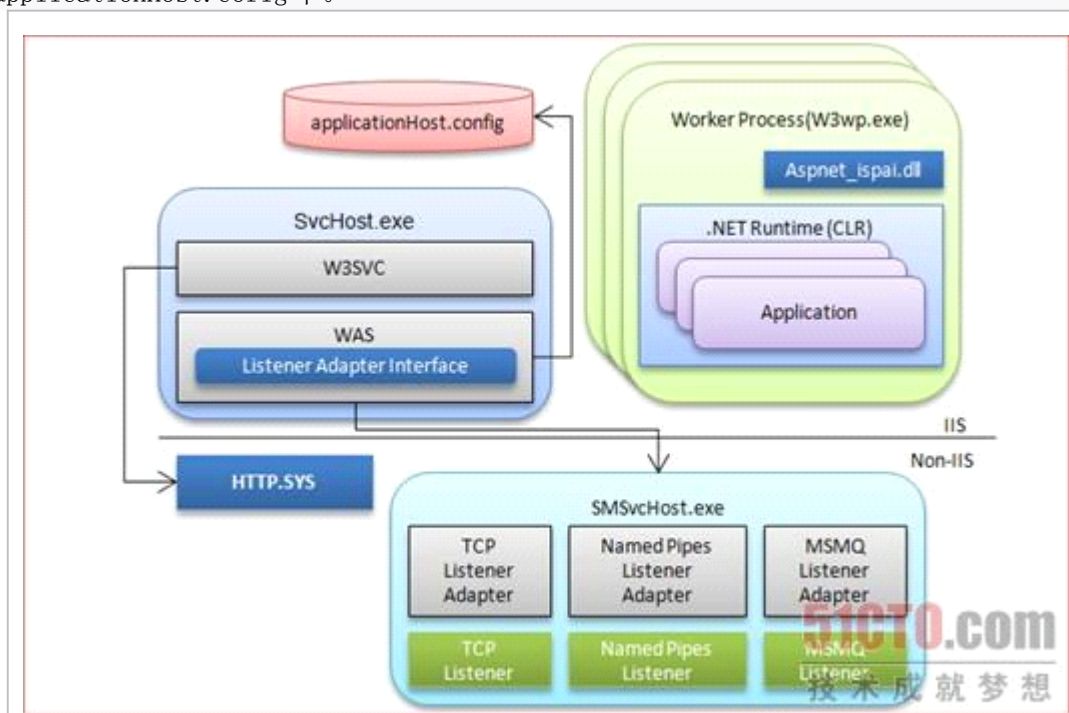
适配器。

图7-11为上述的4个 Windows Service 在服务控制管理器（SCM）中的呈现。



（点击查看大图）图7-11 定义在 SMHost.exe 中的 Windows Service

图7-12揭示了 IIS 7.0 的整体构架及整个请求处理流程。无论是从 W3SVC 接收到的 HTTP 请求，还是通过 WCF 提供的监听适配器接收到的请求，最终都会传递到 WAS。如果相应的工作进程（或者应用程序池）尚未创建，其创建之；否则将请求分发给对应的工作进程进行后续的处理。WAS 在进行请求处理过程中，通过内置的配置管理模块加载相关的配置信息，并对相关的组件进行配置，与 IIS 5.x 和 IIS 6.0 基于 Metabase 的配置信息存储不同的是，IIS 7.0 大都将配置信息存放于 XML 形式的配置文件中。基本的配置存放在 applicationHost.config 中。



（点击查看大图）图7-12 IIS 7.0与 ASP.NET

## 在 WAS 中进行非 HTTP 的服务寄宿

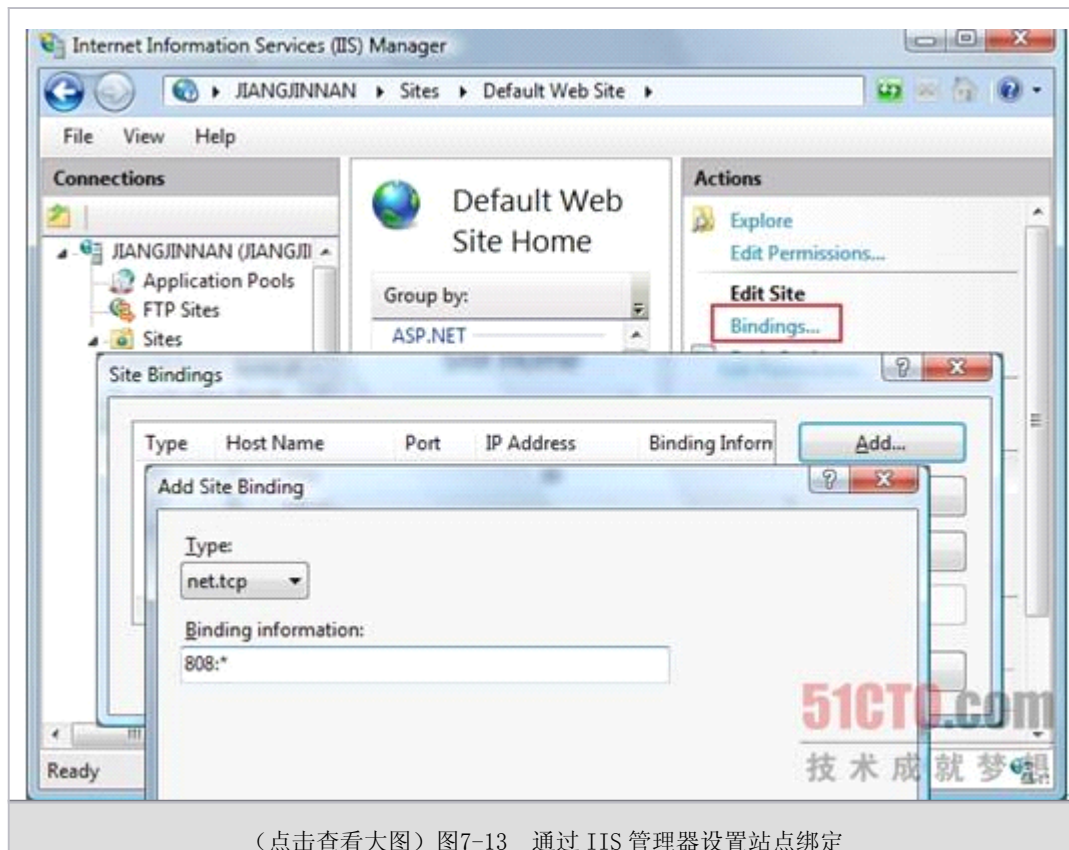
进行非 HTTP 的服务寄宿是 WAS 为 WCF 提供的最显著特性。由于在默认的情况下，IIS 仅仅支持对 HTTP 请求的处理，为了支持非 HTTP 服务寄宿的能力，我们须要用相应的方式对 IIS 相关配置进行相关的修改，从而改变 IIS 默认的请求处理行为。在上面我们说过，IIS 7.0 广泛采用了基于 XML 文件的配置方式，所以最终极的方式就是直接修改相应的配置文件。但是，直接修改配置文件，出错的频率很高，对于很多的配置，我们都可以直接通过 IIS 管理器进行相应的修改。此外，我们可以选择通过命令行的方式修改相应的配置，IIS 为我们提供了一系列的命令。

WAS 的配置保存在配置文件 applicationHost.config 中，该文件保存在%windir%\system32\inetsrv\config目录中。为了实现基于非HTTP的服务寄宿，首先需要做的是为WCF Service 的寄宿应用所在的 Web Site 添加相应的非 HTTP 协议的站点绑定 (site binding)，该操作可以通过执行 Appcmd.exe 命名实现，该命名存放在%windir%\system32\ inetsrv\ 目录中。

以7.4.1节的案例（如何通过 IIS 进行服务寄宿，见第334页）为例，CalculatorService 计算在默认 Web Site 下的 IISHostingDemo Application 下。我们可以通过下面的命令行为 Web Site 添加 net.tcp 的站点绑定。

```
1 appcmd.exe set site "Default Web Site"  
  -+bindings.[protocol='net.tcp',  
2    bindingInformation='808:*']
```

站点绑定添加与修改也可以直接通过 IIS 管理器进行：选择相应站点→点击右边的“Bindings”→在弹出的 Site Bindings 对话框中可以添加新的站点绑定和编辑现有的站点绑定，如图7-13所示。



在站点级别非 HTTP 绑定存在的情况下，还可在应用级别控制对非 HTTP 协议的支持。在默认的情况下，Web 应用并不提供对非 HTTP 协议的支持，须要通过 AppCmd.exe 为应用添加对于某个非 HTTP 协议支持的能力。通过下面的配置对默认站点下的 IISHostingDemo 应用添加了对 net.tcp 支持的能力。

```
3 appcmd.exe set app "Default Web Site/IISHostingDemo"
4 /enabledProtocols:net.tcp
```

