



WCF全面解析

◎蒋金楠 著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

WCF全面解析

【同业力荐】

我经历了COM时代，一直把Don Box的《COM本质论》奉为我的指路明灯。能把SOA机理和WCF这种特定厂商实现的技术讲得如《COM本质论》一样完美透彻的，那必属Artech这本经过自己深研、实践而著的心血结晶——《WCF全面解析》。如果你想成为SOA和WCF方面的专家，那么这本书就是你的最好法宝。想想你作为专家而获得的回报，那么你对这本书购买所付出的，简直是太值了。

——《走出软件作坊》作者 明源软件CTO 阿朱

首先，金楠是一位工作在一线的优秀的WCF技术人员，这符合我对阅读技术图书的第一个要求和期待。其次，金楠的写作文笔、专业责任也给人以充分信任，这在金楠的文字中读者可以体会。这本《WCF全面解析》全面剖析了构建WCF应用所需要的各方面技术，剥丝抽茧，由浅入深，也是我非常欣赏的技术讲述方式。我相信《WCF全面解析》一书是搞WCF朋友的案头必备。

——祝成科技与Boolean.com创始人 .NET技术专家 李建忠

知识全面、论述准确、逻辑严密是本书的特点。这是一本各层次开发人员都可以从中受益的书：对于初、中级开发人员，它可以帮助你获得WCF全方位的知识，系统地梳理WCF的知识结构，提升动手实践能力；对于高级开发人员，它既可以有效弥补你WCF相关知识中的盲点，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。

——资深架构师 曲春雨



策划编辑：张春雨
责任编辑：葛 娜
封面设计：李 玲

上架建议：程序设计>.Net

ISBN 978-7-121-16656-3



9 787121 166563 >

定价：168.00元（上、下册）



WCF全面解析

©蒋金楠 著

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

PDF

内 容 简 介

本书是作者多年潜心研究 WCF 技术的心血之作，也是这些年来从事 WCF 开发的经验总结。书如其名，《WCF 全面解析》涵盖了 WCF 几乎所有的知识点，并对其底层框架进行了“庖丁解牛”式的剖析，力求将 WCF 的整个运行机制完整而清晰地呈现在读者面前。

本书下册主要涉及一些所谓的“高级”话题，主要包括如何在分布式环境中处理异常（第 1 章）；元数据的导入与导出、发布与获取如何实现（第 2 章）；如何利用 WCF 对事务的支持将分布式事务引入服务（第 3 章）；如何利用并发与限流机制提高服务的吞吐量和可用性（第 4 章）；如何利用可靠会话机制确保消息的“使命必达”（第 5 章）；如何利用队列服务提供离线通信的支持（第 6 章）；第 7、8 章主要涉及安全的相关内容，包括传输安全、授权与审核；第 9 章全景展示 WCF 服务端和客户端的运行框架，以及在此基础上的所有扩展可能；最后一章为你带来 WCF 4.0 几个独立的新特性。

本书不仅适合尚未接触过 WCF，希望尽快入门并进行深入研究的开发人员使用，同样也适合对 WCF 有一定了解的开发设计人员和架构师阅读。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。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

WCF 全面解析：全 2 册 / 蒋金楠著. —北京：电子工业出版社，2012.4
ISBN 978-7-121-16656-3

I. ①W… II. ①蒋… III. ①网络服务器—程序设计 IV. ①TP368.5

中国版本图书馆 CIP 数据核字（2012）第 055701 号

策划编辑：张春雨

责任编辑：葛 娜

特约编辑：高洪霞

印 刷：三河市鑫金马印装有限公司
装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1092 1/16 印张：36.5 字数：882 千字

印 次：2012 年 4 月第 1 次印刷

印 数：3 000 册 定价：168.00 元

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

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

服务热线：（010）88258888。

目 录

第 1 章 异常处理 (Exception Handling)	1
1.1 WCF 基本异常处理模式	2
1.1.1 当异常从服务端抛出	2
1.1.2 异常细节的传播	5
1.1.3 自定义异常信息	8
1.2 错误消息与 FaultException 异常	19
1.2.1 从 SOAP Fault 说起	19
1.2.2 唯一可被传播的异常: FaultException	22
1.2.3 FaultException 异常和错误消息之间的转换	26
1.3 WCF 异常处理体系剖析	34
1.3.1 FaultFormatter	35
1.3.2 ServiceDebugBehavior 如何实现对异常细节传播	39
1.4 WCF 异常处理扩展	42
1.4.1 处理器 (ErrorHandler)	42
1.4.2 实例演示: 通过 WCF 扩展实现与 EntLib 的集成 (S105)	43
第 2 章 元数据 (Metadata)	55
2.1 WCF 元数据架构体系简介	56
2.1.1 WS-MEX	56
2.1.2 MetadataSection 与 MetadataSet	70
2.1.3 WCF 元数据架构模型	73
2.2 元数据的导出	74
2.2.1 MetadataExporter 和 WsdlExporter	74
2.2.2 WSDL 导出扩展和策略导出扩展	79
2.3 元数据的发布	81
2.3.1 元数据发布的实现者: ServiceMetadataBehavior	81

2.3.2	MEX 终结点有何不同	83
2.3.3	ServiceMetadataBehavior 是如何实现元数据发布的	85
2.4	元数据的获取和导入	97
2.4.1	自己动手实现元数据的获取	97
2.4.2	MetadaImporter 与元数据导入	102
第 3 章	事务 (Transaction)	108
3.1	WCF 需要怎样的事务控制	109
3.1.1	什么是事务	109
3.1.2	事务的显式控制	110
3.1.3	分布式事务应用场景	113
3.2	Windows 下的事务处理模型	114
3.2.1	事务模型中的三种角色	115
3.2.2	分布式事务是如何实现的	118
3.2.3	System.Transactions 事务	121
3.3	事务处理协议: OleTx 和 WS-AT	135
3.3.1	WS-Coordination	136
3.3.2	WS-AT	140
3.4	WCF 事务编程	142
3.4.1	通过服务契约决定事务流转的策略	142
3.4.2	通过绑定实施事务的流转	144
3.4.3	通过服务 (操作) 行为控制事务	153
3.4.4	实例演示: 创建事务型服务 (S301)	156
3.5	WCF 事务实现原理	166
3.5.1	TransactionFlowAttribute 行为	166
3.5.2	事务绑定	166
3.5.3	事务的自动登记 (Enlistment)	173
3.5.4	OleTx 提升 (OleTx Upgrade) 机制	174
第 4 章	并发与限流 (Concurrency and Throttling)	176
4.1	并发与实例上下文模式	177
4.1.1	同一个服务实例上下文同时处理多个服务调用请求	177
4.1.2	并发中的同步	180
4.1.3	并发与实例上下文模式	182
4.2	同步上下文与线程亲和性	196
4.2.1	倘若去除 ServiceBehaviorAttribute 的 UseSynchronizationContext 属性	196
4.2.2	什么是同步上下文 (SynchronizationContext)	197

4.2.3 WCF 中的同步上下文与线程亲和性	199
4.3 流量限制 (Throttling)	203
4.3.1 如何进行限流控制	203
4.3.2 WCF 限流控制是如何实现的	206
第 5 章 可靠会话 (Reliable Sessions)	210
5.1 可靠消息传输	211
5.1.1 从 TCP 对报文段的可靠交付机制说起	211
5.1.2 WS-RM 简介	213
5.2 编写可靠会话服务	220
5.2.1 实例演示: 通过 WCF 服务传输图片 (S501)	220
5.2.2 可靠会话绑定	234
5.3 可靠会话的实现原理	241
5.3.1 从信道层看可靠会话的实现	241
5.3.2 从传输协议的局限性和消息交换模式看可靠会话的实现	251
5.3.3 可靠会话最佳实践	254
第 6 章 队列服务 (Queued Service)	257
6.1 MSMQ 简介	258
6.1.1 MSMQ 能解决什么问题	258
6.1.2 MSMQ 的安装	259
6.1.3 消息队列	261
6.1.4 MSMQ 编程	263
6.2 从队列服务的终结点谈起	274
6.2.1 地址	274
6.2.2 绑定	276
6.2.3 契约	278
6.3 事务控制	279
6.3.1 MSMQ 事务模型	279
6.3.2 客户端事务	280
6.3.3 服务端事务	282
6.3.4 事务性批量接收	283
6.4 会话	288
6.4.1 客户端会话	288
6.4.2 服务端会话	292
6.5 错误处理	296
6.5.1 接收重试	296

6.5.2	接收错误处理	300
6.5.3	死信消息处理	301
6.5.4	日志 (Journaling) 与跟踪 (Tracing)	303
第 7 章	传输安全 (Transfer Security)	305
7.1	传输安全简介	306
7.1.1	分布式应用中的传输安全隐患	306
7.1.2	非对称加密 (Asymmetric Cryptography)	307
7.1.3	Transport 与 Message 安全模式	312
7.2	认证	318
7.2.1	认证与凭证 (User Credential)	318
7.2.2	绑定、安全模式与客户端凭证类型	323
7.2.3	服务认证	335
7.2.4	客户端认证	351
7.2.5	ServiceCredentials V.S. ClientCredentials	362
7.3	消息保护 (Message Protection)	366
7.3.1	消息的保护级别	366
7.3.2	签名与加密的实现	374
7.3.3	安全会话 (Secure Sessions)	380
第 8 章	授权与审核 (Authorization and Auditing)	386
8.1	身份 (Identity) 与安全主体 (Principal)	387
8.1.1	身份	387
8.1.2	安全主体	391
8.2	Windows 用户组授权	397
8.2.1	Windows 用户组授权与认证的关系	397
8.2.2	Windows 用户组授权编程	398
8.2.3	实例演示: 基于 Windows 用户组的声明式授权 (S801)	399
8.2.4	身份模拟 (Impersonation)	402
8.3	ASP.NET Roles 授权	409
8.3.1	ASP.NET Roles 提供程序	409
8.3.2	ASP.NET Roles 授权与认证的无关性	410
8.3.3	ASP.NET Roles 授权编程	411
8.3.4	实例演示: 不同认证方式下的 ASP.NET Roles 授权	413
8.3.5	实例演示: 通过 WCF 扩展实现授权 (S805)	418
8.4	自定义授权方式	423

8.4.1 通过自定义 AuthorizationPolicy 和 ServiceAuthorizationManager 创建安全主体	423
8.4.2 Claim 和 ClaimSet	426
8.4.3 自定义授权实现原理剖析	427
8.4.4 实例演示：通过自定义 AuthorizationPolicy 和 ServiceAuthorizationManager 实现授权 (S806)	428
8.5 安全审核 (Security Auditing)	434
8.5.1 ServiceSecurityAuditBehavior 服务行为	434
8.5.2 安全审核的实现	435
8.5.3 实例演示：如何实施安全审核	436
第 9 章 扩展 (Extension)	442
9.1 服务端架构体系的构建	443
9.1.1 再谈服务描述 (Service Description)	443
9.1.2 终结点分发器选择机制	446
9.1.3 信道分发器 (ChannelDispatcher)	448
9.1.4 终结点分发器 (EndpointDispatcher)	452
9.1.5 分发运行时 (DispatchRuntime)	453
9.1.6 分发操作 (DispatchOperation)	460
9.2 客户端架构体系的构建	465
9.2.1 创建 ChannelFactory<TChannel>	465
9.2.2 客户端运行时 (ClientRuntime)	467
9.2.3 客户端操作 (ClientOperation)	470
9.2.4 服务代理与服务调用	471
9.3 通过定义四种行为对 WCF 的扩展	474
9.3.1 WCF 四种类型的行为	474
9.3.2 行为方法的执行	476
9.3.3 实例演示：通过扩展确保语言文化一致性 (S901)	477
9.4 ServiceHost 对 WCF 的扩展	488
9.4.1 自定义 ServiceHost 的本质：对服务描述进行定制	488
9.4.2 自定义 ServiceHost 的创建者：ServiceHostFactory	491
9.4.3 实例演示：通过扩展实现基于 IoC 的服务实例的创建 (S903, S904)	493
第 10 章 WCF 4.0 新特性 (New Features in WCF 4.0)	503
10.1 简化开发体验	504
10.1.1 默认终结点	504

10.1.2	默认绑定配置.....	509
10.1.3	默认行为配置.....	510
10.1.4	标准终结点.....	513
10.1.5	无.svc 文件服务激活.....	514
10.2	路由服务(Routing Service).....	516
10.2.1	路由服务就是一个 WCF 服务.....	516
10.2.2	基于消息内容的路由策略.....	520
10.2.3	实例演示：如何使用路由服务（S1001）.....	527
10.2.4	其他路由特性.....	532
10.3	服务发现（Service Discovery）.....	534
10.3.1	WS-Discovery.....	534
10.3.2	可被发现的服务（Discoverable Service）.....	537
10.3.3	目标服务的探测和解析.....	544
10.3.4	实例演示：如何利用服务发现机制实现服务的 “动态”调用（S1002）.....	550
10.3.5	DynamicEndpoint.....	553
10.3.6	服务上/下线通知.....	555
10.3.7	发现代理（Discovery Proxy）.....	563
附录 A	实例列表.....	571
参考文献	573



第 1 章 异常处理

(Exception Handling)

本章旨在阐述如何采用 WCF 提供的编程模式有效地进行异常处理，以及如何在 WCF 异常处理的底层实现进行深入剖析。其中包括对异常的序列化与反序列化、异常的传播、异常的屏蔽等。



无论对于哪种编程语言,异常处理 (Exception Handling) 都是一项必不可少的元素。不同的语言提供了不同的异常处理方式,对于本书的读者群来说,最熟悉的莫过于 .NET 托管语言 (C#或者 VB.NET) 提供的异常处理编程模式。如果采用 C#或者 VB.NET 作为应用的编程语言,则直接通过 throw 语句抛出相应的异常,并通过 Try/Catch/Finally 这样的编程结构实现对异常的捕获和资源的清理。

对于一个非分布式的单进程应用 (整个应用运行在一个单一的进程下) 来说,异常处理无非就是简单地抛出异常和捕获异常而已。但是 WCF 解决的是相关系统之间的互联 (Connected System), 互联系统之间需要实现的是跨进程、跨机器以至于跨网络的交互,异常处理需要考虑的问题就不那么简单了。下面列出了在进行分布式应用的异常处理时需要解决和考虑的基本要素。

- 异常的封送 (Exception Marshaling): 服务端抛出的异常如何进行序列化以便能够传递到客户端。
- 敏感信息的屏蔽 (Sensitive Information Shielding): 抛出的异常往往包含一些敏感的信息,直接将服务操作执行过程抛出的异常信息原封不动地返回客户端,存在极大的安全隐患。
- 系统的集成和互操作 (System Integration and Interoperability): 基于不同厂商和技术平台系统之间的有效集成和互操作也给异常处理提出了新的要求,基于平台 A 的服务对异常的描述若想被基于平台 B 客户端理解,需要按照一种厂商中立 (Vendor Neutral) 的标准来规范对异常的描述。

1.1 WCF 基本异常处理模式

WCF 采用 .NET 托管语言 (C#和 VB.NET) 作为其主要的编程语言,这注定了其编程方式不可能很复杂,WCF 编程模式的简单性同样体现在异常处理上面。

1.1.1 当异常从服务端抛出

我个人倾向于将异常分为两种类型:应用异常 (Application Exception) 和基础结构异常 (Infrastructure Exception)。前者为应用级别,主要体现为执行某个服务操作的业务逻辑抛出的异常。后者则是业务无关的,由 WCF 本身的基础架构抛出,主要体现在对象的序列化、消息的处理、消息传输和消息的分发等操作抛出的异常。在这里我们更多地关注应用异常。

先来看看在不做任何异常处理的情况下,对于服务操作执行过程中抛出的异常,客户端会得到怎样的结果。我们通过实例的形式来演示这种场景。出于简单和易于理解考虑,我们照例沿用在上册中广泛使用的计算服务的例子,而且这个例子还会不断在本书的后续部分出

现。如图 1-1 所示，整个解决方案由三个项目组成，其中类库项目 `Service.Interface` 用于定义服务契约。服务类型定义在控制台程序 `Service` 中，同时它也用于对服务的寄宿。控制台程序 `Client` 模拟进行服务调用的客户端。`Service` 和 `Client` 均引用 `Service.Interface`。在以后的实例演示中，解决方案大都采用这样的结构。

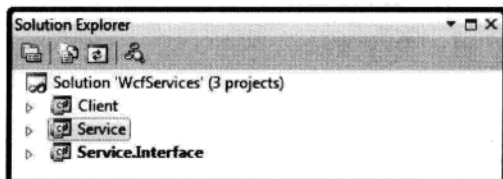


图 1-1 异常抛出实例解决方案结构

下面的代码片段表示服务契约 (`ICalculator`) 和服务类型 (`CalculatorService`) 的定义。在服务契约接口中，我们仅定义了唯一一个用于进行两个整数除法运算的方法 `Divide`。服务契约和服务类型分别定义在项目 `Contracts` 和 `Services` 中。

服务契约 (`ICalculator`):

```
using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface ICalculator
    {
        [OperationContract]
        int Divide(int x, int y);
    }
}
```

服务类型 (`CalculatorService`):

```
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Service
{
    public class CalculatorService : ICalculator
    {
        public int Divide(int x, int y)
        {
            return x / y;
        }
    }
}
```

接下来通过一个控制台应用程序 (`Service` 项目) 对上面定义的 `CalculatorService` 进行寄宿。下面是相关的配置和服务寄宿程序代码。

配置:

```
<configuration>
  <system.serviceModel>
    <services>
```



```

<service name="Artech.WcfServices.Service.CalculatorService">
  <endpoint address="http://127.0.0.1:3721/calculatorservice"
    binding="ws2007HttpBinding"
    contract="Artech.WcfServices.Service.Interface.ICalculator"/>
</service>
</services>
</system.serviceModel>
</configuration>

```

服务寄宿程序:

```

using System;
using System.ServiceModel;
namespace Artech.WcfServices.Service
{
    class Program
    {
        static void Main(string[] args)
        {
            using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
            {
                host.Open();
                Console.Read();
            }
        }
    }
}

```

最后在代表客户端的控制台应用程序 (Client 项目) 中编写调用 CalculatorService 服务的程序, 相关的配置和服务调用代码如下所示。为了让服务端在执行 Divide 操作的时候抛出异常, 特意将第二个参数设置为 0, 以便服务在进行除法运算的时候抛出 DivideByZeroException 异常。

配置:

```

<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="calculatorservice"
        address="http://127.0.0.1:3721/calculatorservice"
        binding="ws2007HttpBinding"
        contract="Artech.WcfServices.Service.Interface.ICalculator"/>
    </client>
  </system.serviceModel>
</configuration>

```

服务调用程序:

```

using System;
using System.ServiceModel;
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Clients
{
    class Program
    {
        static void Main(string[] args)
        {

```

```

using (ChannelFactory<ICalculator> channelFactory = new
    ChannelFactory<ICalculator>("calculatorservice"))
{
    ICalculator calculator = channelFactory.CreateChannel();
    using (calculator as IDisposable)
    {
        int result = calculator.Divide(1, 0);
    }
}
}
}

```

在启动服务后执行客户端服务调用程序，在客户端将会抛出如图 1-2 所示的类型为 `System.ServiceModel.FaultException` 的异常，提示“由于内部错误，服务器无法处理该请求”。(S101)

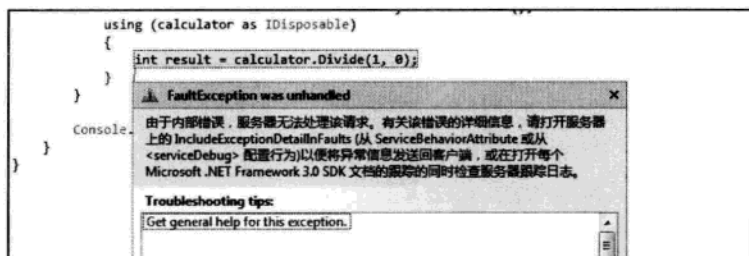


图 1-2 客户端捕获从服务端抛出的异常

上面的实例演示中体现了 WCF 在默认情况下的异常处理行为：对于服务端抛出的异常（这里指应用异常），客户端捕获到的总是一个具有相同消息的 `FaultException` 异常。客户端通过捕获到的异常根本无法确定服务端在执行服务操作的时候遇到的具体错误是什么。

WCF 如此设计主要基于安全的考虑。由于不能保证服务端直接抛出的异常不包含任何敏感信息，所以直接将服务端原始的异常信息暴露给客户端是不安全的。

1.1.2 异常细节的传播

在默认的情况下，如果异常在执行服务操作的过程中抛出，其真正的异常信息并不能被客户端捕获。实际上服务端具体的异常细节信息仅限于服务端可见，并不会传播到客户端。

然而，不论对于开发阶段的调试，还是维护阶段的纠错与排错，如果在客户端调用某个服务操作后能够很直接地获取到从服务端抛出异常的所有细节，无疑是一件很有价值的事情。可以通过在服务上应用 `System.ServiceModel.Description.ServiceDebugBehavior` 服务行为，并将其 `IncludeExceptionDetailInFaults` 属性设置为 `True`，将服务端的异常细节暴露出来。可以通过下面的配置来应用这个服务行为 (S102)。

```

<configuration>
  <system.serviceModel>

```

```

<behaviors>
  <serviceBehaviors>
    <behavior name="serviceDebuBehavior">
      <serviceDebug includeExceptionDetailInFaults="true" />
    </behavior>
  </serviceBehaviors>
</behaviors>
<services>
  <service behaviorConfiguration="serviceDebuBehavior" ...>
    ...
  </service>
</services>
</system.serviceModel>
</configuration>

```

除了通过配置的方式来应用 `ServiceDebugBehavior` 服务行为，还可以采用声明的方式达到一样的效果。具体来说，就是按照如下的方式直接在服务类型上应用 `ServiceBehavior` `Attribute` 特性并将 `IncludeExceptionDetailInFaults` 属性设置为 `True` 即可。

```

[ServiceBehavior(IncludeExceptionDetailInFaults = true)]
public class CalculatorService : ICalculator
{
    //省略服务成员
}

```

如果采用任何一种方式将 `ServiceDebugBehavior` 行为应用到上面演示实例中的服务类型 `CalculatorService` 上，并将 `IncludeExceptionDetailInFaults` 属性设置为 `True`，客户端就会得到如图 1-3 所示的服务端异常的原始错误消息。

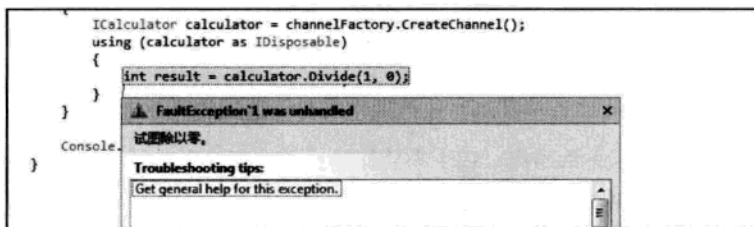


图 1-3 客户端捕获到具有细节信息的异常

从图 1-3 中可以看出客户端捕获到的实际上是一个泛型的 `System.ServiceModel.FaultException<TDetail>` 异常。`FaultException<TDetail>` 继承自 `FaultException`，这两种典型的异常类型在 WCF 异常处理中占有重要的地位，在本章后续章节中还会重点讲述，在这里先做一点简单的介绍。

对于所有从服务端抛出的异常，只有类型为 `FaultException`（或其子类）的异常才能直接被序列化并最终通过消息返回给客户端。`FaultException<TDetail>` 允许将错误细节定义在一个对象上，而泛型参数 `TDetail` 表示这个对象的类型。下面给出了 `FaultException<TDetail>` 的简单定义，可以直接通过指定错误细节对象来创建一个 `FaultException<TDetail>` 对象，而指定的这个错误细节对象对应于只读属性 `Detail`。

```
[Serializable]
public class FaultException<TDetail> : FaultException
{
    // 其他成员
    public FaultException(TDetail detail);
    public TDetail Detail { get; }
}
```

对于前面演示的实例来说，客户端捕获的异常类型实际上是 `FaultException<ExceptionDetail>`，也就是说，其具体的泛型类型参数为 `System.ServiceModel.ExceptionDetail`。我们不妨来看看 `ExceptionDetail` 类型的定义。

```
[DataContract]
public class ExceptionDetail
{
    // 其他成员
    public ExceptionDetail(Exception exception);

    [DataMember]
    public string HelpLink { get; private set; }
    [DataMember]
    public ExceptionDetail InnerException { get; private set; }
    [DataMember]
    public string Message { get; private set; }
    [DataMember]
    public string StackTrace { get; private set; }
    [DataMember]
    public string Type { get; private set; }
}
```

`ExceptionDetail` 是一个应用了 `DataContractAttribute` 特性的数据契约，意味着 `ExceptionDetail` 是一个可以被序列化的对象。再看其具体的属性成员列表，我想对于很多读者来说肯定有一种似曾相识的感觉。是不是觉得 `ExceptionDetail` 具有与 `System.Exception` 类似的属性成员？实际上 `ExceptionDetail` 是 WCF 专门设计出来用于封装服务端抛出的异常信息的，属性 `HelpLink`、`InnerException` 和 `StackTrace` 对应 `Exception` 的同名属性，而属性 `Type` 则表示异常的类型。

也就是说，对于应用了开启 `IncludeExceptionDetailInFaults` 开关的 `ServiceDebugBehavior` 服务行为的服务来说，在服务操作执行过程中抛出的异常的详细信息可以通过捕获的 `FaultException<ExceptionDetail>` 异常中的 `Detail` 属性获取。比如在下面的代码中，我们修改了客户端的代码，将具体的错误信息输出到控制台上。

```
using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorservice"))
{
    ICalculator calculator = channelFactory.CreateChannel();
    using (calculator as IDisposable)
    {
        try
        {
            int result = calculator.Divide(1, 0);
        }
    }
}
```

```

        catch (FaultException<ExceptionDetail> ex)
        {
            Console.WriteLine("Message:{0}", ex.Detail.Message);
            Console.WriteLine("Type:{0}", ex.Detail.Type);
            Console.WriteLine("StackTrace:{0}", ex.Detail.StackTrace);
            Console.WriteLine("HelpLink:{0}", ex.Detail.HelpLink);
            (calculator as ICommunicationObject).Abort();
        }
    }
}

```

输出结果:

Message: 试图除以零。

Type: System.DivideByZeroException

StackTrace: 在 Artech.WcfServices.Services.CalculatorService.Divide(Int32 x, Int32 y)

位置 D:\Demos\Artech.WcfServices\Services\CalculatorService.cs:行号 13

在 SyncInvokeDivide(Object , Object[] , Object[])

在 System.ServiceModel.Dispatcher.SyncMethodInvoker.Invoke(Object instance, Object[] inputs, Object[]& outputs)

在 System.ServiceModel.Dispatcher.DispatchOperationRuntime.InvokeBegin(MessageRpc& rpc)

在 System.ServiceModel.Dispatcher.ImmutableDispatchRuntime.ProcessMessage5(MessageRpc& rpc)

在 System.ServiceModel.Dispatcher.ImmutableDispatchRuntime.ProcessMessage4(MessageRpc& rpc)

在 System.ServiceModel.Dispatcher.ImmutableDispatchRuntime.ProcessMessage3(MessageRpc& rpc)

在 System.ServiceModel.Dispatcher.ImmutableDispatchRuntime.ProcessMessage2(MessageRpc& rpc)

在 System.ServiceModel.Dispatcher.ImmutableDispatchRuntime.ProcessMessage1(MessageRpc& rpc)

在 System.ServiceModel.Dispatcher.MessageRpc.Process(Boolean isOperationContextSet)

HelpLink:

对于应该在何种情况下使用服务行为 `ServiceDebugBehavior` 来将服务端抛出的异常暴露给客户端, 我需要重申一遍: 由于会导致敏感信息泄露的潜在危险, 一般我们仅在调试的时候才会开启该属性。实际上从这个服务行为的命名也可以看出, `ServiceDebugBehavior` 就是用于调试服务的行为。

1.1.3 自定义异常信息

在默认的情况下, 服务端在执行某个服务操作时抛出的异常 (在这里指非 `FaultException` 异常), 其相关的错误信息仅限于服务端可见, 并不会被 WCF 传递到客户端。而通过应用服务行为 `ServiceDebugBehavior`, 则可以将服务端抛出的异常原原本本地传播到客户端。这两种方式体现了两种极端的异常传播机制: 对于基于服务操作执行过程中抛出的异常的错误细节, 要么完全对客户端屏蔽, 要么全部暴露于客户端。

我们往往需要一种折中的异常传播机制：自定义服务端异常信息。这样既可以让客户端得到一个易于理解的错误信息，又在一定程度上避免了一些敏感信息的泄露。

1. 通过 FaultException 直接指定错误信息

对于在服务操作执行中抛出的异常，如果服务的定义者仅仅希望服务的调用者得到一段自定义的错误消息文本（字符串），只需要在服务操作中直接抛出一个 `FaultException` 异常，并将错误信息指定为异常的消息即可。在下面的代码中，`CalcuatorService` 的 `Divide` 对作为被除数的第二参数进行了验证，如果为零，则创建一个 `FaultException`，并指定错误信息（“被除数 `y` 不能为零!”）。

```
public class CalculatorService : ICalculator
{
    public int Divide(int x, int y)
    {
        if (0 == y)
        {
            throw new FaultException("被除数 y 不能为零!");
        }
        return x / y;
    }
}
```

客户端在调用该服务操作的时候，如果传入零作为被除数，将会直接捕获服务端抛出的这个异常（实际上这其中经历了异常对象的序列化、消息交换及异常对象的反序列化等一系列的操作）。那么客户端就可以采用如下的方式来得到指定的错误信息。（S103）

```
using (ChannelFactory<ICalculator> channelFactory =
    new ChannelFactory<ICalculator>("calculatorservice"))
{
    ICalculator calculator = channelFactory.CreateChannel();
    using (calculator as IDisposable)
    {
        try
        {
            int result = calculator.Divide(1, 0);
        }
        catch (FaultException ex)
        {
            Console.WriteLine(ex.Message);
            (calculator as ICommunicationObject).Abort();
        }
    }
}
```

输出结果：

被除数 `y` 不能为零!

虽然在很多情况下，在服务端指定服务操作的过程中，直接抛出含有自定义错误信息的 `FaultException` 异常就能让客户知道遇到的具体错误并进行必要的排错和纠错，但是我们

还是倾向于直接定义一个类型来描述异常信息。服务端根据具体的异常场景创建相应的错误类型对象，并基于该对象创建并抛出上面提到的 `FaultException<TDetail>` 异常。在这个过程中，还涉及错误契约 (Fault Contract) 的概念。

2. 通过 `FaultException<TDetail>` 采用自定义类型封装错误

用于封装异常细节的对象最终需要通过消息交换的方式从服务端传播到客户端，所以该对象必须是一个可序列化的对象。可以在该类型上应用 `SerializableAttribute` 特性，或者将其定义成数据契约。

以上面演示的计算服务为例，现在我们需要定义一个类型来描述运算错误信息。将该类型命名为 `CalculationError`，并将其定义成如下一个数据契约。我们仅仅定义了 `Operation` 和 `Message` 两个数据成员。前者表示导致异常的相应的运算操作，后者表示具体的错误消息。

```
[DataContract(Namespace = "http://www.artech.com/")]
public class CalculationError
{
    public CalculationError(string operation, string message)
    {
        this.Operation = operation;
        this.Message = message;
    }
    [DataMember]
    public string Operation{ get; set; }
    [DataMember]
    public string Message{ get; set; }
}
```

照理说我们已经正确定义了错误细节类型 `CalculationError`，在 `CalculatorService` 的 `Divide` 操作中就可以按照如下的方式直接抛出一个 `FaultException<CalculationError>`，并将创建一个 `CalculationError` 对象作为该异常对象的细节（通过 `Detail` 属性表示）。

```
public class CalculatorService : ICalculator
{
    public int Divide(int x, int y)
    {
        if (0 == y)
        {
            var error = new CalculationError("Divide", "被除数 y 不能为零!");
            throw new FaultException<CalculationError>(error, error.Message);
        }
        return x / y;
    }
}
```

客户端服务调用相关的异常处理做如下相应的修改。我们捕获 `FaultException<CalculationError>` 类型的异常，并打印出运算错误的操作名称和错误信息。

```

using (ChannelFactory<ICalculator> channelFactory = new ChannelFactory
<ICalculator>("calculatorservice"))
{
    ICalculator calculator = channelFactory.CreateChannel();
    using (calculator as IDisposable)
    {
        try
        {
            int result = calculator.Divide(1, 0);
        }
        catch (FaultException<CalculationError> ex)
        {
            Console.WriteLine("运算错误");
            Console.WriteLine("运算操作: {0}", ex.Detail.Operation);
            Console.WriteLine("错误消息: {0}", ex.Detail.Message);
            (calculator as ICommunicationObject).Abort();
        }
    }
}

```

但是客户端程序并不能按照我们事先预想的那样正常运行，而是会抛出如图 1-4 所示的未被处理的 `FaultException` 异常，而不是我们试图捕获的异常类型 `FaultException<CalculationError>`。

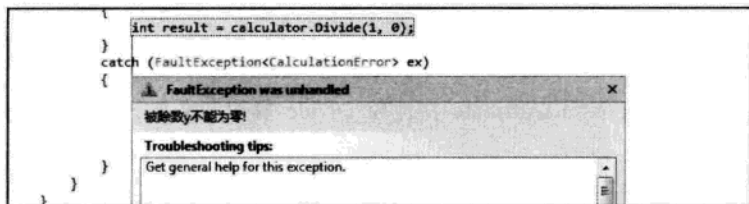


图 1-4 客户端不能正常捕获 `FaultException<CalculationError>` 异常

3. 错误契约 (Fault Contract)

客户端能够捕获到服务端抛出的 `FaultException<TDetail>` 异常，并不是说客户端捕获的异常就是服务端抛出的那个异常。这个过程经历了对异常的序列化和反序列化的过程。错误细节对象被序列化成 XML 并通过 SOAP 消息的形式返回给客户端，客户端需要通过反序列化以重建错误细节对象。而反序列化的前提是需要预先知道其类型。而错误契约 (Fault Contract) 帮助你作为错误细节对象的类型确定下来。