#### 7.4.2 IIS 管道与 ASP.NET 架构 (3)

经过了上面两个步骤，7.4.1节的案例（见第334页）中寄宿在 Web 应用 IISHostingDemo 下的 CalculatorService 就可以采用 NetTcpBinding 进行基于 TCP 的服务寄宿了，我们只须改变终结点的绑定类型。客户端采用相同的绑定通过访问 .svc 文件进行服务调用。

```
1 <?xml version="1.0"?>
2 <configuration>
3   <system.serviceModel>
4     <services>
5       <service name="CalculatorService">
6         <endpoint address="" binding="
netTcpBinding" contract=
7           "Artech.IISHostingDemo.
Contracts.ICalculator" />
```

```

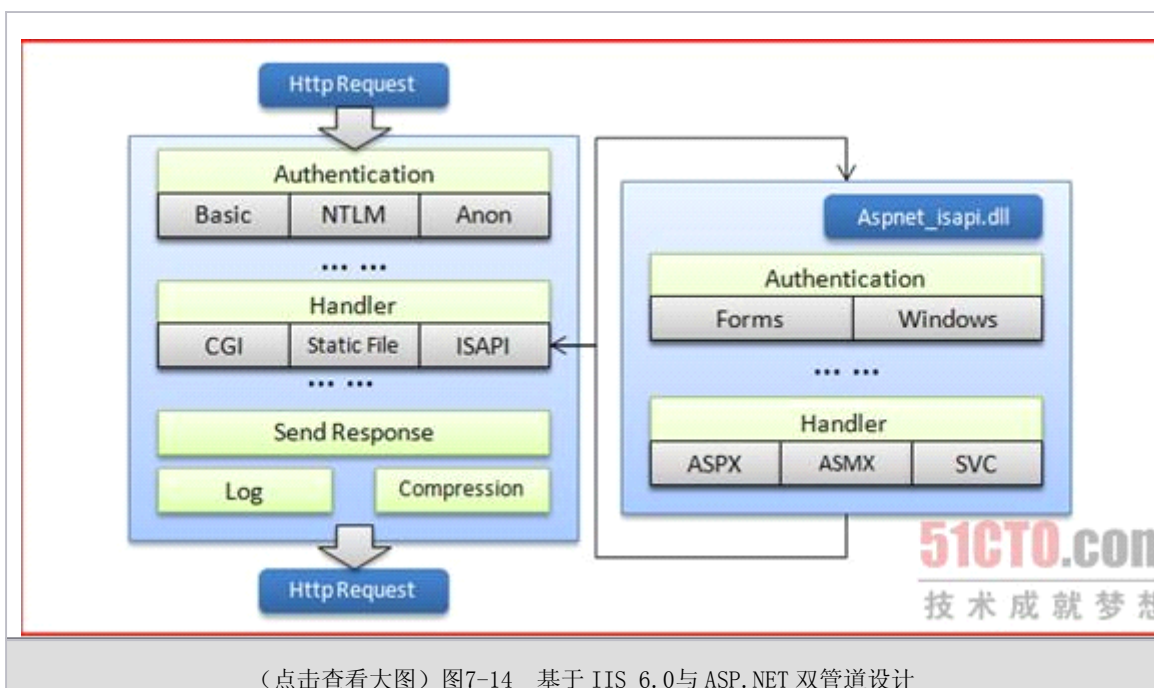
8         </service>
9     </services>
10 </system.serviceModel>
11 </configuration>
12
13 <?xml version="1.0" encoding="utf-8" ?>
14 <configuration>
15     <system.serviceModel>
16         <client>
17             <endpoint name="CalculatorService"
address="net.tcp://localhost/
18             IISHostingDemo/CalculatorService.
svc" binding="netTcpBinding"
19             contract="Artech.IISHostingDemo.
Contracts.ICalculator"/>
20         </client>
21     </system.serviceModel>
22 </configuration>

```

### ASP.NET 集成

从上面对 IIS 5.x 和 IIS 6.0 的介绍中, 我们不难发现这一点, IIS 与 ASP.NET 是两个相互独立的管道 (Pipeline), 在各自管辖范围内, 它们各自具有自己的一套机制对 HTTP 请求进行处理。两个管道通过 ISAPI 实现“连通”: IIS 是第一道屏障, 当对 HTTP 请求进行必要的前期处理 (比如身份验证等) 时, 通过 ISAPI 将请求分发给 ASP.NET 管道。当 ASP.NET 在自身管道范围内完成对 HTTP 请求的处理时, 处理后的结果再返回到 IIS, IIS 对其进行后期处理 (比如日志记录、压缩等), 最终生成 HTTP 响应 (HTTP Response)。从另一个角度讲, IIS 运行在非托管的环境中, 而 ASP.NET 管道则是托管的, 从这个意义上讲, ISAPI 还是连接非托管环境和托管环境的纽带。图7-14反映了 IIS 6.0 与 ASP.NET 之间的桥接关系。





(点击查看大图) 图7-14 基于 IIS 6.0与 ASP.NET 双管道设计

IIS 5.x 和 IIS 6.0把两个管道进行隔离至少带来了下面一些局限与不足。

相同操作的重复执行：IIS 与 ASP.NET 之间具有一些重复的操作，比如身份验证。

动态文件与静态文件处理的不一致：因为只有基于 ASP.NET 动态文件（比如.aspx、.asmx、.svc 等等）的 HTTP 请求才能通过 ASP.NET ISAPI 进入 ASP.NET 管道，而对于一些静态文件（比如.html、.xml、.img 等）的请求，则由 IIS 直接响应，那么 ASP.NET 管道中的一些功能将不能用于这些基于静态文件的请求，比如，我们希望通过 Forms 认证应用于基于图片文件的请求。

IIS 难以扩展：对于 IIS 的扩展基本上就体现在自定义 ISAPI，但是对于大部分人来说，这不是一件容易的事情。因为 ISAPI 是基于 Win32的非托管的 API，并非一种面向应用的编程接口。通常我们希望的是诸如定义 ASP.NET 的 HttpModule 和 HttpHandler 一样，通过托管代码的方式来扩展 IIS。

对于 Windows 平台下的 IIS 来讲，ASP.NET 无疑是一等公民，它们之间不应该是“井水不犯河水”的关系，而应该是“你中有我，我中有你”的关系。为此，在 IIS 7.0中，实现了两者的集成。对于集成模式下的 IIS 7.0，我们获得如下的好处。

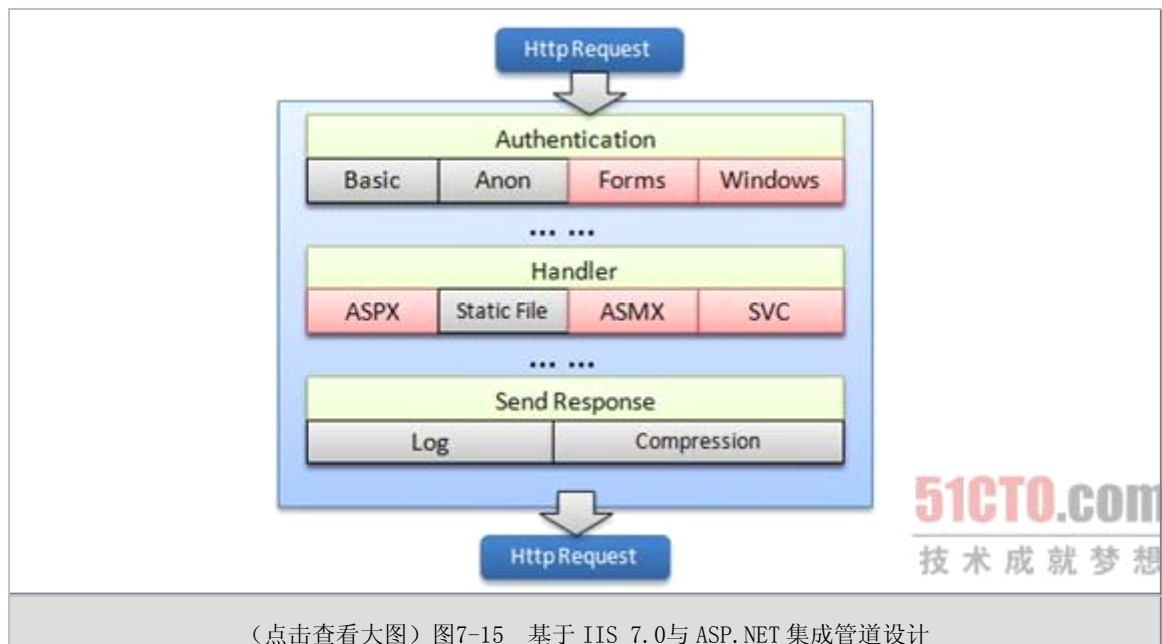
允许通过本地代码（Native Code）和托管代码（Managed Code）两种方式定义 IIS Module，这些 IIS Module 注册到 IIS 中形成一个通用的请求处理管道。由这些 IIS Module 组成的这个管道能够处理所有的请求，不论请求基于怎样的资源类型。比如，可以将 FormsAuthenticationModule 提供的 Forms 认证应用到基于.aspx、CGI 和静态文件的请求；

将 ASP.NET 提供一些强大的功能应用到原来难以企及的地方，比如将 ASP.NET 的 URL

重写功能置于身份验证之前；

采用相同的方式去实现、配置、检测和支持一些服务器特性 (Feature)，比如 Module、Handler 映射、定制错误配置 (Custom Error Configuration) 等。

图7-15演示了在 ASP.NET 集成模式下，IIS 整个请求处理管道的结构。可以看到，原来 ASP.NET 提供的托管组件可以直接应用在 IIS 管道中。



## ASP.NET 管道

以 IIS 6.0 为例，在工作进程 w3wp.exe 中，利用 AspNet\_ispai.dll 加载 .NET 运行时（如果 .NET 运行时尚未加载）。IIS 6.0 引入了应用程序池的概念，一个工作进程对应着一个应用程序池。一个应用程序池可以承载一个或多个 Web 应用，每个 Web 应用映射到一个 IIS 虚拟目录。与 IIS 5.x 一样，每一个 Web 应用运行在各自的应用程序域中。

如果 HTTP.SYS 接收到的 HTTP 请求是对该 Web 应用的第一次访问，在成功加载了运行时后，会通过 AppDomainFactory 为该 Web 应用创建一个应用程序域 (AppDomain)。随后，一个特殊的运行时 IsapiRuntime 被加载。IsapiRuntime 定义在程序集 System.Web 中，对应的命名空间为 System.Web.Hosting。IsapiRuntime 会接管该 HTTP 请求。