WCF 通过 `System.ServiceModel.FaultContractAttribute` 特性定义错误契约。由于错误契约是基于操作级别的，所以该特性应用于服务契约类型的操作契约方法成员上。下面的代码给出了 `FaultContractAttribute` 主要的属性成员定义。

```

[AttributeUsage(AttributeTargets.Method, AllowMultiple = true, Inherited =
false)]
public sealed class FaultContractAttribute : Attribute
{

```

```

public FaultContractAttribute(Type detailType);
public string Action { get; set; }
public Type DetailType { get; }
public bool HasProtectionLevel { get; }
public string Name { get; set; }
public string Namespace { get; set; }
public ProtectionLevel ProtectionLevel { get; set; }
}

```

FaultContractAttribute 的 6 个属性分别具有如下的含义。

- **Action:** 错误消息<Action>报头的值。如果 Action 属性没有被显式指定, 那么它将按照下面的规则进行指定: {服务契约命名空间}/{服务契约名称}/{操作契约名称}{细节类型名称}Fault。
- **DetailType:** 用于封装错误信息的错误细节类型(比如我们前面定义的 CalculationError)。
- **Name 和 Namespace:** 错误细节对象被序列化成 XML 元素的名称和命名空间。如果未做显式设置, WCF 将会使用 DetailType 对应的数据契约名称和命名空间作为这两个属性的值。
- **HasProtectionLevel 和 ProtectionLevel:** 这两个属性涉及保护级别, 属于安全 (Security) 的问题。本书的第 7 章“传输安全 (Transfer Security)”会对其进行详细讲述, 在这里就不多做介绍了。

再次回到之前演示的实例。为了确保错误细节对象能够被正常地序列化和反序列化, 需要按照如下的方式通过 FaultContractAttribute 特性为 Divide 操作定义基于 CalculationError 类型的错误契约。(S104)

```

[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract]
    [FaultContract(typeof(CalculationError))]
    int Divide(int x, int y);
}

```

当客户端调用 CalculatorService 的 Divide 操作执行除法运算, 并传入零作为被除数时, 服务端将会抛出 FaultException<CalculationError>异常。CalculationError 对象最终将被序列化后置于生成的错误消息中。如果采用的消息版本是 Soap12Addressing10 (SOAP 1.2+ WS-Addressing 1.0), 最终会生成如下所示的错误消息。

```

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">
      http://www.artech.com/ICalculator/DivideCalculationErrorFault
    </a:Action>
    <a:RelatesTo>
      urn:uuid:3498ba2d-edd0-4d3b-ba4a-9b35327b5fa3
    </a:RelatesTo>
  </s:Header>

```

```

<s:Body>
  <s:Fault>
    <s:Code>
      <s:Value>s:Sender</s:Value>
    </s:Code>
    <s:Reason>
      <s:Text xml:lang="zh-CN">被除数 y 不能为零!</s:Text>
    </s:Reason>
    <s:Detail>
      <CalculationError xmlns="http://www.artech.com/"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <Message>被除数 y 不能为零!</Message>
        <Operation>Divide</Operation>
      </CalculationError>
    </s:Detail>
  </s:Fault>
</s:Body>
</s:Envelope>

```

错误契约是服务元数据 (Metadata) 的一部分。当服务元数据通过 WSDL 的形式被发布后, 错误契约作为操作描述的一部分被写入。下面的 XML 片段代表了前面定义的服务契约 ICalculator 对应的 WSDL。在 <types> 节点下定义了针对 CalculationError 类型的同名 ComplexType (XSD)。定义了针对这个 ComplexType 的消息 (名称为 ICalculator_Divide_CalculationErrorFault_FaultMessage)。最终在代表 Divide 操作的节点下定义了代表错误消息的 <fault> 元素。该 <fault> 元素代表定义在操作上的错误契约, 通过 FaultContractAttribute 定义的 Name 和 Action 对应同名的属性, 而 message 属性引用针对错误细节类型 XSD 的消息。

```

<wsdl:definitions name="CalculatorService"
  targetNamespace="http://www.artech.com/" >
<wsdl:import namespace="http://tempuri.org/"
  location="http://127.0.0.1:3721/calculator-service/mex?wsdl=wsdl0"/>
<wsdl:types>
  <xs:schema elementFormDefault="qualified"
    targetNamespace="http://www.artech.com/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://www.artech.com/">
    ...
    <xs:complexType name="CalculationError">
      <xs:sequence>
        <xs:element minOccurs="0" name="Message" nillable="true"
          type="xs:string"/>
        <xs:element minOccurs="0" name="Operation" nillable="true"
          type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
    <xs:element name="CalculationError" nillable="true"
      type="tns:CalculationError"/>
  </xs:schema>
</wsdl:types>
  ...
<wsdl:message name="ICalculator_Divide_CalculationErrorFault_FaultMessage">
  <wsdl:part name="detail" element="tns:CalculationError"/>
</wsdl:message>
<wsdl:portType name="ICalculator">
  <wsdl:operation name="Divide">

```

```

<wsdl:input wsaw:Action="http://www.artech.com/ICalculator/Divide"
            message="tns:ICalculator_Divide_InputMessage"/>
<wsdl:output wsaw:Action="http://www.artech.com/ICalculator/DivideResponse"
            message="tns:ICalculator_Divide_OutputMessage"/>
<wsdl:fault
wsaw:Action="http://www.artech.com/ICalculator/DivideCalculationErrorFault"
    name="CalculationErrorFault"
    message="tns:ICalculator_Divide_CalculationErrorFault_FaultMessage"/>
</wsdl:operation>
</wsdl:portType>
...
</wsdl:definitions>

```

对于错误契约的应用,还有一点需要特别说明。不仅仅是在将自定义的错误细节类型(比如我们定义的 `CalculationError`)应用到服务契约相应操作上时才需要显式地在操作方法上应用 `FaultContractAttribute` 特性,对于一些基元类型(比如 `Int32`, `String` 等),也需要这样做。对于计算服务的例子,如果服务端试图抛出一个 `FaultException<string>` 异常,则客户端最后捕获到的仅仅是一个 `FaultException` 异常,而非 `FaultException<string>` 异常。

```

public class CalculatorService : ICalculator
{
    public int Divide(int x, int y)
    {
        if (0 == y)
        {
            throw new FaultException<string>("被除数 y 不能为零!", "被除数 y 不能为零!");
        }
        return x / y;
    }
}

```

在这种情况下,我们依然需要按照如下的方式在相应的操作上面,通过应用 `FaultContractAttribute` 特性显式地将 `String` 类型作为其错误细节类型。

```

[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract]
    [FaultContract(typeof(string))]
    int Divide(int x, int y);
}

```

从 `FaultContractAttribute` 的定义可以看出,该特性可以在同一个目标对象上面多次应用 (`AllowMultiple = true`)。这也很好理解,因为对于同一个服务操作,需要面临不同的异常场景,在不同的情况下,需要抛出基于不同错误细节类型的 `FaultException<TDetail>` 异常。

```

[AttributeUsage(AttributeTargets.Method, AllowMultiple = true, Inherited = false)]
public sealed class FaultContractAttribute : Attribute
{
    //省略成员
}

```

但是如果你在同一操作方法上面应用了多个 `FaultContractAttribute` 特性,则需要遵循

一系列的规则，下面就来逐条介绍它们。

(1) 多次声明相同的错误细节类型

比如在下面的代码中，通过 `FaultContractAttribute` 特性对同一个错误细节类型 `CalculationError` 进行了两次设置。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract]
    [FaultContract(typeof(CalculationError))]
    [FaultContract(typeof(CalculationError))]
    int Divide(int x, int y);
}
```

在对实现了该契约接口的服务进行寄宿的时候会抛出如图 1-5 所示的 `InvalidOperationException` 异常，提示在相同的操作中声明了多个类型一致的错误契约。

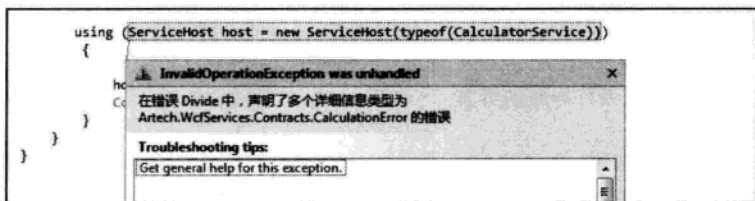


图 1-5 多次声明相同的错误细节类型导致的异常

但是在应用 `FaultContractAttribute` 特性指定相同错误细节类型的同时，指定不同的 `Name` 或者 `Namespace` 却是允许的。比如下面的代码中，在两个 `FaultContractAttribute` 特性中，同样是指定了相同的错误细节类型 `CalculationError`，由于我们为之指定了不同的 `Name`，因此在寄宿服务的时候将不会有上述异常的发生。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract]
    [FaultContract(typeof(CalculationError), Name = "CalculationError")]
    [FaultContract(typeof(CalculationError), Name = "CalculationException")]
    int Divide(int x, int y);
}
```

(2) 多次声明包含相同名称+命名空间的不同错误细节类型

多次声明的错误细节类型虽然不同，但是如果我们为其指定相同的 `Name` 和 `Namespace`，依然是不允许的。比如下面的代码中，通过 `FaultContractAttribute` 特性为 `Divide` 操作指定了两个不同的错误细节类型（`CalculationError` 和 `CalculationException`），但是设置的名称和命名空间都是相同的。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
```



```

{
    [OperationContract]
    [FaultContract(typeof(CalculationError),
        Name = "CalculationError",
        Namespace = "http://www.artech.com/")]
    [FaultContract(typeof(CalculationException),
        Name = "CalculationError",
        Namespace = "http://www.artech.com/")]
    int Divide(int x, int y);
}

```

对于这种情况，在服务寄宿的时候，依然会和上面一样抛出一个 `InvalidOperationException` 异常，提示在同一个命名空间下声明了多个同名的错误契约。



图 1-6 多次声明具有相同有效名称导致的异常

(3) 多次声明包含相同名称+命名空间（通过数据契约形式定义）的不同错误细节类型

还有另一种情况：虽然多次声明的是不同的错误细节类型，但是通过 `DataContractAttribute` 特性定义它们的时候，指定了相同的名称和命名空间。如果我们通过 `FaultContractAttribute` 特性将它们应用到同一个操作上面，又会出现怎样的问题呢？比如在下面的代码中，定义了两个不同错误细节类型（`CalculationError` 和 `CalculationFault`），它们具有相同的数据契约名称（`CalculationError`）和命名空间（`http://www.artech.com/`）。

```

[DataContractAttribute(Namespace = "http://www.artech.com/",
    Name = "CalculationError")]
public class CalculationError
{
    [DataMember]
    public string Operation
    { get; set; }
    [DataMember]
    public string Message
    { get; set; }
}
[DataContractAttribute(Namespace = "http://www.artech.com/",
    Name = "CalculationError")]
public class CalculationFault
{
    [DataMember]
    public string Fault
    { get; set; }
}

```

如果通过下面的方式通过 `FaultContractAttribute` 特性将这两个类型应用到同一个服务操

作上面（没有显式地指定 Name 和 Namespace 属性），服务寄宿不会出什么问题，客户端的方法调用也能正常运行。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract]
    [FaultContract(typeof(CalculationError))]
    [FaultContract(typeof(CalculationFault))]
    int Divide(int x, int y);
}
```

但是当我们试图通过 HTTP-GET 或者标准的 MEX 终结点获取以 WSDL 表示的服务元数据的时候就会出现这个问题。至于为什么会这样的问题，大体可以这样来理解：每个定义的错误契约都有一个以命名空间名称组成的唯一标识，如果两个错误契约具有相同的标识，就会导致冲突。

数据契约是对数据结构的一种描述。如果两个数据契约是等效的，不管其具体的托管类型是什么，WCF 在遇到上述情况的时候，都会自动识别并忽略其中一个，从而保证元数据能够正确产生。比如，如果将 CalculationFault 进行如下的改写（通过 DataMemberAttribute 特性使定义在不同数据契约中的数据成员具有相同的名称），服务的元数据就能够被正常地获得了。

```
[DataContractAttribute(Namespace = "http://www.artech.com/",
Name = "CalculationError")]
public class CalculationFault
{
    [DataMember(Name = "Operation")]
    public string OperationName
    { get; set; }
    [DataMember(Name = "Message")]
    public string Fault
    { get; set; }
}
```

4. 通过 XmlSerializer 对错误细节对象进行序列化

WCF 通过序列化器（Serializer）实现对托管对象的序列化和反序列化。WCF 提供了 DataContractSerializer 和 XmlSerializer 两种主要的序列化器。WCF 采用的默认序列化器是 DataContractSerializer。但是有的时候，需要显式地控制某个服务或者服务的某个操作的序列化行为使用 XmlSerializer。

可以通过使用 XmlSerializerFormatAttribute 特性选择 XmlSerializer 完成序列化和反序列化工作。但是在默认的情况下，XmlSerializerFormatAttribute 特性仅仅控制操作的参数和返回值的序列化行为，而不能控制错误细节对象的序列化行为。可以通过 SupportFaults 属性来显式地选择 XmlSerializer 作为错误细节对象的序列化器。

在下面的代码中，我们将 XmlSerializerFormatAttribute 特性应用在服务契约的 Divide 操

作上面, 并将 `SupportFaults` 属性设为 `True`。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract]
    [FaultContract(typeof(CalculationError), Name = "CalculationError")]
    [XmlSerializerFormat(SupportFaults = true)]
    int Divide(int x, int y);
}
```

对于 `Divide` 操作, WCF 将会采用 `XmlSerializer` 同时作为参数、返回值和错误细节对象的序列化器。比如在这个时候, 采用下面的形式对 `CalculationError` 进行重新定义。

```
[Serializable]
public class CalculationError
{
    [XmlAttributeAttribute("op")]
    public string Operation
    { get; set; }
    [XmlElement("Error")]
    public string Message
    { get; set; }
}
```

在被零除而抛出异常的情况下, WCF 将会生成如下一个错误消息, 其中高亮部分代表 `CalculationError` 对象序列化后的 XML。可见 `CalculationError` 对象是按照传统的 XML 序列化规则方式被序列化的。

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">
      http://www.artech.com/ICalculator/DivideCalculationError
    </a:Action>
    <a:RelatesTo>urn:uuid:7b01995b-9f81-4a08-9fa2-c5ef8c7cacc</a:RelatesTo>
  </s:Header>
  <s:Body>
    <s:Fault>
      <s:Code>
        <s:Value>s:Sender</s:Value>
      </s:Code>
      <s:Reason>
        <s:Text xml:lang="zh-CN">被除数 y 不能为零!!</s:Text>
      </s:Reason>
      <s:Detail>
        <CalculationError op="Divide" xmlns="http://www.artech.com/"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
          <Error>被除数 y 不能为零!!</Error>
        </CalculationError>
      </s:Detail>
    </s:Fault>
  </s:Body>
</s:Envelope>
```

1.2 错误消息与 FaultException 异常

在上一节中，主要站在最终开发者的角度对 WCF 的异常处理编程模式进行了介绍。接下来，需要将我们的目光转移到 WCF 框架内部，深入剖析整个 WCF 异常处理流程。从编程的角度讲，错误信息的载体是 FaultException 异常。而从消息交换的角度来讲，错误信息则是通过错误消息（Fault Message）来承载的。所以说 FaultException 异常和错误消息是错误信息的两种不同的载体。

1.2.1 从 SOAP Fault 说起

消息不但承载着正常服务调用的请求和回复，在出现异常时，消息依然是错误信息的载体。消息不但承载着正常服务调用的请求和回复，在出现异常时，消息依然是错误信息的载体。对于基于 SOAP 的消息交换来说，异常信息被封装在专门的错误消息（Fault Message）中，我们将错误消息中封装错误信息的 <Fault> 元素称为 SOAPFault。由于异常在消息交换中通过错误消息承载，所以很有必要重申一下 SOAPFault 的相关规范。下面首先来看一个具有完整结构的错误消息。

```
<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
    <a:Action s:mustUnderstand="1">
      http://www.artech.com/ICalculator/DivideCalculationErrorFault
    </a:Action>
  </s:Header>
  <s:Body>
    <s:Fault>
      <s:Code>
        <s:Value>s:Sender</s:Value>
        <s:Subcode>
          <s:Value xmlns:a="http://www.artech.com/">a:CalculationError
          </s:Value>
        </s:Subcode>
      </s:Code>
      <s:Reason>
        <s:Text xml:lang="zh-CN">被除数 y 不能为零!</s:Text>
      </s:Reason>
      <s:Node>http://http://www.artech.com/calculationcenter</s:Node>
      <s:Role>http://http://www.artech.com/calculatorservice</s:Role>
      <s:Detail>
        <CalculationError xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://www.artech.com/">
          <Message>被除数 y 不能为零!</Message>
          <Operation>Divide</Operation>
        </CalculationError>
      </s:Detail>
    </s:Fault>
  </s:Body>
</s:Envelope>
```

这是一个结构比较完整的错误消息，它的主体 (Body) 部分包含了构成 SOAP Fault 所有类型的元素。接下来就在这个错误消息的基础上介绍 SOAP 1.2 规范下的 SOAP Fault 具有怎样的结构。SOAP Fault 作为错误 SOAP 消息的主体，用于承载错误相关的信息。对于具体的 SOAP Fault 元素，做了如下几点规范。

- 元素名称必须为 Fault。
- 元素命名空间必须为 http://www.w3.org/2003/05/soap-envelope。
- 元素包含如下子元素：
 - 一个必需的 Code 元素表示错误代码 (SOAP 节点)；
 - 一个必需的 Reason 元素表示出错的原因；
 - 一个可选的 Node 元素表示导致出错的 SOAP 节点 (SOAP Node)；
 - 一个可选的 Role 元素表示 SOAP 节点对应的角色；
 - 一个可选的 Detail 表示对错误的详细描述。

1. Fault Code 元素

SOAP Fault 的 Code 元素，是一个用于表示错误类型的代码。该错误代码可以大致看做对错误的一种分类。SOAP 1.2 对 Code 元素的格式做了如下的规范。

- 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。
 - 一个必需的 Value 元素用于定义错误代码；
 - 一个可选的 SubCode 元素用于定义错误子代码。

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 1-1 列出了所有的可选枚举值。

表 1-1 FaultCodeEnum 枚举值

枚举值	含义
VersionMismatch	命名空间或者名称和规定的 SOAP 规范不匹配
MustUnderstand	目标 SOAP 节点不能理解并处理 mustUnderstand 属性为 “true” 或者 “1” 的 SOAP 报头
DataEncodingUnknown	SOAP 报头或者主体的数据编码方式不被目标 SOAP 节点支持
Sender	消息格式合法或者缺少必要的信息
Receiver	SOAP 节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。
 - 必需的 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做该元素的值。
 - 可选的 Subcode 元素。

可见 Fault Code 是一种具有层级关系的（Hierarchical）的结构（Code 的具有一个 Code 结构的 SubCode）。上面给出的错误消息，就具有一个有两层结构的 Fault Code。

```
<s:Code>
  <s:Value>s:Sender</s:Value>
  <s:Subcode>
    <s:Value xmlns:a="http://www.artech.com/">a:CalculationError</s:Value>
  </s:Subcode>
</s:Code>
```

2. Fault Reason 元素

对于一个错误消息，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 元素用于表示导致错误的原因。SOAP 1.2 对 Reason 元素的格式做了如下的规范。

- 元素名称必须为“Reason”，命名空间名称为 http://www.w3.org/2003/05/soap-envelope”。
- 包含一个或者多个 Text 元素用于描述错误。

对于上面给出的错误消息，具有如下一个 Fault Reason 元素。Text 元素中的 lang 属性表示相应的语言文化。也就是说可通过该属性指定基于不同语言文化的文字用于描绘错误的原因。

```
<s:Reason>
  <s:Text xml:lang="zh-CN">被除数 y 不能为零!</s:Text>
</s: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节点，也可能发生在中间节点，因此为了使 SOAP 错误消息的接收者能够判断导致错误的 SOAP 节点类型，在生成错误消息的时候，可以通过 Node 元素指定节点的类型。SOAP 1.2 对 Node 元素做如下的规范。

- 元素名称必须为“Node”，命名空间名称为“http://www.w3.org/2003/05/soap-envelope”。
- 元素值类型为“xs:anyURI”，即通过 URI 表示的 SOAP 节点（参考 SOAP 报头的 Role 属性）。

在上面给出的错误消息中，我将 Fault Node 指定为 http://http://www.artech.com/calculationcenter。

4. Fault Role 元素

SOAP 节点处理 SOAP 消息时担当着不同的角色。SOAP Fault 的 Role 元素用于表示导致错误的 SOAP 节点对应的角色。SOAP 1.2 对 Role 元素的格式做了如下的规范。

- 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 节点对应的角色（参考 SOAP 报头的 Role 属性）。

在上面给出的错误消息中，我将 Fault Role 指定为 http://http://www.artech.com/calculatorservice。

5. Fault Detail 元素

SOAP 错误消息的接收者除了需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。这样的描述就可以通过 Detail 元素来表示。SOAP 1.2 对 Detail 元素做了如下的规范。

- 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。

对于上面给出的错误消息，我们可以看出该元素对应着我们在 1.1 节介绍的错误细节对象，即 `FaultException<TDetail>` 异常最终序列化生成错误消息的时候，其 Detail 属性表示的错误细节对象被序列化成 Fault Detail 元素。

```
<s:Detail>
  <CalculationError xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.artech.com/">
    <Message>被除数 y 不能为零!</Message>
    <Operation>Divide</Operation>
  </CalculationError>
</s:Detail>
```

1.2.2 唯一可被传播的异常：FaultException

在整个 WCF 体系下，数据存在的形态大体可以分为两种：XML 和托管对象 (Managed Object)。WCF 建立在 .NET 平台下，开发人员利用托管语言 (C# 和 VB.NET) 提供了一个面向对象的编程模型，所以在 WCF 体系顶层的数据形态表现为 .NET 托管对象。而最终服务调用体现在消息的交换上，消息是基于 XML 的（除了少部分非 XML 的消息，比如 JSON）。从数据转换的角度讲，WCF 起到了将数据从这两种形态数据进行转换和适配的作用。

在 WCF 异常处理体系中,对于异常或者错误,在 XML 的世界里最终通过错误消息体现。而在托管对象的世界中,就是相应的 Exception 对象。前面我们站在消息交换的角度对 SOAP Fault 进行了介绍,接下来介绍它在托管世界的对立面,即 FaultException。

WCF 异常处理的编程主要是围绕着 FaultException 这个类型来完成的,所以很有必要深入地认识这个异常。先通过下面的代码片段来看看 FaultException 的基本定义。

```
[Serializable]
public class FaultException : CommunicationException
{
    //其他成员
    public FaultException();
    public FaultException(FaultReason reason);
    public FaultException(string reason);
    public FaultException(FaultReason reason, FaultCode code);
    public FaultException(string reason, FaultCode code);
    public FaultException(FaultReason reason, FaultCode code, string action);
    public FaultException(string reason, FaultCode code, string action);

    public string Action { get; }
    public FaultCode Code { get; }
    public override string Message { get; }
    public FaultReason Reason { get; }
}
```

上面的代码列出了 FaultException 的一些构造函数和公共属性。Action 表示最终生成到错误消息中<Action>报头的值。而 Code 和 Reason 则对应着 SOAP Fault 中的 Code 和 Reason 元素,它们的类型分别为 System.ServiceModel.FaultCode 和 System.ServiceModel.FaultReason。

1. FaultCode

FaultCode 表述错误的代码,该代码大体上可以看成是一种对错误的分类。在序列化 FaultException 对象生成错误消息的时候,该对象最终会生成 SOAP Fault 的 Code 节点。在介绍 SOAP Fault 的时候,我们提到 SOAP Fault 中的 Code 是一种具有层级关系(Hierarchical)的结构,这也体现在 FaultCode 的定义上。从下面对 FaultCode 的定义代码中,可以看到属性 SubCode 的属性是 FaultCode 本身。

```
public class FaultCode
{
    public FaultCode(string name);
    public FaultCode(string name, FaultCode subCode);
    public FaultCode(string name, string ns);
    public FaultCode(string name, string ns, FaultCode subCode);

    public static FaultCode CreateReceiverFaultCode(FaultCode subCode);
    public static FaultCode CreateReceiverFaultCode(string name, string ns);
    public static FaultCode CreateSenderFaultCode(FaultCode subCode);
    public static FaultCode CreateSenderFaultCode(string name, string ns);

    public bool IsPredefinedFault { get; }
    public bool IsReceiverFault { get; }
}
```

```

public bool IsSenderFault { get; }
public string Name { get; }
public string Namespace { get; }
public FaultCode SubCode { get; }
}

```

`FaultCode` 的 `Name` 和 `Namespace` 属性表示 SOAP Code 中 `Value` 元素的值, 而 `SubCode` 属性则自然对应着同名的 `SubCode` 元素。`IsPredefinedFault` 属性表示该 `Fault Code` 是否属于预定义的错误代码。WCF 通过命名空间确定其是否是预定义的 `Fault Code`。具体来讲, 只有具有以下三个命名空间的才属于预定义的错误代码:

- `http://schemas.xmlsoap.org/soap/envelope/` (SOAP 1.1)
- `http://www.w3.org/2003/05/soap-envelope` (SOAP 1.2)
- `http://schemas.microsoft.com/ws/2005/05/envelope/none`

对于名称为 `Sender` 和 `Receiver` 的预定义 `Fault Code`, 它们的 `IsSenderFault` 和 `IsReceiverFault` 分别返回 `true`。`FaultCode` 还定义了两组静态方法 (`CreateSenderFaultCode` 和 `CreateReceiverFaultCode`) 帮助我们方便地创建 `Sender` 和 `Receiver` 预定义 `FaultCode`。比如下面我们调用静态方法 `CreateSenderFaultCode` 创建一个 `FaultCode`, 该 `FaultCode` 的内容和我们前面给定的错误消息中的 `Fault Code` 是一致的。

```

var subCode = new FaultCode("CalculationError", "http://www.artech.com/");
FaultCode code = FaultCode.CreateSenderFaultCode(subCode);

```

对应错误消息中的 `Fault Code` 元素:

```

<s:Code>
  <s:Value>s:Sender</s:Value>
  <s:Subcode>
    <s:Value xmlns:a="http://www.artech.com/">a:CalculationError</s:Value>
  </s:Subcode>
</s:Code>

```

2. FaultReason

`FaultReason` 用于定义错误的原因, 在 SOAP Fault 中对应的元素为 `Reason`。下面给出 `FaultReason` 的基本定义。

```

public class FaultReason
{
    public FaultReason(IEnumerable<FaultReasonText> translations);
    public FaultReason(FaultReasonText translation);
    public FaultReason(string text);

    public FaultReasonText GetMatchingTranslation();
    public FaultReasonText GetMatchingTranslation(CultureInfo cultureInfo);
    public override string ToString();

    public SynchronizedReadOnlyCollection<FaultReasonText> Translations { get; }
}

```

虽然 SOAP Fault 的 Reason 的值仅仅是一个字符文本,但是出于对本地化 (Localization) 的支持,允许我们基于不同语言文化定义不同的内容。基于语言文化的 Reason 文本通过具有如下定义的类型 `System.ServiceModel.FaultReasonText` 表示。而属性 `FaultReason` 的 `Translations` 只读属性返回一个 `FaultReasonText` 的集合。

```
public class FaultReasonText
{
    public FaultReasonText(string text);
    public FaultReasonText(string text, CultureInfo cultureInfo);
    public FaultReasonText(string text, string xmlLang);
    public bool Matches(CultureInfo cultureInfo);

    public string Text { get; }
    public string XmlLang { get; }
}
```

从上面对 `FaultReasonText` 的定义中可以看到,可以通过指定 `CultureInfo` 和 `String` 对象指定基于某种语言文化的错误原因。如果没有显式指定 `CultureInfo`,默认采用的是当前线程的语言文化 (`CurrentUICulture`)。

对于对 `FaultReason` 对象的构建,既可以通过指定一个 `FaultReasonText` 集合创建支持多语言文化的 `FaultReason`,也可以通过指定单个 `FaultReasonText` 创建基于某个单一语言文化的 `FaultReason`。但最简单的莫过于直接指定一个字符串表示的 Reason 文本,它会默认采用当前线程的语言文化。

```
IList<FaultReasonText> reasonTexts = new List<FaultReasonText>();
reasonTexts.Add(new FaultReasonText("The input parameter is
invalid!", "en-US"));
reasonTexts.Add(new FaultReasonText("输入参数不合法!", "zh-CN"));
FaultReason reason = new FaultReason(reasonTexts);
```

在上面的代码中,创建了一个 `FaultReason` 对象对两种语言文化提供支持 (简体中文和美式英语)。该 `FaultReason` 在 SOAP Fault 中将会表示成如下一段 XML。

```
<s:Reason>
  <s:Text xml:lang="en-US">The input parameter is invalid!</s:Text>
  <s:Text xml:lang="zh-CN">输入参数不合法!</s:Text>
</s:Reason>
```

`FaultException` 异常提供了针对 `Code` 和 `Reason` 两个基本 SOAP Fault 元素的定义,而不能提供对 `Detail` 元素的定义,因为该元素是 `FaultException<TDetail>` 异常类型定义的。

3. FaultException<TDetail>

当从服务端抛出异常时,如果需要通过一个对象用于描述错误的消息信息,不管该对象的类型是基元类型 (比如 `String`, `Int` 等) 还是自定义类型 (比如自定义数据契约),都不得使用泛型的 `FaultException<TDetail>` 异常对象。下面给出了 `FaultException<TDetail>` 主要成员定义。

```
[Serializable]
public class FaultException<TDetail> : FaultException
{
    //其他成员
    public FaultException(TDetail detail);
    public FaultException(TDetail detail, FaultReason reason);
    public FaultException(TDetail detail, string reason);
    protected FaultException(SerializationInfo info, StreamingContext context);
    public FaultException(TDetail detail, FaultReason reason, FaultCode code);
    public FaultException(TDetail detail, string reason, FaultCode code);
    public FaultException(TDetail detail, FaultReason reason, FaultCode code,
        string action);
    public FaultException(TDetail detail, string reason, FaultCode code, string
        action);

    public TDetail Detail { get; }
}
```

`FaultException<TDetail>`直接继承自 `FaultException`，泛型参数表示错误细节类型。通过相应的构造函数在创建 `FaultException<TDetail>` 对象的时候指定类型为 `TDetail` 的错误细节对象，该对象通过只读属性 `Detail` 获取。错误细节类型必须是可序列化的，一般将其定义成数据契约的形式。该类型通过 `FaultContractAttribute` 特性应用在服务契约相应的操作上。

WCF 的服务将整个 `FaultException<TDetail>` 进行序列化并据此生成一个错误消息，其 `Detail` 属性表示的错误细节对象被序列化后的 XML 作为 SOAP Fault 的 `Detail` 元素。

到此为止，我们分别站在消息交换和编程的角度介绍了 SOAP Fault 和 `FaultException` 异常。在服务执行过程中，我们手工抛出 `FaultException` 异常，WCF 服务端框架会对该异常对象进行序列化并最终生成错误消息。当 WCF 客户端框架接收到该错误消息之后，对错误消息进行反序列化，重新生成并抛出 `FaultException` 异常。

WCF 框架自动为我们做了这么多“幕后”工作，使得开发人员可以完全采用编写一般的 .NET 应用程序的模式进行异常的处理。我们只需要在相应地方抛出相应异常，对于潜在出错的方法调用进行相应的异常捕获和处理。

WCF 的异常处理框架的核心功能就是实现 `FaultException` 异常和错误消息之间的转换，接下来着重讨论这个话题。

1.2.3 FaultException 异常和错误消息之间的转换

WCF 并不直接进行 `FaultException` 异常和错误消息之间的转换，它们之间的转换是通过另外一个作为中介的 `System.ServiceModel.Channels.MessageFault` 对象来完成的。`Message`（错误消息）、`MessageFault` 和 `FaultException` 通过如图 1-7 所示的“三角”关系实现了相互之间的转换。

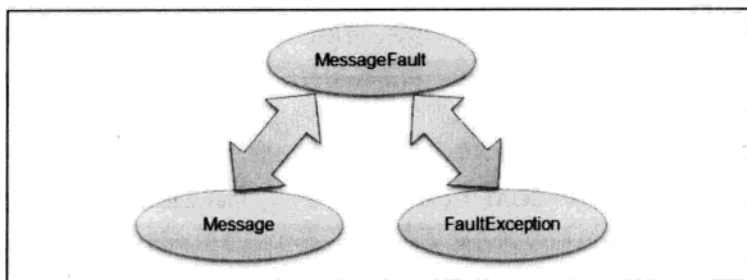


图 1-7 MessageFault、Message 和 FaultException 的“三角”转换关系

在介绍 MessageFault 之前，先来看看 MessageFault 类的基本定义。

```

public abstract class MessageFault
{
    //其他成员
    public static MessageFault CreateFault(Message message, int maxBufferSize);
    public static MessageFault CreateFault(FaultCode code, FaultReason reason);
    public static MessageFault CreateFault(FaultCode code, string reason);
    public static MessageFault CreateFault(FaultCode code, FaultReason reason,
        object detail);
    public static MessageFault CreateFault(FaultCode code, FaultReason reason,
        object detail, XmlObjectSerializer serializer);
    public static MessageFault CreateFault(FaultCode code, FaultReason reason,
        object detail, XmlObjectSerializer serializer, string actor);
    public static MessageFault CreateFault(FaultCode code, FaultReason reason,
        object detail, XmlObjectSerializer serializer, string actor, string node);

    public T GetDetail<T>();
    public T GetDetail<T>(XmlObjectSerializer serializer);
    public XmlDictionaryReader GetReaderAtDetailContents();

    public void WriteTo(XmlDictionaryWriter writer, EnvelopeVersion version);
    public void WriteTo(XmlWriter writer, EnvelopeVersion version);

    public virtual string Actor { get; }
    public abstract FaultCode Code { get; }
    public abstract bool HasDetail { get; }
    public bool IsMustUnderstandFault { get; }
    public virtual string Node { get; }
    public abstract FaultReason Reason { get; }
}
  
```

从上面给出的对 MessageFault 的定义可以看出，它的四个属性成员（Code、Reason、Node、Actor）和 FaultException，以及 SOAP Fault 的相应的子元素是相匹配的（对于 SOAP 1.2 规范中 SOAP Fault 的 Role 元素，在 SOAP 1.1 中的名称为 Actor）。而另一个元素 Detail 则可以通过两个泛型方法 GetDetail<T>获得。由于此操作需要对错误细节对象进行反序列化，所以需要指定错误细节类型对应的序列化器，默认情况下采用的是 DataContractSerializer。而属性 IsMustUnderstandFault 表述此错误是否是由于识别 SOAP 报头失败而造成的。实际上，它和 FaultCode 的 IsPredefinedFault 相对应，只要具有预定义的错误代码，IsMustUnderstandFault 就返回 True。

通过 `MessageFault` 众多的 `CreateFault` 静态方法, 可以以不同的组合方式指定构成 SOAP Fault 的 5 个元素。如果指定了错误细节对象, 需要指定与之匹配的序列化器以实现对其的序列化和反序列化。两个重载的 `WriteTo` 方法实现对 `MessageFault` 进行序列化, 并将序列化后的 XML 通过 `XmlDictionaryWriter` 或者 `XmlWriter` 写入相应的“流”中。

由于不同的 SOAP 规范的版本 (SOAP 1.1 和 SOAP 1.2) 对 `Message Fault` 的结构进行了不同的规定, 因此在调用 `WriteTo` 的时候需要显式地指定基于哪个版本进行写入 (SOAP 的版本通过 `EnvelopeVersion` 表示)。下面的示例代码创建了一个 `MessageFault` 对象, 分别针对 SOAP 1.1 和 SOAP 1.2 写到两个不同的 XML 文件中。读者可以仔细辨别最终生成的错误消息到底有多大的差别。