IsapiRuntime 会首先创建一个 IsapiWorkerRequest 对象，用于封装当前的 HTTP 请求，并将该 IsapiWorkerRequest 对象传递给 ASP.NET 运行时：HttpRuntime，从此时起，HTTP 请求正式进入了 ASP.NET 管道。根据 IsapiWorkerRequest 对象，HttpRuntime 会创建用于



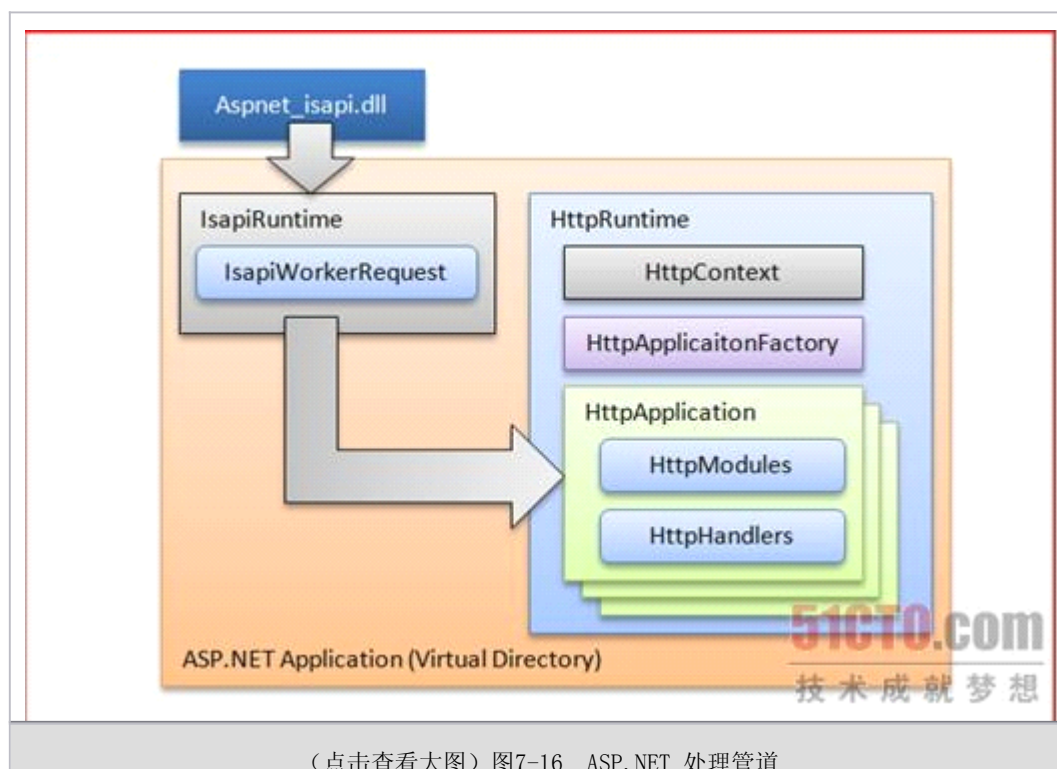
表示当前 HTTP 请求的上下文 (Context) 对象: HttpContext。

随着 HttpContext 被成功创建, HttpRuntime 会利用 HttpApplicationFactory 创建新的或获取现有的 HttpApplication 对象。实际上, ASP.NET 维护着一个 HttpApplication 对象池, HttpApplicationFactory 从池中选取可用的 HttpApplication 用户处理 HTTP 请求, 处理完毕后将其释放到对象池中。HttpApplicationFactory 负责处理当前的 HTTP 请求。

在 HttpApplication 初始化过程中, 会根据配置文件加载并初始化相应的 HttpModule 对象。对于 HttpApplication 来说, 在它处理 HTTP 请求的不同阶段会触发不同的事件 (Event), 而 HttpModule 的意义在于通过注册 HttpApplication 的相应的事件, 将所需的操作注入整个 HTTP 请求的处理流程。ASP.NET 的很多功能, 比如身份验证、授权、缓存等, 都是通过相应的 HttpModule 实现的。

#### 7.4.2 IIS 管道与 ASP.NET 架构 (4)

而最终完成对 HTTP 请求的处理实现在另一个重要的对象中: HttpHandler。对于不同的资源类型, 具有不同的 HttpHandler。比如 .aspx 页对应的 HttpHandler 为 System.Web.UI.Page, WCF 的 .svc 文件对应的 HttpHandler 为 System.ServiceModel.Activation.HttpHandler。上面整个处理流程如图7-16所示。



#### HttpApplication

HttpApplication 是整个 ASP.NET 基础架构的核心, 它负责处理分发给它的 HTTP 请求。由于一个 HttpApplication 对象在某个时刻只能处理一个请求, 只有完成对某个请求的处理

后，HttpApplication 才能用于后续的请求的处理。所以，ASP.NET 采用对象池的机制来创建或获取 HttpApplication 对象。具体来讲，当第一个请求抵达的时候，ASP.NET 会一次创建多个 HttpApplication 对象，并将其置于池中，选择其中一个对象来处理该请求。当处理完毕，HttpApplication 不会被回收，而是释放到池中。对于后续的请求，空闲的 HttpApplication 对象会从池中取出，如果池中所有的 HttpApplication 对象都处于繁忙的状态，ASP.NET 会创建新的 HttpApplication 对象。

HttpApplication 处理请求的整个生命周期是一个相对复杂的过程，在该过程的不同阶段会触发相应的事件。我们可以注册相应的事件，将处理逻辑注入到 HttpApplication 处理请求的某个阶段。接下来介绍的 HttpModule 就是通过 HttpApplication 事件注册的机制实现相应的功能的。表7-3按照实现的先后顺序列出了 HttpApplication 在处理每一个请求时触发的事件名称。

表7-3

名称	描述
BeginRequest	HTTP 管道开始处理请求时， 会触发 BeginRequest 事件
AuthenticateRequest ， PostAuthenticateRequest	ASP.NET 先后触发这两个事件， 使安全模块对请求进行身份验证
AuthorizeRequest， PostAuthorizeRequest	ASP.NET 先后触发这两个事件， 使安全模块对请求进程授权
ResolveRequestCache ， PostResolveRequestCache	ASP.NET 先后触发这两个事件， 以使缓存模块利用缓存的直接 对请求直接进程响应（缓存模块 可以将响应内容进程缓存，对于 后续的请求，直接将缓存的内容返回， 从而提高响应能力）
PostMapRequestHandler	对于访问不同的资源类型， ASP.NET 具有不同的 HttpHandler 对其进程处理。对于每个请求， ASP.NET 会通过扩展名选择匹配

	相应的 <code>HttpHandler</code> 类型，成功匹配后，该实现被触发
<code>AcquireRequestState</code> , <code>PostAcquireRequestState</code>	ASP.NET 先后触发这两个事件，使状态管理模块获取基于当前请求相应的状态，比如 <code>SessionState</code>
<code>PreRequestHandlerExecute</code> , <code>PostRequestHandlerExecute</code>	ASP.NET 最终通过一请求资源类型相对应的 <code>HttpHandler</code> 实现对请求的处理，在实行 <code>HttpHandler</code> 前后，这两个实现被先后触发
<code>ReleaseRequestState</code> , <code>PostReleaseRequestState</code>	ASP.NET 先后触发这两个事件，使状态管理模块释放基于当前请求相应的状态
<code>UpdateRequestCache</code> , <code>PostUpdateRequestCache</code>	ASP.NET 先后触发这两个事件，以使缓存模块将 <code>HttpHandler</code> 处理请求得到的相应保存到输出缓存中
<code>LogRequest</code> , <code>PostLogRequest</code>	ASP.NET 先后触发这两个事件为当前请求进程日志记录
<code>EndRequest</code>	整个请求处理完成后， <code>EndRequest</code> 事件被触发

对于一个 ASP.NET 应用来说，`HttpApplication` 派生于 `global.asax` 文件，我们可以通过创建 `global.asax` 文件对 `HttpApplication` 的请求处理行为进行定制。`global.asax` 采用一种很直接的方式实现了这样的功能，这种方式不是我们常用的方法重写（`Method Overriding`）或事件注册，而是直接采用方法名匹配。在 `global.asax` 中，我们按照这样的方法命名规则进行事件注册：`Application_{Event Name}`。比如 `Application_BeginRequest` 方法用于处理 `HttpApplication` 的 `BeginRequest` 事件。如何通过 VS 创建一个 `global.asax` 文件，下面是默认的定义。

```

1  <%@ Application Language="C#" %>
2  <script runat="server">
3      void Application_Start(object sender, EventArgs e)
4      {}
5      void Application_End(object sender, EventArgs e)

```

```

6      {}
7      void Application_Error(object sender, EventArgs e)
8      {}
9      void Session_Start(object sender, EventArgs e)
10     {}
11     void Session_End(object sender, EventArgs e)
12     {}
13 </script>

```

#### 7.4.2 IIS 管道与 ASP.NET 架构 (5)

##### HttpModule

ASP.NET 为创建各种 .NET Web 应用提供了强大的平台，它拥有一个具有高度可扩展性的引擎，并且能够处理对于不同资源类型的请求。那么，是什么成就了 ASP.NET 的高可扩展性呢？HttpModule 功不可没。

从功能上讲，HttpModule 之于 ASP.NET，就好比 ISAPI Filter 之于 IIS 一样。IIS 将接收到的请求分发给相应的 ISAPI Extension 之前，注册的 ISAPI Filter 会先截获该请求。ISAPI Filter 可以获取甚至修改请求的内容，完成一些额外的功能。与之相似地，当请求转入 ASP.NET 管道时，最终负责处理该请求的是与请求资源类型相匹配的 HttpHandler 对象，但是在 Handler 正式工作之前，ASP.NET 会先加载并初始化所有配置的 HttpModule 对象。HttpModule 在初始化的过程中，会将一些功能注册到 HttpApplication 相应的事件中，那么在 HttpApplication 整个请求处理生命周期中的某个阶段，相应的事件会被触发，通过 HttpModule 注册的事件处理程序也得以执行。

所有的 HttpModule 都实现了 IHttpModule 接口，下面是 IHttpModule 的定义。其中 Init 方法用于实现 HttpModule 自身的初始化，该方法接受一个 HttpApplication 对象，有了这个对象，事件注册就很容易了。

```

1  public interface IHttpModule
2  {
3      void Dispose();
4      void Init(HttpApplication context);
5  }

```

ASP.NET 提供的很多基础构件（Infrastructure）功能都是通过相应的 HttpModule 实现的，下面类列出了一些典型的 HttpModule：

**OutputCacheModule:** 实现了输出缓存（Output Caching）的功能。

**SessionStateModule:** 在无状态的 HTTP 协议上实现了基于会话（Session）的状态。

WindowsAuthenticationModule+FormsAuthenticationModule+PassportAuthenticationModule:实现了3种典型的身份认证方式:Windows 认证、Forms 认证和 Passport 认证。

UrlAuthorizationModule + FileAuthorizationModule: 实现了基于 URI 和文件 ACL (Access Control List) 的授权。

而另外一个重要的 HttpModule 与 WCF 相关, 那么就是 System.ServiceModel.Activation.HttpModule。HttpModule 定义在 System.ServiceModel 程序集中, 在默认的情况下, HttpModule 完成了基于 IIS 的寄宿工作, 关于这点, 会在本节的后续部分予以详细介绍。

除了这些系统定义的 HttpModule 之外, 我们还可以自定义 HttpModule。通过 Web.config, 可以很容易地将其注册到 Web 应用中。

### **HttpHandler**

如果说 HttpModule 相当于 IIS 的 ISAPI Filter 的话, 我们可以说 HttpHandler 则相当于 IIS 的 ISAPI Extension, HttpHandler 在 ASP.NET 中扮演请求的最终处理者的角色。对于不同资源类型的请求, ASP.NET 会加载不同的 Handler 来处理, 也就是说.aspx page 与.asmx web service 对应的 Handler 是不同的。

所有的 HttpHandler 都实现了接口 IHttpHandler。下面是 IHttpHandler 的定义, 方法 ProcessRequest 提供了处理请求的实现。

```
6 public interface IHttpHandler
7 {
8     void ProcessRequest(HttpContext context);
9     bool IsReusable { get; }
10 }
```

对于某些 HttpHandler, 具有一个与之相关的 HttpHandlerFactory, 用于创建或获取相应的 HttpHandler。HttpHandlerFactory 实现接口 IHttpHandlerFactory, 方法 GetHandler 用于创建新的 HttpHandler, 或者获取已经存在的 HttpHandler。

```
11 public interface IHttpHandlerFactory
12 {
13     IHttpHandler GetHandler(HttpContext
context, string requestType,
14         string url, string pathTranslated);
15     void ReleaseHandler(IHttpHandler handler);
16 }
```

HttpHandler 和 HttpHandlerFactory 的类型都可以通过相同的方式配置到 Web.config

中。下面一段配置包含对3种典型的资源类型的 HttpHandler 配置：.aspx、.asmx 和.svc。可以看到基于 WCF Service 的 HttpHandler 类型为：System.ServiceModel.Activation.HttpHandler。关于 HttpHandler 的实现，会在本节的后续部分予以详细介绍。

```
17 xml version="1.0" encoding="utf-8" ?>
18 <configuration>
19   <system.web>
20     <httpHandlers>
21       <add path="*.svc" verb="*" type="
System.ServiceModel.Activation.
22         HttpHandler, System.ServiceModel,Version=3.0.0.0, Culture=
23         neutral, PublicKeyToken=
b77a5c561934e089" validate="false"/>
24       <add path="*.aspx" verb="*" type="System.Web.UI.
25         PageHandlerFactory" validate="True"/>
26       <add path="*.asmx" verb="*" type="System.Web.Services.Protocols.
27         WebServiceHandlerFactory,
System.Web.Services, Version=2.0.0.0,
28         Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
29         validate="False"/>
30     </>httpHandlers
31   </>system.web
32 </>configuration
```

### 7.4.3 IIS 服务寄宿实现详解（1）

通过上面的介绍，相信读者已经对 IIS 和 ASP.NET 的请求处理管道有了一个大致的了解，在此基础上理解基于 IIS 服务寄宿的实现机制就显得相对容易了。概括地说，基于 IIS 的服务寄宿依赖于两个重要的对象：System.ServiceModel.Activation.HttpModule 和 System.ServiceModel.Activation.HttpHandler。

#### 通过 HttpModule 实现服务寄宿

在默认的情况下，基于 IIS 的服务寄宿是通过一个特殊的 HttpModule 实现的，其类型为 System.ServiceModel.Activation.HttpModule，是一个定义在 System.ServiceModel 程序集中的内部类型。HttpModule 的定义大体上如下面的代码所示，我们很清楚地看到其实现的原理：将实现 WCF Service 请求处理的逻辑注册到 HttpApplication 的 PostAuthenticationRequest 事件中。

```
1 internal class HttpModule : IHttpModule
2 {
3     //其他成员
```

```

4      public void Init(HttpApplication context)
5      {
6          context.PostAuthenticateRequest +=
new EventHandler(HttpModule.
7              ProcessRequest);
8      }
9      private static void ProcessRequest(object sender, EventArgs e)
10     {
11         //服务请求处理实现
12     }
13 }

```

System.ServiceModel.Activation.HttpModule 是一个特殊的 HttpModule, 说它特别是因为当 HttpModule 注册到 HttpApplication 的 PostAuthenticateRequest 事件处理程序执行时, 不会再将请求进一步分发给后续的请求处理步骤。换句话说, 就 HttpApplication 从 BeginRequest 到 EndRequest 整个请求处理的生命周期来说, 对于基于 .svc 文件的请求仅仅延续到 PostAuthenticateRequest 阶段。我们可以通过一种简单的方式来证明这一点。

在 7.4.1 节的案例 (如何通过 IIS 进行服务寄宿, 见第 334 页) 中, 演示了基于 IIS 进行 WCF 服务寄宿的必要步骤。我们将服务实现和相应的 .Svc 文件定义在一个对应于虚拟目录的 ASP.NET Website 中, 现在为之添加一个 global.asax。在该 global.asax, 通过如下的代码注册了 HttpApplication 处理请求的前 3 个事件: BeginRequest、AuthenticateRequest 和 PostAuthenticateRequest, 当这 3 个事件触发时, 将一段代表当前事件的名称写入 EventLog 中。

```

14 <%@ Application Language="C#" %>
15 <%@ Import Namespace="System.Diagnostics"%>
16 <script runat="server">
17
18     void Application_BeginRequest(object sender, EventArgs e)
19     {
20         string message = string.Format
21         ("BeginRequest Event is raised at {0}",
22             DateTime.Now);
23         EventLog.WriteEntry("Application",
24             message, EventLogEntryType.
25             Information);
26     }
27
28     void Application_AuthenticateRequest
29     (object sender, EventArgs e)
30     {
31         string message = string.Format

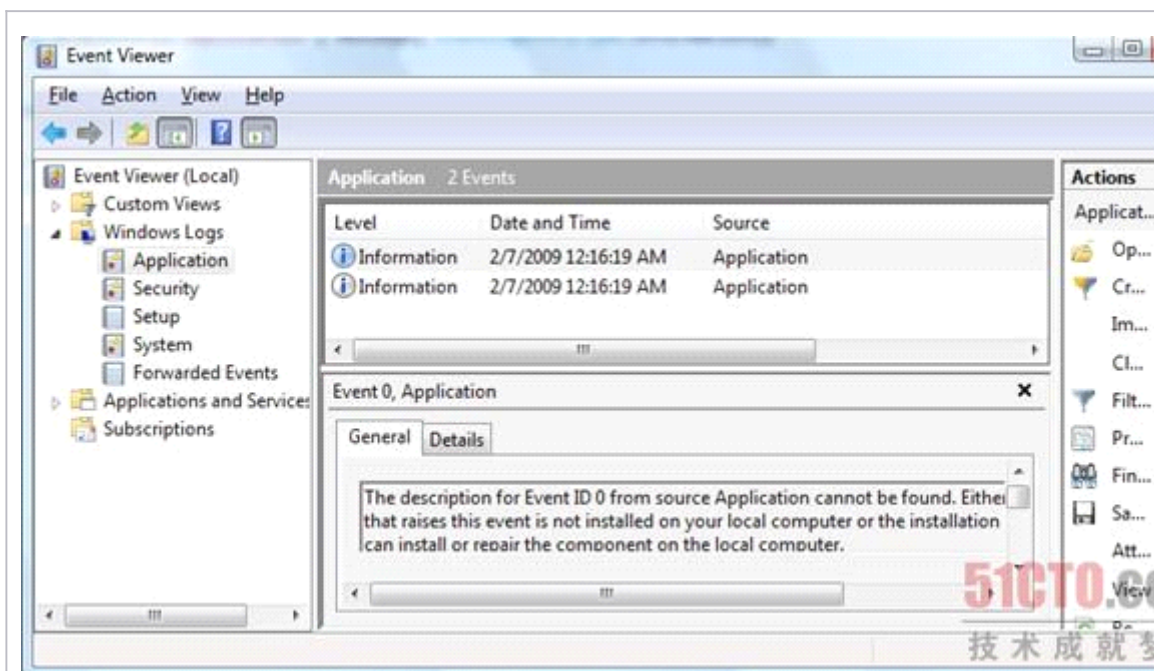
```

```

("AuthenticateRequest Event is raised at
29         {0}", DateTime.Now);
30         EventLog.WriteEntry("Application",
message, EventLogEntryType.
31         Information);
32     }
33
34     void Application_PostAuthenticateRequest
(object sender, EventArgs e)
35     {
36         string message = string.Format
("PostAuthenticateRequest Event is raised
37         at {0}", DateTime.Now);
38         EventLog.WriteEntry("Application",
message, EventLogEntryType.
39         Information);
40     }
41 </script>

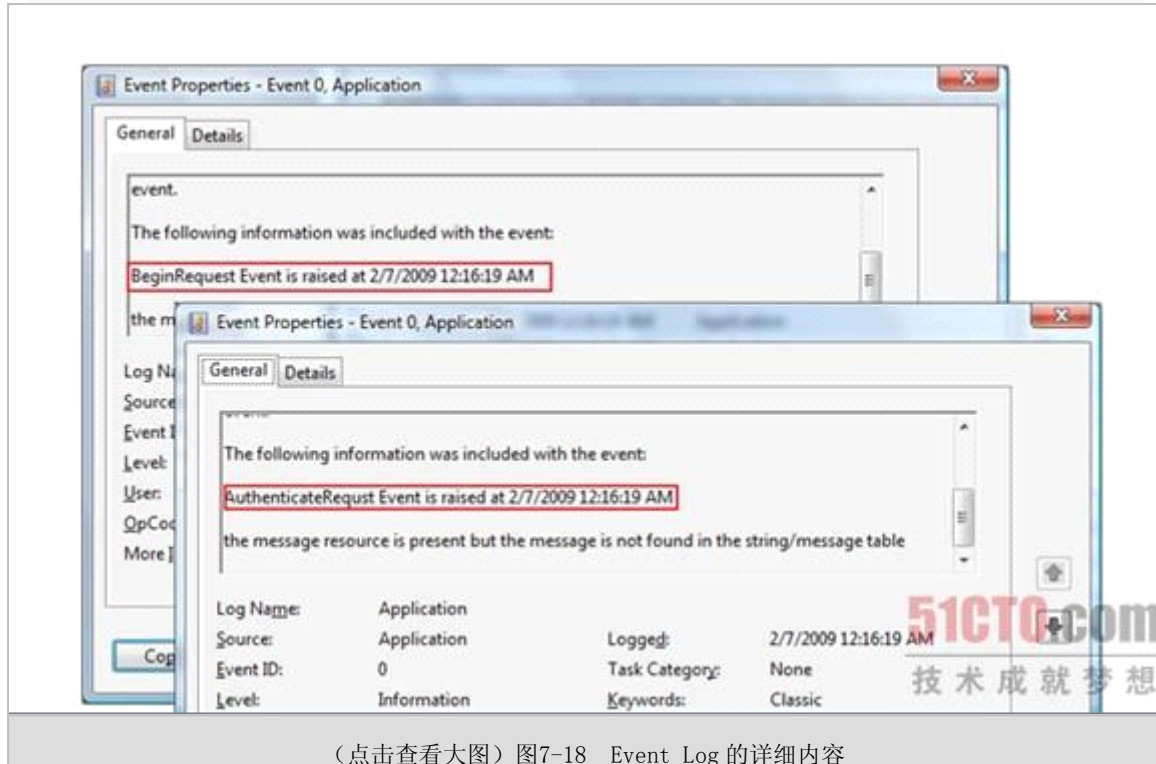
```