```
static void Main(string[] args)
{
    FaultCode code = FaultCode.CreateSenderFaultCode(
        new FaultCode("CalculationError", "http://www.artech.com/"));
    IList<FaultReasonText> reasonTexts = new List<FaultReasonText>();
    reasonTexts.Add(new FaultReasonText("The input parameter is invalid!",
        "en-US"));
    reasonTexts.Add(new FaultReasonText("输入参数不合法!", "zh-CN"));
    FaultReason reason = new FaultReason(reasonTexts);

    CalculationError detail = new CalculationError("Divide", "被除数 y 不能为
        零!");
    MessageFault fault = MessageFault.CreateFault(code, reason, detail,
        new DataContractSerializer(typeof(CalculationError)),
        "http://http://www.artech.com/calculatorservice",
        "http://http://www.artech.com/calculationcenter");

    string fileName1 = @"fault.soap11.xml";
    string fileName2 = @"fault.soap12.xml";
    WriteFault(fault, fileName1, EnvelopeVersion.Soap11);
    WriteFault(fault, fileName2, EnvelopeVersion.Soap12);
}

static void WriteFault(MessageFault fault, string fileName, EnvelopeVersion
version)
{
    using (FileStream stream = new FileStream(fileName, FileMode.Create,
        FileAccess.Write))
    {
        using (XmlDictionaryWriter writer =
            XmlDictionaryWriter.CreateTextWriter(stream, Encoding.UTF8, false))
        {
            fault.WriteTo(writer, version);
            Process.Start(fileName);
        }
    }
}
```

SOAP 1.1 (fault.soap11.xml):

```
<Fault xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <faultcode xmlns="" xmlns:a="http://www.artech.com/">
    a:CalculationError
```



```

</faultcode>
<faultstring xml:lang="en-US" xmlns="">The input parameter is invalid!
</faultstring>
<faultactor xmlns="">http://http://www.artech.com/calculatorservice</
faultactor>
<detail xmlns="">
  <CalculationError xmlns="http://www.artech.com/"
                    xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
    <Message>被除数 y 不能为零!</Message>
    <Operation>Divide</Operation>
  </CalculationError>
</detail>
</Fault>

```

SOAP 1.2 (fault.soap12.xml):

```

<Fault xmlns="http://www.w3.org/2003/05/soap-envelope">
  <Code>
    <Value>Sender</Value>
    <Subcode>
      <Value xmlns:a="http://www.artech.com/">a:CalculationError</Value>
    </Subcode>
  </Code>
  <Reason>
    <Text xml:lang="en-US">The input parameter is invalid!</Text>
    <Text xml:lang="zh-CN">输入参数不合法!</Text>
  </Reason>
  <Node>http://http://www.artech.com/calculationcenter</Node>
  <Role>http://http://www.artech.com/calculatorservice</Role>
  <Detail>
    <CalculationError xmlns="http://www.artech.com/"
                      xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
      <Message>被除数 y 不能为零!</Message>
      <Operation>Divide</Operation>
    </CalculationError>
  </Detail>
</Fault>

```

1. 如何实现 Message 和 MessageFault 之间的转换

MessageFault 可以作为 Message (错误消息) 和 FaultException 异常之间进行转换的中介。WCF 定义相应的应用编程接口实现了 Message 和 MessageFault, 以及 MessageFault 和 FaultException 异常之间的转换。下面先来关注一下如何实现 Message 和 MessageFault 之间的转换。

由于 MessageFault 对应错误消息中主体部分的 Fault 元素, 即 SOAP Fault, 所以对于一个给定的表示错误消息的 Message 对象, 可以通过提取 SOAP Fault 部分内容, 进而创建相应的 MessageFault 对象。MessageFault 提供了如下一个 CreateFault 静态方法, 使我们能通过传入一个 Message 对象创建 MessageFault (参数 maxBufferSize 为最大消息缓冲区大小)。

```

public abstract class MessageFault
{
  //其他成员
  public static MessageFault CreateFault(Message message, int maxBufferSize);
}

```

在下面的代码中,借助于 `Message` 的静态方法 `CreateMessage`,通过逐个指定 `FaultCode`、`FaultReason`、`Detail` 和 `Action` 的方式创建了一个错误消息。然后将其传入 `MessageFault` 的 `CreateFault` 静态方法创建 `MessageFault` 对象。最后通过 `MessageFault` 的 `GetDetail<T>` 方法得到错误细节对象,通过输出的信息可以证实该 `MessageFault` 中的错误信息和创建消息时指定的消息是一致的。

```
CalculationError detail = new CalculationError("Divide", "被除数 y 不能为零!");
FaultCode code = FaultCode.CreateSenderFaultCode(new
FaultCode("CalculationError", "http://www.artech.com/"));
IList<FaultReasonText> reasonTexts = new List<FaultReasonText>();
Message message = Message.CreateMessage(MessageVersion.Soap12WSAddressing10,
code, "被除数 y 不能为零!", detail,
"http://www.artech.com/calculator/service/dividecalculationerrorfault");
MessageFault messageFault = MessageFault.CreateFault(message, int.MaxValue);
detail = messageFault.GetDetail<CalculationError>();
Console.WriteLine("Operation: {0}", detail.Operation);
Console.WriteLine("Message: {0}", detail.Message);
```

输出结果:

```
Operation: Divide
Message: 被除数 y 不能为零!
```

既然可以通过提取错误消息的 SOAP Fault 创建相应的 `MessageFault`,同样可以通过给定的 `MessageFault` 对象,基于某种消息版本和 `Action` 报头,创建一个错误消息。`Message` 类型中定义的下面这个静态的 `CreateMessage` 方法可以帮我们实现这样的操作。

```
public abstract class Message : IDisposable
{
    //其他成员
    public static Message CreateMessage(MessageVersion version, MessageFault
    fault, string action);
}
```

下面的例子中,通过调用 `MessageFault` 的 `CreateFault` 方法创建了一个 `MessageFault` 对象。将其传入 `Message` 的 `CreateMessage` 静态方法,并指定不同的 `MessageVersion` (`MessageVersion.Soap11WSAddressingAugust2004` 和 `MessageVersion.Soap12WSAddressing10`),创建了不同的错误消息。有兴趣的读者可以仔细分析一下基于不同的消息版本,针对同一个 `MessageFault` 对象创建的错误消息都有哪些差异。

```
MessageFault messageFault =
MessageFault.CreateFault(FaultCode.CreateSenderFaultCode("Infrastructure",
"http://www.artech.com/"), "Message Timeout");
Message messageSoap11 =
Message.CreateMessage(MessageVersion.Soap11WSAddressingAugust2004, message
Fault, "http://www.artech.com/calculatefault");
Message messageSoap12 = Message.CreateMessage(MessageVersion.Soap12
WSAddressing10, messageFault, "http://www.artech.com/calculatefault");
using (XmlWriter writer1 = new XmlTextWriter("faultmessage.soap11.addressing2004.
xml", Encoding.UTF8))
using (XmlWriter writer2 = new XmlTextWriter("faultmessage.soap12.
addressing10.xml", Encoding.UTF8))
```

```

{
    messageSoap11.WriteMessage(writer1);
    messageSoap12.WriteMessage(writer2);
}
Process.Start("faultmessage.soap11.addressing2004.xml");
Process.Start("faultmessage.soap12.addressing10.xml");

```

SOAP 1.1+WS-Addressing 2004 的错误消息 (faultmessage.soap11.addressing2004.xml):

```

<s:Envelope xmlns:a="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://www.artech.com/calculatefault</a:Action>
  </s:Header>
  <s:Body>
    <s:Fault>
      <faultcode xmlns:a="http://www.artech.com/">a:Infrastructure</faultcode>
      <faultstring xml:lang="en-US">Message Timeout</faultstring>
    </s:Fault>
  </s:Body>
</s:Envelope>

```

SOAP 1.2+WS-Addressing 1.0 的错误消息 (faultmessage.soap12.addressing10.xml):

```

<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://www.artech.com/calculatefault</a:Action>
  </s:Header>
  <s:Body>
    <s:Fault>
      <s:Code>
        <s:Value>s:Sender</s:Value>
        <s:Subcode>
          <s:Value xmlns:a="http://www.artech.com/">a:Infrastructure</s:Value>
        </s:Subcode>
      </s:Code>
      <s:Reason>
        <s:Text xml:lang="en-US">Message Timeout</s:Text>
      </s:Reason>
    </s:Fault>
  </s:Body>
</s:Envelope>

```

2. 如何实现 MessageFault 和 FaultException 之间的转换

上面介绍的是 MessageFault 和 Message 之间的转换关系,下面介绍 MessageFault、Message 和 FaultException“三角”关系中的另一组转换关系,即 MessageFault 和 FaultException 之间的转换。

WCF 将实现 MessageFault 和 FaultException 之间的转换的应用编程接口定义在 FaultException 类型中。两个静态 CreateFault 方法实现 MessageFault 向 FaultException 的转换,而实例方法 CreateMessageFault 则将 FaultException 对象转换成相应的 MessageFault 对象。三个方法定义如下,其中 faultDetailTypes 代表错误细节类型列表,这是为对 FaultException <TDetail>对象的反序列化服务的。

```

[Serializable]
public class FaultException : CommunicationException
{
    //其他成员
    public static FaultException CreateFault(MessageFault messageFault,
        params Type[]
        faultDetailTypes);
    public static FaultException CreateFault(MessageFault messageFault,
        string action, params Type[] faultDetailTypes);

    public virtual MessageFault CreateMessageFault();
}

```

在下面的代码中,通过调用 `MessageFault` 的静态方法 `CreateFault` 创建了一个 `MessageFault` 对象。然后调用 `FaultException` 的静态方法 `CreateFault` 创建 `FaultException` 对象。由于构建 `MessageFault` 时传入一个 `CalculationError` 作为错误细节,所以返回的异常类型应该是 `FaultException<CalculationError>` 对象。最后将该异常对象的相关信息在控制台上输出。

```

FaultCode code = FaultCode.CreateSenderFaultCode(new FaultCode("Calculation
Error", "http://www.artech.com/"));
IList<FaultReasonText> reasonTexts = new List<FaultReasonText>();
reasonTexts.Add(new FaultReasonText("The input parameter is invalid!",
    "en-US"));
reasonTexts.Add(new FaultReasonText("输入参数 y 不合法!", "zh-CN"));
FaultReason reason = new FaultReason(reasonTexts);
CalculationError detail = new CalculationError("Divide", "被除 y 数能为零!");
MessageFault fault = MessageFault.CreateFault(code, reason, detail,
    new DataContractSerializer(typeof(CalculationError)),
    "http://http://www.artech.com/calculatorservice",
    "http://http://www.artech.com/calculationcenter");

FaultException<CalculationError> exception =
    FaultException.CreateFault(fault, typeof(CalculationError)) as
    FaultException<CalculationError>;
Console.WriteLine("Fault Code: {0}", exception.Code.Name);
Console.WriteLine("\tSubCode: {0}:{1}", exception.Code.SubCode.Namespace,
    exception.Code.SubCode.Name);
Console.WriteLine("Fault Reason:");
foreach (var reasonText in exception.Reason.Translations)
{
    Console.WriteLine("\t{0}:{1}", reasonText.XmlLang, reasonText.Text);
}
Console.WriteLine("Detail:");
Console.WriteLine("\tOperation:{0}", exception.Detail.Operation);
Console.WriteLine("\tMessage:{0}", exception.Detail.Message);

```

输出结果:

```

Fault Code: Sender
          SubCode: http://www.artech.com/:CalculationError
Fault Reason:
          en-US:The input parameter is invalid!
          zh-CN:输入参数不合法!
Detail:
          Operation:Divide
          Message:被除数 y 不能为零!

```

上面给出的是如何将一个 MessageFault 对象转换成一个 FaultException 异常的例子，如果需要进行相反的操作，只需要直接调用 FaultException 异常实例的 CreateMessageFault 方法即可。清楚了应该调用怎样的 API 进行 MessageFault 和 FaultException 之间的转换，下面进一步深入了解其内部的实现原理。在自身的异常处理框架内部，WCF 实际上是通过 FaultFormatter 对象实现两者之间的转换的。

3. FaultException 与 MessageFault 转换的核心：FaultFormatter

在上册的第 6 章“消息 (Message)”中谈到：WCF 借助于 MessageFormatter 实现方法调用和消息之间的转换。具体来说，客户端通过 ClientMessageFormatter 将服务操作方法调用转换成请求消息（其中主要涉及对参数对象的序列化），将接收到的回复消息转换成服务操作方法对应的返回值或者 out/ref 参数。服务端则通过 DispatchMessageFormatter 实现与此相反的操作。

MessageFormatter 实现了在正常的服务调用过程中方法调用和消息之间的转换。但是当异常（这里指的是 FaultException 异常）从服务端抛出后，WCF 需要一个相似的组件实现类似的功能，即在服务端对异常对象进行序列化并生成错误消息，在客户端对接收到的错误消息进行反序列化重建并抛出异常。这样的一个使命由 FaultFormatter 担当。不过由于 MessageFault 是 FaultException 和错误消息进行转换的中介，所以 FaultFormatter 并不直接进行两者之间的转换，而是实现 FaultException 和 MessageFault 之间的转换。

FaultFormatter 仅仅是 WCF 的一个内部类型，但是对该对象的深刻认识将非常有助于我们理解 WCF 的整个异常处理机制。FaultFormatter 在客户端和服务端所扮演的角色是不同的。

- 客户端通过解析回复错误消息生成的 MessageFault，该 MessageFault 最终被转换成 FaultException 异常并抛出；
- 服务端则将抛出的 FaultException 异常转换成 MessageFault，以便后续的步骤生成相应的错误消息。

客户端和服务端这种职责的不同可以通过 FaultFormatter 实现的两个接口 (IClientFaultFormatter 和 IDispatchFaultFormatter) 的定义看出来。

```
internal interface IClientFaultFormatter
{
    FaultException Deserialize(MessageFault messageFault, string action);
}
internal interface IDispatchFaultFormatter
{
    MessageFault Serialize(FaultException faultException, out string action);
}
```

内部接口 IClientFaultFormatter 和 IDispatchFaultFormatter 分别定义了 FaultFormatter 在客户端和服务端的职能，即它们分别实现对 FaultException 对象的反序列化和序列化。在对

`FaultException` 对象进行序列化时需要提取 `Action` 属性作为错误消息的 `Action` 报头，而对 `MessageFault` 进行反序列化生成 `FaultException` 对象的时候需要从外部指定 `Action` 属性的值，所以两个方法各有一个 `action` 参数。

WCF 定义了一个内部类 `FaultFormatter` 实现了这两个接口，并将其作为服务端和客户端的 `FaultFormatter`。下面给出了 `FaultFormatter` 类型的定义。