如果我们上面的说法成立的话，只有 `HttpApplication` 的最初3个事件被触发。此外，`HttpModule` 注册的操作会先于定义在 `global.asax` 的 `Application_PostAuthenticateRequest` 方法执行，那么在整个服务调用过程中，只有 `Application_BeginRequest` 和 `Application_AuthenticateRequest` 这两个方法会被执行。这一点可以从 `EventLog` 得到证实。当我们执行7.4.1节的案例（如何通过 IIS 进行服务寄宿，见第334页）中的代表客户端应用程序时，`EventLog` 中 `WindowsLog` 的 `Application` 分组中，会多出两个日志项目（之前已经将日志清空），如图7-17所示。





日志的内容正是我们在 Application\_BeginRequest 和 Application\_AuthenticateRequest 方法中定义的日志文本。可见仅仅这两个方法被成功执行，Application\_PostAuthenticateRequest 方法却没有被执行。可以想象，后续的事件也不可能被触发，如图7-18所示。



(点击查看大图) 图7-18 Event Log 的详细内容

### 7.4.3 IIS 服务寄宿实现详解 (2)

到现在为止，我们仅仅介绍了如何处理基于 .svc 文件的请求，并没有说明 .svc 文件对应的 WCF Service 是如何被寄宿的。服务的寄宿发生在对服务 .svc 文件的第一次访问时，具体的实现很简单：ServiceMode 根据请求的目的地址加载相应的 .svc 文件，通过解析定义在 <%ServiceHost%> 指令的 Factory 和 Service 属性得到 ServiceHostFactory 和 Service 的类型（Factory 默认为 System.ServiceModel.ServiceHostFactory），通过反射创建继承自基类 System.ServiceModel.Activation.ServiceHostFactoryBase 的 ServiceHostFactory 对象。最后通过 ServiceHostFactory 创建的继承自基类 System.ServiceModel.ServiceHostBase 的 ServiceHost 对象对 Service 进行寄宿。

#### 自定义 ServiceHost 在基于 IIS 寄宿中的应用

在介绍 WCF 的自我寄宿中，我们曾谈到如何创建和使用自定义 ServiceHost。对于 WCF 的自我寄宿模式，由于整个寄宿过程完全是由自定义的托管代码完成，我们可以直接创建和使用自定义的 ServiceHost 对象。但是，对于 IIS 寄宿来说，整个寄宿过程完全由 IIS 和

ASP.NET 完成，那么在这种情况下，如何应用我们自定义的 ServiceHost 呢？

无论对于何种寄宿方式，最终的寄宿都依赖一个继承自 System.ServiceModel.ServiceHostBase 的 ServiceHost 对象。对于自我寄宿模式来说，ServiceHost 对象可直接通过我们自己的寄宿程序创建，而对于 IIS 寄宿模式，则通过继承自 System.ServiceModel.Activation.ServiceHostFactoryBase 的 ServiceHostFactory 对象创建。基于 IIS 寄宿模式下应用自定义 ServiceHost 的首要步骤就是创建相应的 ServiceHostFactory 类型。ServiceHostFactoryBase 定义了一个构造函数和一个 CreateServiceHost 方法。CreateServiceHost 方法的参数 constructorString 代表用于创建 ServiceHostBase 的一些初始化数据，实际上为 .svc 文件中 <%ServiceHost%> 指令的 Service 属性值。

.svc 文件 <%ServiceHost%> 指令的 Service 属性值一般为服务类型的有效名称，可以包含程序集名称 (AssemblyName)，也可以只含 Type 的 FullName。ServiceHostBase 的创建依赖于服务的类型，对于包含程序集名称的服务类型表示，我们可以直接通过给定的名称加载相应的程序集并得到服务类型。但是，如果仅仅包含类型的名称，WCF 需要加载相应的程序集。程序集的加载过程相对复杂，好在这个过程实现在 ServiceHostFactory 中，所以一般情况下，我们都让自定义的 ServiceHostFactory 继承自 ServiceHostFactory，仅仅重写 CreateServiceHost 方法就可以了。ServiceHostFactoryBase 与 ServiceHostFactory 定义如下：

```
1  public abstract class ServiceHostFactoryBase
2  {
3      protected ServiceHostFactoryBase();
4      public abstract ServiceHostBase
5      CreateServiceHost(string
6      constructorString, Uri[] baseAddresses);
7  }
8  public class ServiceHostFactory : ServiceHostFactoryBase
9  {
10     public ServiceHostFactory();
11     public override ServiceHostBase
12     CreateServiceHost(string
13     constructorString, Uri[] baseAddresses);
14     protected virtual ServiceHost
15     CreateServiceHost(Type serviceType, Uri[]
16     baseAddresses);
17 }
```

在 7.3 节，已经创建了一个自动发布服务元数据的自定义 ServiceHost：MexServiceHost。现在我们通过直接继承 ServiceHostFactory 方式创建自定义的 ServiceHostFactory：MexServiceHostFactory。

```

15 using System;
16 using System.ServiceModel;
17 using System.ServiceModel.Activation;
18 namespace Artech.CustomServiceHost
19 {
20     public class MexServiceHostFactory : ServiceHostFactory
21     {
22         protected override ServiceHost
23 CreateServiceHost(Type serviceType,
24 Uri[] baseAddresses)
25     {
26         return new MexServiceHost
27 (serviceType, baseAddresses);
28     }
29 }