```
internal class FaultFormatter : IClientFaultFormatter,
IDispatchFaultFormatter
{
    public FaultException Deserialize(MessageFault messageFault, string action);
    public MessageFault Serialize(FaultException faultException, out string action);
}
```

WCF 将绝大部分序列化和反序列化的工作都交付给两个序列化器：`DataContractSerializer` 和 `XmlSerializerObjectSerializer`，对于 `FaultException` 异常对象的序列化自然也不例外。WCF 定义了两个具体的类型 `DataContractSerializerFaultFormatter` 和 `XmlSerializerFaultFormatter`。它们直接继承自 `FaultFormatter`，分别采用 `DataContractSerializer` 和 `XmlSerializerObjectSerializer` 作为相应的序列化器。`IClientFaultFormatter`、`IDispatchFaultFormatter`、`FaultFormatter`、`DataContractSerializerFaultFormatter` 和 `XmlSerializerFaultFormatter` 之间的关系可以简单地通过图 1-8 所示的类图表示。

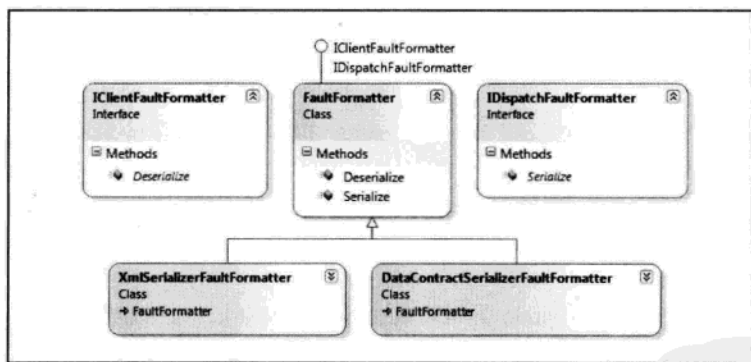


图 1-8 FaultFormatter 体系结构

1.3 WCF 异常处理体系剖析

WCF 客户端和服务端的异常处理框架体系相互协作，使得开发人员可以按照我们熟悉的方式进行异常的处理，即在服务操作执行过程中抛出异常（`FaultException`），在调用服务时捕获异常。我们完全感觉不到“分布式”的存在，就像是在调用典型的“本地”操作一般。为了实现这样的效果，WCF 在内部为我们做了很多。

消息交换是 WCF 进行通信的唯一手段，消息不仅仅是正常服务调用请求和回复的载体，

服务端抛出的异常也是通过消息的形式传向客户端的。所以实现异常与消息之间的转换是整个异常处理体系的核心，而 WCF 的异常处理框架就着力于完成这样的功能。

可以这样来简单地描述 WCF 异常处理框架的功能实现：WCF 服务端将抛出的 `FaultException` 异常进行序列化，并根据消息的 SOAP 规范（SOAP 1.1 或 SOAP 1.2）和 WS-Addressing 规范（WS-Addressing 2004 和 WS-Addressing 1.0）生成错误消息。被传入信道层并经过一系列的信道后，该错误消息最终借助于传输层返回到客户端。客户端信道层接收到该错误消息并经过相应的处理后，将其反序列化以重建 `FaultException` 异常并最终将重建的 `FaultException` 异常抛出。对于最终的开发者而言，感觉就像服务端抛出的 `FaultException` 异常直接被客户端捕获了一样。

WCF 并不直接进行 `FaultException` 和错误消息之间的转换，而是借助于 `MessageFault` 这一中间对象。图 1-9 体现了错误（Fault）在整个 WCF 异常处理过程中的流转。

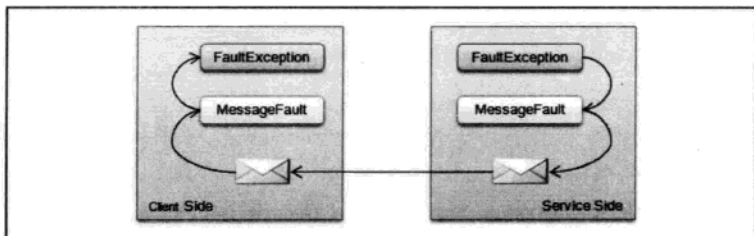


图 1-9 错误（Fault）的三种状态在 WCF 异常处理过程中的流转

对 `FaultException` 进行序列化和反序列化的核心对象是 `FaultFormatter`，要了解 WCF 整个异常处理框架的实现原理，首先需要知道 `FaultFormatter` 是如何创建的。

1.3.1 FaultFormatter

WCF 的服务端和客户端均需要一个 `FaultFormatter` 对象，分别用于对 `FaultException` 异常对象的序列化和反序列化。下面分别介绍 `FaultFormatter` 对象在服务端和客户端是如何被创建的。

1. DispatchFaultFormatter

`FaultFormatter` 在服务端创建于服务寄宿之时。在 `ServiceHost` 被初始化的过程中，WCF 会为服务的每个终结点创建相应的终结点分发器（`EndpointDispatcher`）。而对于每一个被创建出来的终结点分发器都具有一个相应的分发运行时（`DispatchRuntime`）。分发运行时是整个 WCF 运行时框架的核心，一系列的对象和组件被它引用以实现对整个消息分发和操作执行行为的控制。

在分发运行时的初始化过程中, WCF 会根据服务的描述创建一系列的分发操作 (DispatchOperation)。分发操作可以看成是服务操作在运行时的表示, 而最终对服务操作的执行就是通过它来完成的。WCF 会通过当前终结点的契约描述 (ContractDescription), 遍历每一个操作 (OperationDescription), 并借此创建相应的分发操作。ServiceEndpoint、ContractDescription 和 OperationDescription 三者之间的关系通过下面的类型定义代码一目了然。

```
public class ServiceEndpoint
{
    //其他成员
    public ContractDescription Contract { get; }
}
public class ContractDescription
{
    //其他成员
    public OperationDescriptionCollection Operations { get; }
}
public class OperationDescription
{
    //其他成员
    public FaultDescriptionCollection Faults { get; }
}
```

对于作为操作描述的 OperationDescription 类型, 有一个只读属性 Faults, 类型为 System.ServiceModel.Description.FaultDescriptionCollection, 表示与本操作相关的所有错误描述的集合。该集合的每一个元素为 System.ServiceModel.Description.FaultDescription 对象, 它们是通过对应应用在操作上的 FaultContractAttribute 特性进行反射而创建的。FaultDescription 的定义如下, 它与 FaultContractAttribute 类具有一样的属性成员。

```
public class FaultDescription
{
    //其他成员
    public string Action { get; internal set; }
    public Type DetailType { get; set; }
    public bool HasProtectionLevel { get; }
    public string Name { get; set; }
    public string Namespace { get; set; }
    public ProtectionLevel ProtectionLevel { get; set; }
}
```

当 WCF 服务端运行时以操作描述为基础创建相应的 DispatchOperation 后, 会根据错误描述创建 FaultFormatter 对象, 声明类型为 IDispatchFaultFormatter。

```
public sealed class DispatchOperation
{
    //其他成员
    public SynchronizedCollection<FaultContractInfo> FaultContractInfos
    { get; }
    internal IDispatchFaultFormatter FaultFormatter { get; set; }
}
```

通过上面的代码片段还会看到, 除了一个内部属性 FaultFormatter 之外, 还具有一个类

型为 `SynchronizedCollection<FaultContractInfo>` 的属性 `FaultContractInfos`。而作为集合元素的 `System.ServiceModel.Dispatcher.FaultContractInfo` 对象表示错误契约相关的信息，该集合与操作描述（`OperationDescription`）的 `Faults` 属性相匹配。实际上 `FaultContractInfo` 仅仅包含两项用于实现序列化的信息，即错误细节类型和 `Action`。

```
public class FaultContractInfo
{
    //其他成员
    public string Action { get; }
    public Type Detail { get; }
}
```

序列化和反序列化需要以类型的确定为前提，所以 `FaultFormatter` 在进行序列化或者反序列化过程之前，需要确定错误细节的类型。`MessageFault` 是对错误消息主体部分的描述，本身没有 `Action` 属性（对应于错误消息的报头）。对于一个 SOAP 消息来说，`<Action>` 报头是必不可少的，而 `FaultException` 类型也具有相应的 `Action` 属性定义。在 WCF 服务端框架内，在实现 `FaultException` 异常对象向错误消息转换的过程中，除了要提供与 `FaultException` 对等的 `MessageFault` 之外，还需要提供 `FaultException` 的 `Action` 属性值。这正是为何 `FaultFormatter` 在进行序列化工作的时候依赖于一个 `FaultContractInfo` 集合的原因。在构建 `FaultFormatter` 这么一个对象的时候，就需要传入一个这样的集合对象，这可以从 `FaultFormatter` 的构造函数看出来。

```
internal class FaultFormatter : IClientFaultFormatter,
IDispatchFaultFormatter
{
    //其他成员
    internal FaultFormatter(Type[] detailTypes);
    internal FaultFormatter(SynchronizedCollection<FaultContractInfo>
        faultContractInfoCollection);
}
```

关于序列化过程中对 `Action` 的指定，WCF 内部采用这样一个规则：如果 `FaultException` 对象本身具有一个 `Action`，则返回该值，否则就在 `FaultContractInfo` 列表中找到一个与错误细节类型相匹配的 `FaultContractInfo` 对象。如果该对象具有一个有效的 `Action` 属性，则返回之，如果该 `FaultContractInfo` 仍然没有定义 `Action` 属性，那么 WCF 会根据采用的 WS-Addressing 版本选择默认的 `Action` 值。

```
WS-Addressing 2004: http://www.w3.org/2005/08/addressing/soap/fault
WS-Addressing 1.0: http://schemas.xmlsoap.org/ws/2004/08/addressing/fault
```

2. ClientFaultFormatter

`FaultFormatter` 在客户端的创建方式与服务端有点相似，同样是基于 `FaultDescription` 的方式创建，所以在这里仅对整个过程做一个概括性的介绍。

在 `ChannelFactory<TChannel>` 开启之后，客户端运行时（`Client Runtime`）被 WCF 创建出来。在客户端运行时初始化过程中，WCF 为每一个操作创建客户端操作（`ClientOperation`）。

这与在服务端初始化分发运行时 (DispatchRuntime) 分发操作的创建类似。和 DispatchOperation 定义一样, ClientOperation 同样有 FaultContractInfos 和 FaultFormatter 两个属性, 不过 FaultFormatter 的类型为 IClientFaultFormatter。

```
public sealed class ClientOperation
{
    //其他成员
    public SynchronizedCollection<FaultContractInfo> FaultContractInfos
    { get; }
    internal IClientFaultFormatter FaultFormatter { get; set; }
}
```

3. DataContractSerializerFaultFormatter 还是 XmlSerializerFaultFormatter

为了满足不同的序列化方式的需要, WCF 异常处理框架使用两个不同的 FaultFormatter, 即 DataContractSerializerFaultFormatter 和 XmlSerializerFaultFormatter, 它们分别利用 DataContractSerializer 和 XmlSerializer 这两个不同的序列化器实现针对 FaultException 异常对象的序列化和反序列化。那么具体对 FaultFormatter 的选择是如何实现的呢?

在 WCF 服务端和客户端的异常处理框架体系内, 对 FaultFormatter 的提供机制最终是通过 DispatchOperation 和 ClientOperation 的 FaultFormatter 属性实现的。在 DispatchOperation 和 ClientOperation 被创建的时候, 并不会伴随着 FaultFormatter 的创建。在默认的情况下, WCF 采用懒情加载 (Lazy Loading) 的方式创建 FaultFormatter。也就是说 WCF 在真正使用到 FaultFormatter 的时候, 才动态地创建该对象。通过这种方式创建的永远是 DataContractSerializerFaultFormatter, 这也正是 WCF 采用 DataContractSerializerFaultFormatter 作为默认的 FaultFormatter 的原因。

在 1.1 节中, 我们说可以在服务契约、服务类型或者服务操作方法上面应用 XmlSerializerFormatAttribute 特性让 WCF 采用 XmlSerializer 作为序列化器对 FaultException 异常进行序列化和反序列化。最终体现在 WCF 内部会根据这样一个特性选择 XmlSerializerFaultFormatter 而不是 DataContractSerializerFaultFormatter 作为最终的 FaultFormatter。那么对 XmlSerializerFaultFormatter 的选择又是如何实现的呢?

WCF 对 XmlSerializerFaultFormatter 的选择是通过一个类型 System.ServiceModel.Description.XmlSerializerOperationBehavior 的特殊的操作行为实现的。从下面给出的 XmlSerializerOperationBehavior 的基本定义可以看到, ApplyClientBehavior 和 ApplyDispatchBehavior 方法分别将当前的 ClientOperation 和 DispatchOperation 的 FaultFormatter 设置为 XmlSerializerFaultFormatter。

```
public class XmlSerializerOperationBehavior : IOperationBehavior,
    IWSDLExportExtension
{
    //其他成员
    void IOperationBehavior.AddBindingParameters(OperationDescription
    description,
```

```

        BindingParameterCollection parameters);
void IOperationBehavior.ApplyClientBehavior(OperationDescription
description,
    ClientOperation proxy)
{
    proxy.FaultFormatter = this.CreateXmlSerializerFaultFormatter();
    //伪代码
}
void IOperationBehavior.ApplyDispatchBehavior(OperationDescription
description,
    DispatchOperation dispatch)
{
    dispatch.FaultFormatter = this.CreateXmlSerializerFaultFormatter();
    //伪代码
}
void IOperationBehavior.Validate(OperationDescription description);
}

```

无论是客户端还是服务器端，在初始化操作描述的时候，WCF 会通过反射确定服务契约或者操作方法上面是否应用了 `XmlSerializerFormatAttribute` 特性，从而决定是否添加 `XmlSerializerOperationBehavior` 这样一个操作行为到该操作的行为列表中。操作一旦具有了这样一个行为，那么对应的 `ClientOperation` 和 `DispatchOperation` 自然选择了 `XmlSerializerFaultFormatter`。

4. 异常的抛出、序列化、反序列化与捕获

如果服务操作在执行过程中抛出 `FaultException` 异常，WCF 会获取当前分发操作的 `FaultFormatter`，调用 `Serialize` 方法对异常对象进行序列化。序列化完成后得到相应的 `MessageFault` 对象和 `Action` 值，这两个值最终通过调用 `Message` 的 `CreateMessage` 静态方法生成一个错误消息对象。

客户端的服务调用最终通过客户端操作对象完成。当调用服务获得回复消息后，如果回复消息是错误消息，WCF 会调用 `MessageFault` 的 `CreateFault` 将消息转换成 `MessageFault` 对象，并获取 `Action` 值。最终通过客户端操作得到 `FaultFormatter`，传入 `MessageFault` 对象和 `Action` 值调用 `Deserialize` 方法在客户端重建 `FaultException` 异常对象并将其抛出来。

1.3.2 ServiceDebugBehavior 如何实现对异常细节传播

服务端只有抛出 `FaultException` 异常才能被正常地序列化成错误消息向客户端传播。对于一般的异常（比如执行 `Divide` 操作抛出的 `DivideByZeroException` 异常），在默认的情况下，异常信息是无法向客户端传播的。倘若为某个服务应用了 `ServiceDebugBehavior` 这样一个服务行为，并开启了 `IncludeExceptionDetailInFaults` 开关，异常信息将会原封不动地传播到客户端。WCF 内部是如何处理抛出的非 `FaultException` 异常的呢？

在执行服务操作过程中，如果抛出一个非 `FaultException` 异常，WCF 会先判断服务是否应用了一个开启了 `IncludeExceptionDetailInFaults` 开关的 `ServiceDebugBehavior` 服务

行为。如果没有, WCF 会手工创建一个 `MessageFault` 对象, 并根据当前线程的语言文化从资源文件中获取一段固定的文本作为 `MessageFault` 的 `FaultReason` (就是我们在 1.1 节的例子中看到的那段文字)。

此外, 固定的 `FaultCode` 被创建出来作为该 `MessageFault` 的 `Code`。WCF 最终将该 `MessageFault` 转换成一个错误消息, 并采用固定的值作为该消息的 `Action` 报头。所以无论服务端抛出怎样的异常, 客户端捕获的总是具有相同信息的 `FaultException` 异常。

如果服务行为 `ServiceDebugBehavior` 的 `IncludeExceptionDetailInFaults` 开启, WCF 则会基于该抛出的异常创建 `ExceptionDetail` 对象, 并将该对象作为细节对象创建 `MessageFault` (采用固定 `FaultCode`), 最终将此 `MessageFault` 转换生成错误消息, 当然 `Action` 也是采用固定的预定值。在这种情况下, 服务端抛出的信息总是能够原封不动地传递到客户端。而客户端捕获的总是一个泛型的 `FaultException<ExceptionDetail>` 异常。

1. ExceptionDetail 对象为何能被反序列化

对于异常对象的序列化和反序列化工作, 最终都会落在 `FaultFormatter` 这么一个对象上。无论是序列化还是反序列化, 都需要预先确定对象的类型。`FaultFormatter` 依赖创建时指定的一个 `FaultContractInfo` 列表来获知具体的类型, 而该列表最初来源于应用在操作方法上的 `FaultContractAttribute` 定义。

那么对于应用了 `IncludeExceptionDetailInFaults` 属性为 `True` 的 `ServiceDebugBehavior` 服务行为, 客户端是如何将错误消息中显示错误细节的 XML 反序列化生成 `ExceptionDetail` 对象的呢? 由于我们不曾通过 `FaultContractAttribute` 特性将 `ExceptionDetail` 类型应用在相应的操作方法上面, 因此 `FaultFormatter` 无法确定反序列化对象的类型, 照理说反序列化是无法成功的, 这是为何呢?

原因其实很简单, WCF 在初始化 `FaultFormatter` 的时候会基于 `ExceptionDetail` 类型创建 `FaultContractInfo` 对象, 并将其添加到属于自己的 `FaultContractInfo` 列表中。相应的实现基本上可以通过下面的伪代码体现。

```
internal class FaultFormatter : IClientFaultFormatter,
IDispatchFaultFormatter
{
    //其他成员
    private FaultContractInfo[] faultContractInfos;
    internal FaultFormatter(SynchronizedCollection<FaultContractInfo>
        faultContractInfoCollection)
    {
        List<FaultContractInfo> list= new List<FaultContractInfo>
            (faultContractInfoCollection);
        faultContractInfoCollection.Add(new FaultContractInfo(
            "http://schemas.microsoft.com/net/2005/12/windowscommunicationfoundation/
            dispatcher/fault", typeof(ExceptionDetail)));
        this.faultContractInfos = list.ToArray();
    }
}
```

2. 如果直接抛出 `FaultException<ExceptionDetail>` 呢

既然 `FaultFormatter` 能够自动实现基于 `ExceptionDetail` 对象的序列化和反序列化, 就意味着我们可以在具体的服务操作中直接抛出 `FaultException<ExceptionDetail>` 异常, 而无须再将 `ExceptionDetail` 作为错误契约类型通过 `FaultContractAttribute` 特性应用到相应的服务操作上面了。以计算服务为例, 在 `Divide` 方法中我们直接用 `ExceptionDetail` 封装在运算过程中抛出的异常, 最终抛出 `FaultException<ExceptionDetail>` 异常。

```
[ServiceBehavior(Namespace = "http://www.artech.com/")]
public class CalculatorService : ICalculator
{
    public int Divide(int x, int y)
    {
        try
        {
            return x / y;
        }
        catch (Exception ex)
        {
            throw new FaultException<ExceptionDetail>(new ExceptionDetail(ex),
                ex.Message);
        }
    }
}
```

在客户端就可以直接捕获 `FaultException<ExceptionDetail>` 异常了。下面的代码中, 将捕获的 `FaultException<ExceptionDetail>` 异常相关的信息打印出来。

```
using (ChannelFactory<ICalculator> channelFactory = new ChannelFactory
<ICalculator>(
    "calculatorservice"))
{
    ICalculator calculator = channelFactory.CreateChannel();
    using (calculator as IDisposable)
    {
        try
        {
            int result = calculator.Divide(1, 0);
        }
        catch (FaultException<ExceptionDetail> ex)
        {
            Console.WriteLine("Action: {0}", ex.Action);
            Console.WriteLine("Code: {0}:{1}", ex.Code.Namespace, ex.Code.Name);
            Console.WriteLine("Detail");
            Console.WriteLine("\tMessage: {0}", ex.Detail.Message);
            Console.WriteLine("\tType: {0}", ex.Detail.Type);
        }
    }
}
```

输出结果:

```
Action:
http://schemas.microsoft.com/net/2005/12/windowscommunicationfoundation/di
spatcher/fault
```

Code: http://www.w3.org/2003/05/soap-envelope: Sender

Detail:

Message: 试图除以零。

Type: System.DivideByZeroException

1.4 WCF 异常处理扩展

对异常处理方式的定制是一种非常常见的 WCF 扩展行为，这是通过自定义错误处理器 (ErrorHandler) 来实现的。

1.4.1 处理器 (ErrorHandler)

作为 WCF 异常处理扩展体系的核心，错误处理器实际上是一个很简单的组件。错误处理器实现了具有如下定义的 `System.ServiceModel.Dispatcher.IErrorHandler` 接口。它仅仅具有 `HandleError` 与 `ProvideError` 两个方法成员。

```
public interface IErrorHandler
{
    bool HandleError(Exception error);
    void ProvideFault(Exception error, MessageVersion version, ref Message
        fault);
}
```

从 WCF 运行时架构体系来讲，错误处理器隶属于信道分发器 (`ChannelDispatcher`)，它维护着错误处理器列表。可以将多个错误处理器应用到到相应信道分发器上，这些错误处理器最终连成一个管道。每个错误处理器用于实现某个单一的异常处理任务，比如日志记录、敏感信息的屏蔽等 (关于以 `ChannelDispatcher` 和 `EndpointDispatcher` 为核心的 WCF 服务端消息分发体系，请参阅本书的第 9 章“扩展 (Extension)”)。

```
public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public Collection<IErrorHandler> ErrorHandlers { get; }
}
```

当异常抛出的时候，WCF 遍历相应信道分发器的 `ErrorHandlers` 属性表示的错误处理器集合，并调用错误处理器的 `ProvideFault` 方法。`ProvideFault` 具有三个参数，第一个参数 `error` 表示抛出的异常，其类型可能是 `FaultException`，也可能是一般的 CLR 异常。后面的两个参数分别表示消息的版本和承载错误信息的错误消息。通过创建或者替换 `fault` 参数表示的错误消息，能很容易地控制最终返回到客户端的错误信息。

对于 `ProvideFault` 方法的执行，有一些值得注意的地方。该方法的执行发生在会话终止之前，并且是在执行操作方法的线程中执行的。换句话说，`ProvideFault` 方法是以与操作方法同步的方式执行的，所以在自定义 `ErrorHandler` 的时候，不应该将一些比较耗时的操作实

现在 ProvideFault 方法中。

当与信道分发器关联的所有错误处理器的 ProvideFault 执行后，错误消息被传送回客户端。此后，错误处理器的 HandleError 方法被依次调用，用于进行一些本地的异常处理操作。HandleError 的操作是服务端的本地行为，与客户端无关。HandleError 方法是以异步的方式被执行的，所以可以实现一些相对耗时的操作，比如异常日志记录。

HandleError 方法具有一个布尔类型的返回值。如果信道分发器的每一个 ErrorHandler 均返回 False，WCF 会将最终得到的异常视为“未被处理的异常”。在这种情况下，如果采用会话，会话信道（Session Channel）会被中止，而实例上下文也会被回收。

注：在 MSDN 中对于 HandleError 的返回值具有这样的描述：当 ErrorHandler 列表的任何一个 HandleError 方法返回 True 时，后续 ErrorHandler 的 HandleError 将不会执行。但是，经过对 WCF 相关实现代码的分析，基于试验证明：当多个 ErrorHandler 被应用到某个 ChannelDispatcher 上面后，后续 ErrorHandler 的执行与前面 ErrorHandler 的 HandleError 返回值无关。

1.4.2 实例演示：通过 WCF 扩展实现与 EntLib 的集成（S105）

在上册的最后一章，笔者给出了一个具体的应用 WCF 的分布式应用实例。在这个例子中，利用 WCF 的扩展实现了一些设计、架构模式，比如 AOP、IoC 等还通过 WCF 扩展实现了与微软企业库（EntLib）异常处理应用块（EHAB，Exception Handling Application Block）的集成。当时由于缺乏相应的背景知识，不可能介绍具体的实现，现在可以详细来讲述这是如何实现的。考虑到不是每个人都了解微软企业库的 EHAB，所以在这里还是很有必要对 EHAB 这样一个简单的异常处理框架进行概要性的介绍。

1. 异常处理应用块简介

EHAB 采用基于“策略”的异常处理机制，异常处理策略通过配置定义。EHAB 中的异常处理策略大致可以通过下面的公式表示。

$$\text{异常处理策略 (Exception Handling Policy)} = \text{异常类型 (Exception Type)} + \\ \text{异常处理器 (Exception Handler)} + \text{异常后续处理方式 (Post Handling Action)}$$

EHAB 中的异常处理策略表达的是这样的意思：当出现某种类型的异常时，应该采用怎样的方式处理，以及在处理之后是否抛出原始异常或者处理后异常。EHAB 的异常处理机制是基于“类型”的，而异常处理逻辑则实现在一个个异常处理器中（Exception Handler）。异常后续处理方式主要分为三种情形：抛出原始异常、抛出处理后的异常和不做任何操作，这三种处理方式定义在 Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.PostHandlingAction 枚举中。

```
public enum PostHandlingAction
{
    None,
    NotifyRethrow,
    ThrowNewException
}
```

EHAB 异常处理策略可以通过配置的方式定义。下面的配置中演示了针对 `SqlException` 的处理, 我们通过 `LoggingExceptionHandler` 和 `ReplaceHandler` 这两个异常处理器先后对抛出的 `SqlException` 异常进行处理, 前者对抛出的异常信息进行日志记录, 后者将其替换成自定义的 `DbException`。当抛出的异常先后被这两个异常处理器处理之后, 将替换后的 `DbException` 抛出来 (`postHandlingAction="ThrowNewException"`)。

```
<configuration>
...
<exceptionHandling>
  <exceptionPolicies>
    <add name="data access policy">
      <exceptionTypes>
        <add type="System.Data.SqlClient.SqlException, System.Data"
          postHandlingAction="ThrowNewException" name="SqlException">
          <exceptionHandlers>
            <add name="Logging Handler"
              type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Logging.LoggingExceptionHandler"... />
            <add name="Replace Handler" type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.ReplaceHandler"
              exceptionMessage="Encounter data access error."
              replaceExceptionType="Artech.PetShop.Common.DbException, Artech.PetShop.Common"/>
          </exceptionHandlers>
        </add>
      </exceptionTypes>
    </add>
  </exceptionPolicies>
</exceptionHandling>
</configuration>
```

由于异常处理策略完全定义在配置中, 因此在编程的时候, 仅需要按照下面的方式指定相应的策略名称即可 (`Data Access Policy`)。关于 EHAB, 限于篇幅, 只能点到为止, 有兴趣的读者可以参阅微软企业库官方文档。

```
try
{
    return this.MembershipProxy.ValidateUser(username, password);
}
catch (Exception ex)
{
    if (ExceptionPolicy.HandleException(ex, "data access policy"))
    {
        throw;
    }
}
```


2. 基本原理介绍

在一个基于 WCF 的分布式应用中，服务端和客户端需要进行单独的异常处理。在服务端，让 EHAB 处理抛出的异常是很容易的，只需要按照上面的代码所示的方式调用 `ExcpetionPolicy` 的 `HandleException` 方法，传入抛出的异常并指定相应的异常处理策略名称即可。

客户端进行服务调用抛出的异常类型总是 `FaultException` (包括 `FaultException <TDetail>`)。而 EHAB 采用的是完全基于异常类型的处理方法，即抛出的异常类型决定了异常处理方式。也就是说，即使是两种完全不同的出错场景，只要抛出的异常具有相同的类型，EHAB 都会采用相同的方式来处理该异常。如果采用 EHAB，意味着只有唯一一种异常处理方式。

接下来介绍的解决方案通过一种变通的方式解决了上面的问题，它与通过 `ServiceDebugBehavior` 服务行为实现异常的传播有点类似。服务端抛出的异常先通过 EHAB 按照配置好的异常处理策略进行相应的处理。然后将处理后的异常相关的信息（包括异常类型的 `AssemblyQualifiedName`）封装到一个类似于 `ExceptionDetail` 的可序列化对象中，以该对象为基础创建 `MessageFault`，并进一步生成错误消息传回客户端。

客户端在接收到该错误消息后，提取服务端异常相关的信息，利用反射重建异常对象（已经明确了异常类型的 `AssemblyQualifiedName` 使异常对象的重建变成可能）并将其抛出。对于客户端的应用程序来说，就像是捕获从服务端抛出的异常一样了。通过 EHAB 针对客户端配置的异常处理策略对抛出的异常进行处理，这种异常处理方式依然是场景驱动的。

在本例中，我们通过如下一个名称为 `ServiceExceptionDetail` 的类型来封装异常相关信息。为了简单起见，直接让 `ServiceExceptionDetail` 继承自 `ExceptionDetail`。由于 `ServiceExceptionDetail` 对象需要从服务端向客户端传递，将其定义成数据契约。在 `ServiceExceptionDetail` 中仅仅定义了一个唯一的 `AssemblyQualifiedName` 属性，表示异常的类型、程序集有效名称，这是为了基于反射的异常重建的需要。在 `ServiceExceptionDetail` 中，定义了三个字符串常量表示对应 SOAP Fault 的 `SubCode` 名称和命名空间，以及对应错误消息的 `Action`。

```
using System;
using System.Runtime.Serialization;
using System.ServiceModel;
namespace Artech.EntLibIntegration
{
    [DataContract(Namespace = "http://www.artech.com/")]
    public class ServiceExceptionDetail : ExceptionDetail
    {
        public const string FaultSubCodeNamespace =
            "http://www.artech.com/exceptionhandling/";
        public const string FaultSubCodeName = "ServiceError";
        public const string FaultAction = "http://www.artech.com/fault";

        [DataMember]
        public string AssemblyQualifiedName { get; private set; }
    }
}
```

```

[DataMember]
public new ServiceExceptionDetail InnerException{ get; private set; }

public ServiceExceptionDetail(Exception ex)
    : base(ex)
{
    this.AssemblyQualifiedName = ex.GetType().AssemblyQualifiedName;
    if (null != ex.InnerException)
    {
        this.InnerException = new ServiceExceptionDetail
            (ex.InnerException);
    }
}

public override string ToString()
{
    return this.Message;
}
}
}

```

整个解决方法实现的原理大体上可以通过图 1-10 表示。有人觉得这和基于 ServiceDebugBehavior 服务行为向客户端暴露异常详细信息一样,异常信息会完全暴露给客户端,因而存在敏感信息泄露的危险。但是如果将敏感信息屏蔽的操作定义在相关的异常处理策略中,并通过 EHAB 来实现,那么最终传递给客户端的信息已经是经过处理的了。

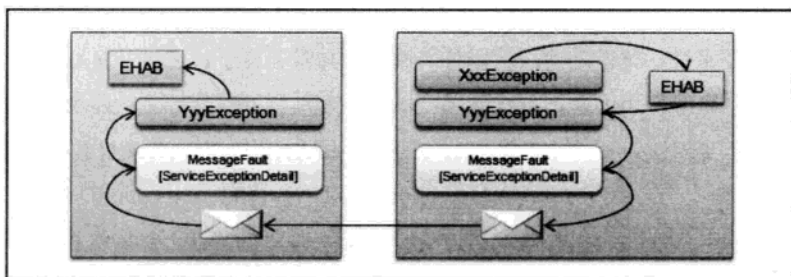


图 1-10 WCF 与 EHAB 集成的实现原理

3. 异常处理、封装与重建

从上面给出的整个解决方案实现原理介绍中,可以看出这个结构体系需要解决如下三个功能。

- 通过 EHAB 处理服务端抛出的原始异常 (XxxException): 利用 EHAB 针对预定义的异常处理策略对服务操作抛出的异常进行处理,处理后的异常表示为 YyyException (YyyException 可能就是抛出的异常 XxxException,也有可能是对 XxxException 的封装或者替换)。
- 通过 MessageFault 封装 EHAB 处理后的异常 (YyyException): 创建 ServiceExceptionDetail 对象封装通过 EHAB 处理后的异常,再借此创建 MessageFault 对象。最终针对 MessageFault 对象生成错误消息回复给客户端。

- 客户端实现异常的重建 (YyyException): 客户端接收到错误消息后, 提取异常相关信息并重建异常对象, 使得客户端可以利用 EHAB 针对基于客户端的异常处理策略对其进行相应的处理。

4. 自定义错误处理器实现基于 EHAB 的异常处理和封装

为了实现利用 EHAB 自动处理服务操作抛出的异常, 以及对处理后异常的封装和传递, 定义了如下一个名称为 `ServiceErrorHandler` 的自定义错误处理器。

```
using System;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling;
namespace Artech.EntLibIntegration
{
    public class ServiceErrorHandler : IErrorHandler
    {
        public string ExceptionPolicyName { get; private set; }

        public ServiceErrorHandler(string exceptionPolicyName)
        {
            this.ExceptionPolicyName = exceptionPolicyName;
        }

        public bool HandleError(Exception error)
        {
            return false;
        }

        public void ProvideFault(Exception error, MessageVersion version, ref Message fault)
        {
            if (typeof(FaultException).IsInstanceOfType(error))
            {
                return;
            }
            try
            {
                if (ExceptionPolicy.HandleException(error, this.ExceptionPolicyName))
                {
                    fault = Message.CreateMessage(version, BuildFault(error),
                        ServiceExceptionDetail.FaultAction);
                }
            }
            catch (Exception ex)
            {
                fault = Message.CreateMessage(version, BuildFault(ex),
                    ServiceExceptionDetail.FaultAction);
            }
        }

        private MessageFault BuildFault(Exception error)
        {
            ServiceExceptionDetail exceptionDetail = new
                ServiceExceptionDetail(error);
            FaultCode code = FaultCode.CreateReceiverFaultCode(
                ServiceExceptionDetail.FaultSubCodeName,
```

```

        ServiceExceptionDetail.FaultSubCodeNamespace);
    return MessageFault.CreateFault(code, new FaultReason(error.Message),
        exceptionDetail);
    }
}

```

`ServiceErrorHandler` 的只读属性 `ExceptionPolicyName` 表示服务端配置的异常处理策略的名称, 该属性在构造函数中指定。在 `ProvideFault` 方法中, 先判断抛出的异常是否是 `FaultException`, 如果是则不做处理 (在这种情况下, 一般是服务提供者人为抛出的, 并不希望再做进一步的处理)。否则调用 `ExceptionPolicy` 的 `HandleException` 方法, 传入异常处理策略名称, 对该异常进行处理。对于处理后的异常, 通过 `BuildFault` 方法创建 `ServiceExceptionDetail` 对象对异常信息进行封装, 并最终生成错误消息。

5. 通过服务行为应用自定义错误处理器

上面自定义的错误处理器 `ServiceErrorHandler` 最终通过如下定义的服务行为 `Exception HandlingBehaviorAttribute` 应用到信道分发器上。具体来说, 在 `ApplyDispatchBehavior` 方法中根据指定的异常策略名称创建 `ServiceErrorHandler` 对象, 并将其添加到当前 `ServiceHost` 的所有信道终结点的 `ErrorHandlers` 集合中。