```

MexServiceHostFactory 可以通过<%ServiceHost%>的 Factory 指令指定在服务对应的.svc 文件中。

```

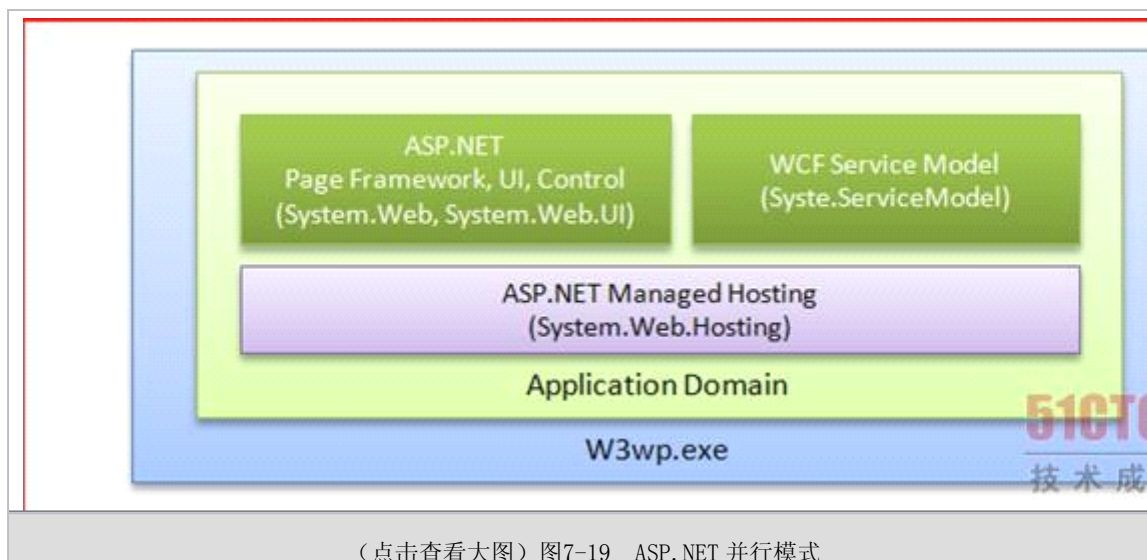
29 <%@ ServiceHost Service="CalculatorService" Factory="Artech.
30 CustomServiceHost.MexServiceHostFactory"%>

```

### ASP.NET 并行 (Side by Side) 模式

对于基于 IIS 服务寄宿，System.ServiceModel.Activation.HttpModule 将基于.svc 的请求劫持并分发给 WCF 的服务模型，从而结束了请求在 ASP.NET 管道的旅程。除了 ASP.NET 提供的一些少量的底层服务，比如动态编译和 AppDomain 管理等，绝大部分 ASP.NET 对传统的 ASP.NET 资源的请求处理机制将不会应用在基于 WCF Service 的请求处理流程中。从这个意义上讲，我们可以说 WCF Service 的运行模式和 ASP.NET 运行时采用的是一种并行的模式。

你完全可以把一个映射到某个 IIS 虚拟目录的 ASP.NET Website 同时作为 asmx Web Service 和.svc WCF Service 的宿主。在这种情况下，ASP.NET .aspx Page、.asmx Web Service 和 WCF Service 运行在同一个 AppDomain 中。但是 HttpRuntime 对于.aspx Page 和.asmx Web Service 的处理机制并不会应用于对.svc WCF Service 请求。我们把 WCF Service 这种寄宿模式称为 ASP.NET 并行 (Side by Side) 模式，图7-19揭示了这种寄宿模式。



在图7-19体现的这种情况下(ASP.NET .aspx Page 和 .svc WCF Service 共存于同一个 AppDomain), .aspx 可以直接定位 WCF Service, 它们之间还可以共享一个基于 AppDomain 的状态, 比如类型的静态属性。但是很多 ASP.NET 特性将不能被 WCF Service 使用, 比如:

HttpContext: 对于 WCF Service 来说, HttpContext.Current 永远为 null。

基于文件或 URL 的授权: 基于 .svc 文件的 ACL(Access Control List)的授权和 ASP.NET 通过 <authorization> 定义的基于 URL 的授权都将失去效力。原因很简单, System.ServiceModel.Activation.HttpModule 在 PostAuthenticateRequest 阶段就将请求劫持, 而授权 (Authorization) 发生在 PostAuthenticateRequest 之后。

HttpModule 扩展: 作用于 PostAuthenticateRequest 事件后期的 HttpModule 将不会生效。

身份模拟 (Impersonation): 即使通过配置 <identity impersonate="true" /> 允许身份模拟, WCF Service 总是运行在 IIS 进程账号下。

### 7.4.3 IIS 服务寄宿实现详解 (3)

不过, WCF 服务模型通过自己的方式解决了上面的问题, 比如:

OperationContext: ASP.NET HttpContext 是基于当前的请求, WCF 的 OperationContext 是基于当前的操作, 本质上是一样的基于上下文的容器。

ServiceAuthorizationBehavior: ServiceAuthorizationBehavior 是一个 Service 行为, 用于实现 WCF 的授权。

DispatchMessageInspector + 自定义 Channel: DispatchMessageInspector 和自定义 Channel 分别在服务模型和信道层对入栈消息进行额外的筛选和处理, 和自定义 HttpModule 异曲同工。

基于操作的身份模拟（Impersonation）：WCF 自身也提供了基于操作的身份模拟实现。

为什么 WCF 要采用这种于 ASP.NET 并行的模式，而不像 Web Service 一样采用与 ASP.NET 完全兼容呢？这主要是因为 WCF 和 .asmx Web Service 有本质的区别：Web Service 总是采用 IIS 寄宿，并使用 HTTP 作为传输，而 WCF 则具有不同的寄宿方式，对于传输协议的选择也没有限制。在默认的情况下，不论采用何种寄宿方式，WCF 本身的行为应该保持一致。所以，让 WCF 服务的行为独立于寄宿的环境与传输协议，是采用并行模式的主要原因。

## ASP.NET 兼容模式

虽然在默认的情况下，IIS 的寄宿采用 ASP.NET 并行的模式。但是在一个 Web 应用中，尤其是一些 AJAX 的 Web 应用，却明确地需要以一种 ASP.NET 兼容模式处理 WCF Service 请求。比如，在 WCF Service 的操作中，须要获取 ASP.NET 应用的 SessionState，或者是需要通过基于 .svc 文件的 ACL 对 WCF Service 进行授权等。

WCF 对此提供了支持，实现起来也很简单，对于编程来说，仅仅需要在 Service 类型加上一个特殊的 `AspNetCompatibilityRequirementsAttribute` 特性，并将 `RequirementsMode` 属性指定为 `AspNetCompatibilityRequirementsMode.Allowed`，事例代码如下：

```
1 [AspNetCompatibilityRequirements(  
2   RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]  
3 public class CalculatorService:ICalculator  
4 {  
5     //省略成员  
6 }
```

除此之外，WCF 的配置也需要做一些修改，我们需要将 `<serviceHostingEnvironment/>` 配置节的 `aspNetCompatibilityEnabled` 属性设为 `true`。

```
7 <?xml version="1.0"?>  
8 <configuration>  
9   <system.serviceModel>  
10     <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>  
11     <!--其他配置-->  
12   </system.serviceModel>  
13 </configuration>
```

在 ASP.NET 兼容模式下，ASP.NET 将会采用与处理 .aspx、.asmx 一样的方式来处理基于 .svc 的请求，对 WCF Service 请求的处理将会贯穿 `HttpApplication` 请求处理的整个生命周期（从 `BeginRequest` 到 `EndRequest`）。对于 ASP.NET 兼容模式，`System.ServiceModel.Activation.HttpModule` 将忽略对 `HttpApplication` 对象 `PostAuthenticateRequest` 事件的注册，原本实现在 `HttpModule` 中对 WCF Service 的请求处理逻辑将实现在一个 `HttpHandler` 中：`System.ServiceModel.Activation.HttpHandler`。如同 `System.Web.UI.Page`（本质上是一个 `HttpHandler`）负责最终处理对 .aspx 的请求一样，`System.ServiceModel.Activation.HttpHandler` 服务负责最终对 .svc 的请求。`HttpHandler` 是一个定义在 `System.ServiceModel` 程序集中的内

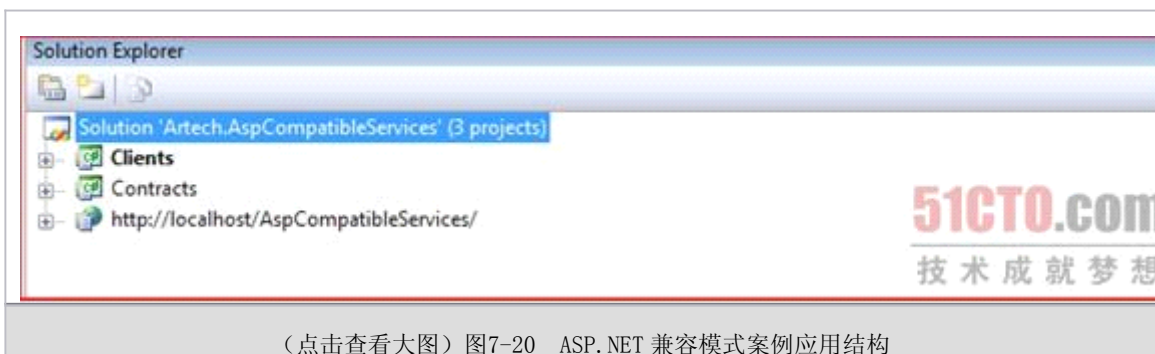
部类型。HttpHandler 的定义如下，请求处理实现在 ProcessRequest 方法中，具体的逻辑与实现在 System.ServiceModel.Activation.HttpModule 中的是完全一致的。

```
14 internal class HttpHandler : IHttpHandler, IRequiresSessionState
15 {
16     public HttpHandler();
17     public void ProcessRequest(HttpContext context);
18
19     public bool IsReusable { get; }
20 }
```

#### 7.4.4 案例演示：利用 ASP.NET 兼容模式创建支持会话 (Session) 的 WCF 服务 (1)

通过上面的介绍，我们对 WCF 两种基于 IIS 的寄宿模式有了一个清晰的认识，并对基于 ASP.NET 兼容模式下的编程有了一个大致的了解。现在，我们通过一个具体的案例进一步加深读者对 ASP.NET 兼容模式的理解。

由于在 ASP.NET 兼容模式下，ASP.NET 采用与 .aspx Page 完全一样的方式处理基于 .svc 的请求，换言之，我们就可以借助当前 HttpContext 的 SessionState 维护会话状态，进而创建一个支持会话的 WCF Service。和上面的一个案例一样，本案例采用如图7-20所示的3层结构。



##### 步骤一 定义服务契约：ICalculator

案例依然沿用计算服务的例子，不过与原来直接通过传入操作数并得到运算结果的方式不同，为了体现会话状态的存在，我们将本案例的 WCF 服务定义成“累积计算服务”：保留上一次运算的结果，并将其作为后续运算的操作数。为此，定义了如下一个接口作为服务契约：前面4个操作代表基本的加、减、乘、除运算，计算结果通过 GetResult 方法获得。

```
1 using System.ServiceModel;
2 namespace Artech.AspCompatibleServices.Contracts
3 {
4     [ServiceContract]
5     public interface ICalculator
6     {
```

```

7         [OperationContract]
8         void Add(double x);
9         [OperationContract]
10        void Subtract(double x);
11        [OperationContract]
12        void Multiply(double x);
13        [OperationContract]
14        void Divide(double x);
15        [OperationContract]
16        double GetResult();
17    }
18 }

```

## 步骤二 实现服务: CalculatorService

服务的实现和.svc 都定义在一个 ASP.NET Web 站点项目中。对于定义在 CalculatorService 中的每次运算, 先通过 HttpContext 从 SessionState 中取出上一次运算的结果, 完成运算后再将新的运算结果保存到 SessionState 中。通过在 CalculatorService 上应用 AspNetCompatibilityRequirementsAttribute 实现对 ASP.NET 兼容模式的支持。

```

19 using System.ServiceModel.Activation;
20 using System.Web;
21 using Artech.AspCompatibleServices.Contracts;
22
23 [AspNetCompatibilityRequirements(
24     RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
25 public class CalculatorService:ICalculator
26 {
27     public void Add(double x)
28     {
29         HttpContext.Current.Session["__Result"] = GetResult() + x;
30     }
31
32     public void Subtract(double x)
33     {
34         HttpContext.Current.Session["__Result"] = GetResult() - x;
35     }
36
37     public void Multiply(double x)
38     {
39         HttpContext.Current.Session["__Result"] = GetResult() * x;
40     }
41

```

```

42     public void Divide(double x)
43     {
44         HttpContext.Current.Session["__Result"] = GetResult() / x;
45     }
46
47     public double GetResult()
48     {
49         if (HttpContext.Current.Session["__Result"] == null)
50         {
51             HttpContext.Current.Session["__Result"] = 0.0;
52         }
53         return (double)HttpContext.Current.Session["__Result"];
54     }
55 }

```

下面是 CalculatorService 对应的 .svc 的定义和 Web.config。为了简洁，在 <@ServiceHost%>指令中，仅仅设置一个必需属性 Service。对于 ASP.NET 兼容模式的支持，配置<serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>必不可少。

```

56 <%@ ServiceHost Service="CalculatorService" %>
57 <?xml version="1.0"?>
58 <configuration>
59     <system.serviceModel>
60         <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
61         <services>
62             <service name="CalculatorService">
63                 <endpoint binding="wsHttpBinding" contract="Artech.
64                     AspCompatibleServices.Contracts.ICalculator" />
65             </service>
66         </services>
67     </system.serviceModel>
68 </configuration>

```

#### 7.4.4 案例演示：利用 ASP.NET 兼容模式创建支持会话（Session）的 WCF 服务（2）

##### 步骤三 创建客户端：Client

CalculatorService 的客户端应用通过一个 Console 应用程序模拟，其服务调用方式并无特别之处，下面是相关的代码和配置。

```

1  using System;
2  using System.ServiceModel;
3  using Artech.AspCompatibleServices.Contracts;
4  namespace Artech.AspCompatibleServices.Clients
5  {
6      class Program

```



```

7      {
8          static void Main(string[] args)
9          {
10             using (ChannelFactory<ICalculator> channelFactory = new
11                 ChannelFactory<ICalculator>("CalculatorService"))
12             {
13                 ICalculator proxy = channel
14 Factory.CreateChannel();
15                 Console.WriteLine("初始值为:
16 {0}", proxy.GetResult());
17                 proxy.Add(1);
18                 Console.WriteLine("Add(3)",
19 proxy.GetResult());
20                 Console.WriteLine("运算结果为:
21 {0}", proxy.GetResult());
22                 proxy.Multiply(10);
23                 Console.WriteLine("Multiply(10)",
24 proxy.GetResult());
25                 Console.WriteLine("运算结果为: {0}",
26 proxy.GetResult());
27                 proxy.Subtract(2);
28                 Console.WriteLine("Subtract(2)",
29 proxy.GetResult());
30                 Console.WriteLine("运算结果为: {0}",
31 proxy.GetResult());
32             }
33             Console.Read();
34         }
35     }
36 }
37
38 <?xml version="1.0" encoding="utf-8" ?>
39 <configuration>
40     <system.serviceModel>
41         <client>
42             <endpoint address="http://
43 localhost/AspCompatibleServices/
44 CalculatorService.svc"
45             binding="wsHttpBinding" contract="Artech.
46             AspCompatibleServices.Contracts.ICalculator"
47             name="CalculatorService"/>
48         </client>
49     </system.serviceModel>
50 </configuration>

```

但运行客户端的程序，输出的结果并不像我们希望的那样。从下面的结果可以看出，每

次通过 `GetResult()` 方法得到的结果都是0，也就是说，服务端并没有将运算结果保存下来。

```
41 初始值为: 0
42 Add (3)
43 运算结果为: 0
44 Multiply (10)
45 运算结果为: 0
46 Subtract (2)
47 运算结果为: 0
48 允许 Cookie 传递
```

要解释这个问题，得从 Session 的实现机制说起。众所周知，HTTP 是无状态(Stateless)的传输协议，对于服务端来说，它收到的每个 HTTP 请求都是全新的请求。ASP.NET 会话(Session)的实现很简单，就是让每次 HTTP 请求携带 Session 的识别信息(Session ID)，那么服务就可以根据此信息判断请求来自哪个客户端了。关于 Session 识别信息的保存，ASP.NET 有两种方式：Cookie 和 URL，前者将其放到 Cookie 中，每次 HTTP 请求将会携带该 Cookie 的值，后者则将其作为请求 URL 的一部分。一般情况下采用基于 Cookie 的实现机制，如果 Cookie 禁用则采用后者。

那么对于 ASP.NET 兼容模式下的 WCF 也一样，要想让服务端能够识别会话，就需要让每个服务调用的 HTTP 请求携带 Session 的识别信息，我们也可以通过传递 Cookie 的方式来解决这个问题。对于 WCF 来说，Cookie 传递能够通过 Binding 来控制，对于 `WsHttpBinding` 来说，默认情况下并不允许 Cookie 的传递。我们可以通过 `WsHttpBinding` 的 `AllowCookies` 来控制是否允许传递 Cookie，该属性可以通过配置进行设置。为此，我们对客户端的配置进行了如下的修改。再次运行案例程序，将会得到你期望的输出。

```
49 <?xml version="1.0" encoding="utf-8" ?>
50 <configuration>
51     <system.serviceModel>
52         <client>
53             <endpoint address="
http://localhost/AspCompatibleServices/
54                 CalculatorService.svc"
55                 binding="wsHttpBinding" contract="Artech.
56                 AspCompatibleServices.Contracts.ICalculator"
57                 name="CalculatorService" bindingConfiguration="
58                 "CookieAllowableBinding"/>
59         </client>
60         <bindings>
61             <wsHttpBinding>
62                 <binding name="
CookieAllowableBinding" allowCookies="true"/>
63             </wsHttpBinding>
```

```
64         </bindings>
65     </system.serviceModel>
66 </configuration>
```

客户端输出结果:

```
67 初始值为: 0
68 Add (3)
69 运算结果为: 3
70 Multiply (10)
71 运算结果为: 30
72 Subtract (2)
73 运算结果为: 28
```

## 7.5 通过 Windows Service 进行服务寄宿

Windows Service, 全名为 Windows NT Service, 为我们提供了一种独特的进程模型 (Process Model), 这种进程模型适合那种不需要 UI 的、长时间运行于后台的应用程序。Windows Service 为 WCF 的寄宿提供了一种很好的解决方案, 基于 Windows Service 的服务寄宿具有下面一些特点。

**显示进程激活:** 进程的激活依赖于 Windows Service 的启动, 这与 IIS 通过 IIS 请求式进行激活不同。我们可以通过手工的方式启动 Windows Service, 也可以通过配置使 Windows Service 在机器启动时自动激活。

**易于管理:** Windows 为我们提供了一个有关的管理工具: 服务控制管理器 (SCM: Service Control Manager), 便于我们进行 Windows Service 的管理, 包括查看、启动、终止和配置每一个安装的 Windows Service。

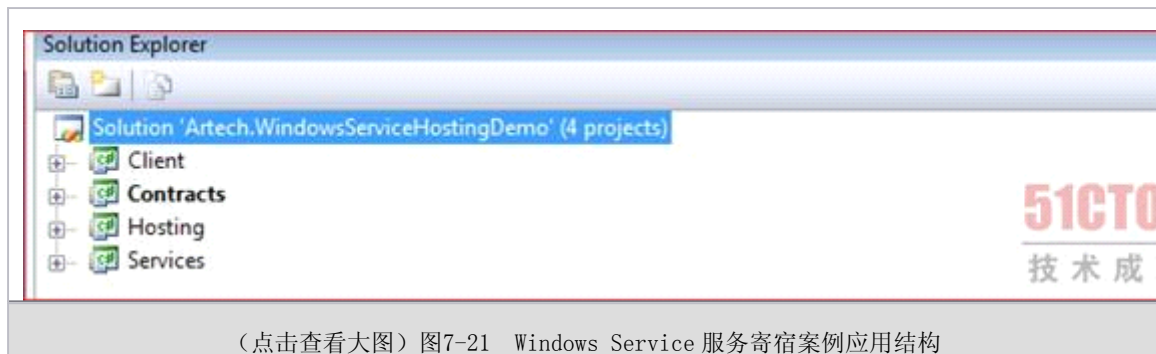
**多传输协议支持:** 我们可以通过托管代码编写用于寄宿 WCF 服务的 Windows Service, 显式控制服务开启、关闭的逻辑, 和一般应用程序一样, Windows Service 也可以具有自己的配置文件, 所以和 WCF 的自我寄宿一样, 可以采用任意的 Binding, 以提供对不同传输协议的支持。

由于 Windows Service 本身并不是本书介绍的重点, 所以在这里不会着太多的笔墨对 Windows Service 进行过多的介绍, 而是通过一个简单的案例演示基于 Windows Service 进行服务寄宿的基本步骤。

### 7.5.1 案例演示: 如何通过创建 Windows Service 寄宿 WCF 服务

整个解决方案的结构如图7-21所示。和本书一贯采用的结构一样, Contracts 和 Services 为一般的 Class Library 项目, 用于定义服务契约和服务。Hosting 为 Windows Services, 定义了实现服务寄宿的 Windows Service 和相应的安装器 (Installer)。Client

为 Console 应用，模拟客户端。



### 步骤一 创建服务契约和服务

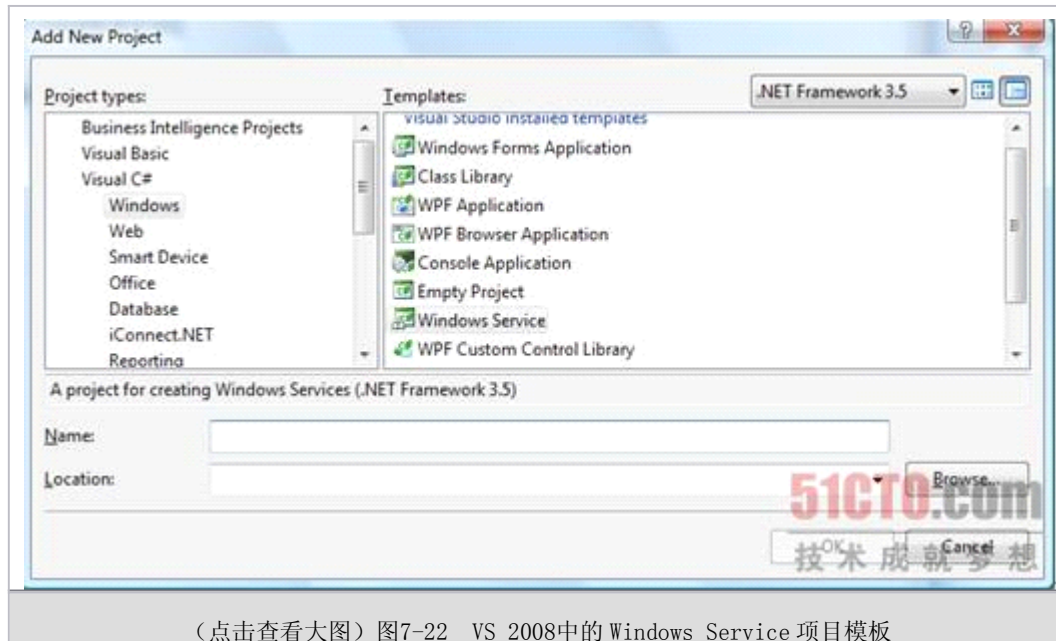
本案例依然沿用我们熟悉的计算服务的例子，下面是服务契约接口和实现接口的服务类。

```
1 using System.ServiceModel;
2 namespace Artech.WindowsServiceHostingDemo.Contracts
3 {
4     [ServiceContract(Namespace="http://www.artech.com/artech/")]
5     public interface ICalculator
6     {
7         [OperationContract]
8         double Add(double x, double y);
9     }
10 }
11 using Artech.WindowsServiceHostingDemo.Contracts;
12 namespace Artech.WindowsServiceHostingDemo.Services
13 {
14     public class CalculatorService:ICalculator
15     {
16         #region ICalculator Members
17
18         public double Add(double x, double y)
19         {
20             return x + y;
21         }
22
23         #endregion
24     }
25 }
```

### 步骤二 创建 Windows Service 项目

在添加新项目模板列表中,选择 Windows Service 选项创建一个 Windows Service 项目

Hosting，如图7-22所示。



在项目 Hosting 中，添加一个类型为 Windows Service 的项目（Item），其名为 CalculateWinService。CalculateWinService 继承自基类 ServiceBase，通过重写 OnStart 和 OnStop 方法实现在 Windows Service 启动时实现对 WCF Service 的寄宿，ServiceHost 在 Windows Service 终止时被关闭。最后创建 App.config，并添加相应的 WCF 配置。

```
26 using System.ServiceModel;
27 using System.ServiceProcess;
28 using Artech.WindowsServiceHostingDemo.Services;
29 namespace Artech.WindowsServiceHostingDemo.Hosting
30 {
31     public partial class CalculateWinService : ServiceBase
32     {
33         private ServiceHost _serviceHost;
34
35         public CalculateWinService()
36         {
37             InitializeComponent();
38         }
39
40         protected override void OnStart(string[] args)
41         {
42             if (this._serviceHost != null)
43             {
```

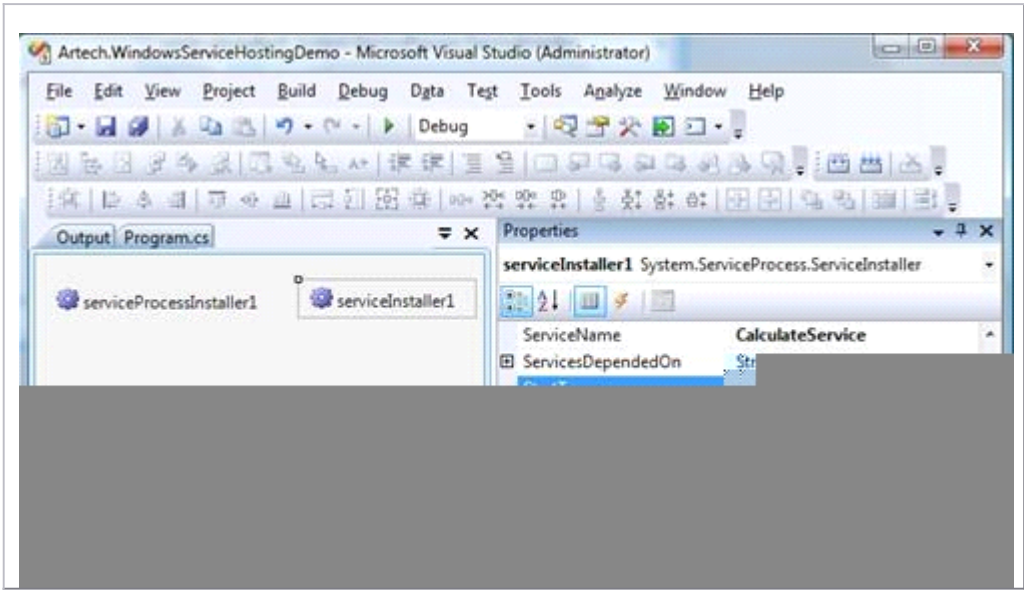
```

44         this._serviceHost.Close();
45     }
46     this._serviceHost = new ServiceHost
    (typeof(CalculatorService));
47     this._serviceHost.Open();
48 }
49
50     protected override void OnStop()
51     {
52         if (this._serviceHost != null)
53         {
54             this._serviceHost.Close();
55         }
56     }
57 }
58 }
59 <?xml version="1.0" encoding="utf-8" ?>
60 <configuration>
61     <system.serviceModel>
62         <services>
63             <service name="Artech.WindowsServiceHostingDemo.Services.
64                 CalculatorService">
65                 <endpoint address="net.tcp://127.0.0.1:9999/
66                     CalculatorService"
67                         binding="netTcpBinding" bindingConfiguration=""
contract=
68
69 "Artech.WindowsServiceHostingDemo.Contracts.ICalculator"/>
70             </service>
71         </services>
72     </system.serviceModel>
73 </configuration>

```

在 CalculateWinService 的设计模式下，点击右键选择“Add Installer”选项，会为你创建一个名为 ProjectInstaller 的安装项目（Item）。在 ProjectInstaller 中，会添加两个组件：serviceProcessIntaller1 和 serviceIntaller1，类型分别为 System.ServiceProcess.ServiceProcessInstaller 和 System.ServiceProcess.ServiceInstaller。这两个组件用于 Windows Service 的安装，前者基于整个 Service 进程，一个 Windows Service 项目（Project）具有一个 ServiceProcessInstaller，后者基于单个的 Windows Service。你可以通过属性窗口编辑它们的属性，ServiceInstaller 最重要的就是 StartType 和 ServiceName，用于控制 Windows Service 的启动方式。该属性具有3个选项：Automatic（机器启动时自动启动）、Manual（手

工启动)、Disabled (禁用)。ServiceName 用于指定在服务控制管理器中 Windows Service 显示的名称, 如图7-23所示。



(点击查看大图) 图7-23 可视化设置 Service Installer 的属性

### 步骤三 安装 Windows Service

Windows Services 项目经过编译后生成一个.exe 文件, 通过执行 installutil.exe 以命名行的形式安装定义在指定程序集中的 Windows Service, installutil.exe 位于 %windir%\Microsoft. NET\Framework\v2.0.50727目录下。