```

using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
namespace Artech.EntLibIntegration
{
    public class ExceptionHandlingBehaviorAttribute:Attribute,IServiceBehavior
    {
        public string ExceptionPolicyName { get; private set; }
        public ExceptionHandlingBehaviorAttribute(string exceptionPolicyName )
        {
            this.ExceptionPolicyName = exceptionPolicyName;
        }
        public void AddBindingParameters(ServiceDescription serviceDescription,
            ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
            BindingParameterCollection bindingParameters){}
        public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
            ServiceHostBase serviceHostBase)
        {
            foreach (ChannelDispatcher channelDispatcher in
                serviceHostBase.ChannelDispatchers)
            {
                channelDispatcher.ErrorHandlers.Add(new
                    ServiceErrorHandler(this.ExceptionPolicyName));
            }
        }
        public void Validate(ServiceDescription serviceDescription,
            ServiceHostBase serviceHostBase){}
    }
}

```

从上面的代码中可以看到, 服务行为 `ExceptionHandlingBehaviorAttribute` 被定义成特性, 意味着我们可以以声明的形式将其应用到服务类型上。

6. 定义 `ExceptionHandler` 用于客户端异常处理

服务端抛出的异常经过自定义的错误处理器 `ServiceErrorHandler` 处理之后, 会抛出 `FaultException<ServiceExceptionDetail>` 异常并被客户端捕获。客户端需要做的就是根据 `ServiceExceptionDetail` 对象承载的错误详细信息重建被封装的异常, 并交由客户端配置的异常策略进行处理。为此定义了如下一个名为 `ExceptionHandler` 的静态帮助类。捕捉到的异常直接调用 `HandleException` 方法进行处理, 参数名称 `exceptionPolicy` 代表 EHAB 异常策略名称。

```
using System;
using System.ServiceModel;
using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling;
namespace Artech.EntLibIntegration
{
    public static class ExceptionHelper
    {
        public static bool HandleException(Exception ex, string exceptionPolicy)
        {
            FaultException<ServiceExceptionDetail> faultException = ex as
                FaultException<ServiceExceptionDetail>;
            if (faultException != null)
            {
                ex = GetException(faultException.Detail);
            }
            return ExceptionPolicy.HandleException(ex, exceptionPolicy);
        }
        public static Exception GetException(ServiceException
            Detail exceptionDetail)
        {
            Type exceptionType = Type.GetType(exceptionDetail.AssemblyQualifiedName);
            if (null == exceptionDetail.InnerException)
            {
                return (Exception)Activator.CreateInstance(exceptionType,
                    exceptionDetail.Message);
            }

            Exception innerException = GetException(exceptionDetail.
                InnerException);
            return (Exception)Activator.CreateInstance(exceptionType,
                exceptionDetail.Message, innerException);
        }
    }
}
```

7. 实例演示

接下来将上面定义的行为应用到真正的实例之中, 看看它们是否会按照我们之前希望的方式进行异常的处理。还是用我们熟悉的计算服务的例子。现在采用这样的异常处理策略: 服务端将所有运算操作过程中抛出的异常封装成 `CalculationException`, 而客户端将最终处理

过的异常(被封装的异常)的相关信息输出出来。`CalculationException` 被定义在 `Service.Interface` 项目中, 相关的定义代码如下所示。

```
using System;
namespace Artech.EntLibIntegration.Service.Interface
{
    [global::System.Serializable]
    public class CalculationException : Exception
    {
        public CalculationException() {}
        public CalculationException(string message) : base(message) {}
        public CalculationException(string message, Exception inner) :
            base(message, inner) {}
        protected CalculationException(
            System.Runtime.Serialization.SerializationInfo info,
            System.Runtime.Serialization.StreamingContext context)
            : base(info, context) {}
    }
}
```

服务端进行异常的封装可以通过现有的 `WrapHandler` 来实现, 而客户端对异常信息的显示通过如下的一个自定义异常处理器 `ErrorReportingHandler` 来实现。简单起见, 只通过控制台打印出异常的消息和类型。顺便提一下, 我们采用的是 `EntLib 4.1` 版本, 不是最新的 `5.0`, 两个版本中自定义异常处理器的方式是完全不同的。

```
using System;
using System.Text;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration.ObjectBuilder;
using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling;
using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Configuration;
using Microsoft.Practices.ObjectBuilder2;
namespace Artech.EntLibIntegration.Client
{
    [ConfigurationElementType(typeof(ErrorReportingHandlerData))]
    public class ErrorReportingHandler : IExceptionHandler
    {
        public Exception HandleException(Exception exception, Guid
            handlingInstanceId)
        {
            Console.WriteLine("{0} [{1}]", exception.Message,
                exception.GetType().FullName);
            return exception;
        }
    }
    [Assembler(typeof(ErrorReportingHandlerAssembler))]
    public class ErrorReportingHandlerData : ExceptionHandlerData
    {
        public ErrorReportingHandlerData() {}
        public ErrorReportingHandlerData(string name, Type type)
            : base(name, type) {}
    }
    public class ErrorReportingHandlerAssembler : IAssembler<IExceptionHandler,
        ExceptionHandlerData>
    {
        public IExceptionHandler Assemble(IBuilderContext context,
```

```

        ExceptionHandlerData objectConfiguration, IConfigurationSource
        configurationSource, ConfigurationReflectionCache reflectionCache)
    {
        ErrorReportingHandlerData handlerData =
            (ErrorReportingHandlerData)objectConfiguration;
        return new ErrorReportingHandler();
    }
}

```

我们将自定义异常处理器 `ErrorReportingHandler` 定义在 `Client` 项目中。除了异常处理器类之外，还需要定义配套的 `ErrorReportingHandlerData`（异常处理器的配置元素类）和 `ErrorReportingHandlerAssembler`（通过配置元素创建异常处理器）。需要为 `Client` 项目添加如下三个 `EntLib` 相关程序集引用，可以从 `EntLib` 官方网站上下载，也可以直接使用本例提供的源代码相应程序集。

- `Microsoft.Practices.EnterpriseLibrary.Common.dll`
- `Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.dll`
- `Microsoft.Practices.ObjectBuilder2.dll`

由于需要通过 `ServiceExceptionDetail` 对象对服务端抛出来的异常进行封装并通过错误消息传递给客户端，同时需要为错误消息指定一个固定值（作为常量定义在 `ServiceExceptionDetail` 类型中）作为 `<Action>` 报头，因此需要将 `ServiceExceptionDetail` 定义成服务契约 `ICalculator` 的 `Divide` 操作的错误契约并指定 `Action` 属性值。

```

[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract]
    [FaultContract(typeof(ServiceExceptionDetail), Action =
        ServiceExceptionDetail.FaultAction)]
    int Divide(int x, int y);
}

```

现在将上面定义的服务行为 `ExceptionHandlingBehaviorAttribute` 以声明的方式应用到 `CalculatorService` 上，并指定异常处理策略名称为“`service policy`”。

```

[ExceptionHandlingBehavior("service policy")]
public class CalculatorService : ICalculator
{
    public int Divide(int x, int y)
    {
        return x / y;
    }
}

```

名称为 `servicepolicy` 的异常处理策略定义在配置文件中。该异常策略仅针对 `DivideByZeroException` 异常进行处理，我们通过 `WrapHandler` 将抛出的 `DivideByZeroException` 异常封装成上面定义的 `CalculationException`，并且将封装的异常消息指定为“运算错误”。下面的 XML 片段给出了包含异常策略和服务终结点在内的整个服务端配置。

```

<configuration>
  <configSections>
    <section name="exceptionHandling"
type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Configuration.
ExceptionHandlingSettings,
Microsoft.Practices.EnterpriseLibrary.ExceptionHandling" />
  </configSections>
  <exceptionHandling>
    <exceptionPolicies>
      <add name="service policy">
        <exceptionTypes>
          <add type="System.DivideByZeroException, mscorlib"
            postHandlingAction="ThrowNewException" name="DivideByZeroException">
            <exceptionHandlers>
              <add exceptionMessage="运算错误"
wrapExceptionType="Artech.EntLibIntegration.Service.Interface.CalculationE
xception,Artech.EntLibIntegration.Service.Interface"
type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.WrapHandler,
Microsoft.Practices.EnterpriseLibrary.ExceptionHandling"
              name="Wrap Handler" />
            </exceptionHandlers>
          </add>
        </exceptionTypes>
      </add>
    </exceptionPolicies>
  </exceptionHandling>
</system.serviceModel>
<services>
  <service name="Artech.EntLibIntegration.Service.CalculatorService">
    <endpoint address="http://127.0.0.1:3721/calculatorservice"
      binding="ws2007HttpBinding"
      contract="Artech.EntLibIntegration.Service.Interface.
ICalculator"/>
  </service>
</services>
</system.serviceModel>
</configuration>

```

在客户端配置中定义了一个名称为 `clientpolicy` 的异常处理策略。该策略采用我们自定义的 `ErrorReportingHandler` 将所有捕获异常的消息打印出来。下面是包括异常策略和客户端终结点的整个客户端配置。

```

<configuration>
  <configSections>
    <section name="exceptionHandling"
type="Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Configuration.
ExceptionHandlingSettings, Microsoft.Practices.EnterpriseLibrary.
ExceptionHandling" />
  </configSections>
  <exceptionHandling>
    <exceptionPolicies>
      <add name="client policy">
        <exceptionTypes>
          <add type="System.Exception, mscorlib" postHandlingAction="None"

```



```

        name="System.Exception">
    <exceptionHandlers>
        <add type="Artech.EntLibIntegration.Client.ErrorReportingHandler,
            Artech.EntLibIntegration.Client"
            name="ErrorReportingHandler" />
    </exceptionHandlers>
    </add>
    </exceptionTypes>
    </add>
    </exceptionPolicies>
    </exceptionHandling>
    <system.serviceModel>
    <client>
        <endpoint name="calculatorservice"
            address="http://127.0.0.1:3721/calculatorservice"
            binding="ws2007HttpBinding"
            contract="Artech.EntLibIntegration.Service.Interface.
                ICalculator"/>
    </client>
    </system.serviceModel>
    </configuration>

```

客户端对进行服务调用抛出的异常进行处理时需要用到上面定义的帮助类 `ExceptionHandler`。服务调用代码如下所示，我们将服务调用放在 `try/catch` 块中，通过调用 `ExceptionHandler` 的 `HandleException` 方法对抛出的异常进行调用。

```

using System;
using System.ServiceModel;
using Artech.EntLibIntegration.Service.Interface;
using Artech.EntLibIntegration;
namespace Artech.WcfServices.Client
{
    class Program
    {
        static void Main(string[] args)
        {
            using (ChannelFactory<ICalculator> channelFactory = new
                ChannelFactory<ICalculator>("calculatorservice"))
            {
                ICalculator calculator = channelFactory.CreateChannel();
                using (calculator as IDisposable)
                {
                    try
                    {
                        int result = calculator.Divide(1, 0);
                    }
                    catch (Exception ex)
                    {
                        if (ExceptionHandler.HandleException(ex, "client policy"))
                        {
                            throw;
                        }
                    }
                }
            }
        }
    }
}

```

```
        }  
    }  
    }  
    Console.Read();  
}  
}
```

由于我们将第二个参数指定成上面定义的异常策略的名称 ("client policy") 作为 `HandleException` 的第二个参数, 所以将会按照对应的策略进行异常的处理, 即直接利用 `ErrorReportingHandler` 将异常信息打印出来。因此, 当运行整个实例的时候, 客户端控制台会出现如下的结果, 这充分证明了服务端对异常的封装和客户端对异常信息的显示都能够按照我们定义的异常策略正常进行。

运算错误[Artech.EntLibIntegration.Service.Interface.CalculationException]



第2章 元数据 (Metadata)

客户端和服务端借助于终结点进行通信。服务的提供者通过一个或者多个终结点将服务发布出来，服务的消费者则通过创建与之匹配的终结点进行服务的调用。站在服务消费者的角度，这样一个“匹配”的终结点该如何创建呢？或者说客户端基于何种信息才能够有效调用目标服务的终结点呢？这就是元数据需要解决的问题。



可以将服务的元数据看成是它的所有终结点的描述。它以一种易于交换的数据格式 (WSDL、XSD 或者 WS-Policy 等) 描述该服务的所有终结点信息。WCF 为我们提供了一个完整的元数据架构体系, 使我们很容易地进行元数据的导出 (Exporting)、发布 (Publishing)、获取 (Retrieving) 和导入 (Importing)。

2.1 WCF 元数据架构体系简介

服务的元数据实际上是对它具有的所有终结点的描述。终结点由地址 (Address)、绑定 (Binding) 和契约 (Contract) 三要素组成。地址决定了服务的位置并实现相应的寻址机制, 契约描述了消息交换模式 (Message Exchange Patter, MEP) 及消息的结构 (Schema), 绑定则通过创建信道栈实现对消息的编码、传输和基于某些特殊的功能 (比如事务、可靠传输及安全传输等) 对消息进行相应的处理。服务的消费者通过获取用于描述服务端的元数据, 并在此基础上重建客户端终结点, 才能确保:

- 请求消息被发送到正确的目标地址;
- 使用一致的传输协议以实现消息的正常传输;
- 采用服务端期望的消息交换模式;
- 发送的消息具有能够识别的格式;
- 使用相匹配的消息编码方式以确保服务端能够对接收到的消息进行正常解码;
- 对消息进行与服务端一致性的处理以确保对事务、可靠传输、消息安全等协议的实现。

WCF 是基于 SOA 的分布式通信平台, 而 SOA 的一个重要特性就是实现跨平台互操作性。因为元数据本身采用一种开放的标准来表示, 所以能够确保服务消费者正常调用可能位于异质平台上的服务。目前元数据具有如下三种比较典型的描述方式。

- XSD: 通过 XML Schema 的形式描述组成消息的结构;
- WSDL: 通过一个完整的 Web Service Description Language 文档对服务进行全面的描述;
- WS-Policy 策略: 通过 WS-Policy 规范以断言 (Assertion) 形式对服务能力和特性进行描述。

跨平台不仅仅要求承载服务描述信息的元数据本身采用一种开放的标准或者规范来表示, 同时要求元数据交换 (Metadata Exchange, MEX) 也要按照共同遵守的规范来进行。在 WS-* 规范体系中, WS-Metadata Exchange (以后简称 WS-MEX) 对元数据的交换进行了标准化的规范。

2.1.1 WS-MEX

WS-MEX 是一个关于如何进行元数据交换的 WS 规范。它和其他的 WS-* 规范 (比如 WSDL、WS-Addressing、WS-Transfer 和 WS-Policy 等) 组成了一个完整的描述 Web 服务元数据和元数据交换的规范体系。在正式介绍 WS-MEX 之前, 先来大概了解一下这些辅助性 WS-* 规范。

1. WS-Policy

一个 Web 服务除了实现通过服务契约定义的业务功能之外，还需要具有一些与业务无关的行为（Behavior）和能力（Capability），比如事务流转、可靠消息传输和传输安全等，我们可以将其统称为 Web 服务的策略（Policy）。WS-Policy 提供了一个基于 XML 的框架模型和语法，用于描述 Web 服务的能力、要求和行为属性。WS-Policy 属于 WS-* 体系中的一个基础性规范，为其他的 WS 规范（比如 WS-AT、WS-RM 和 WS-Security 等）提供一种统一的策略描述。

W3C 先后在 2006 年和 2007 年推出了 WS-Policy 1.2 和 WS-Policy 1.5。在这里仅针对 WS-Policy 1.5 简单地介绍一下一个完整的 WS 策略具有怎样的结构。对于希望深入了解 WS-Policy 的读者，可以直接下载官方文档（<http://www.w3.org/TR/ws-policy/>）。

WS-Policy 采用一个基于 XML 的策略表达式（Policy Expression）来表示一个策略。下面的 XML 片段表示的策略表达式服务于 WS-Security，这是一个基于如何实现消息安全的 WS 规范。这个策略表达式体现的含义是：需要对消息的主体部分采用签名或者加密。

```
(01) <wsp:Policy
      xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
      xmlns:wsp="http://www.w3.org/ns/ws-policy" >
(02)   <wsp:ExactlyOne>
(03)     <wsp:All>
(04)       <sp:SignedParts>
(05)         <sp:Body/>
(06)       </sp:SignedParts>
(07)     </wsp:All>
(08)     <wsp:All>
(09)       <sp:EncryptedParts>
(10)         <sp:Body/>
(11)       </sp:EncryptedParts>
(12)     </wsp:All>
(13)   </wsp:ExactlyOne>
(14) </wsp:Policy>
```

策略体现在对 Web 服务相关实体的要求（Requirements）、能力（Capabilities）和特性（Characteristics）等行为属性的描述。WS-Policy 通过断言（Assertion）的形式来表示这些单一的行为属性，然后通过一定的规则将相关的策略断言有机地组合在一起，以实现对整个 Web 服务目标实体的完整描述。

（1）策略表达式（Policy Expression）

一个策略表达式通过一个 XML 信息集（XML InfoSet）描述一个完整的策略。策略表达式具有两种表示形式：标准形式（Normal Form）和简写形式（Compact Form）。一个策略表达式是一个策略选择项（Policy Alternative）的集合。要求满足某个策略，意味着需要满足该集合的至少一个策略选择项。

(2) 策略选择项 (Policy Alternative)

一般情况下, 策略往往是承载一些确保服务正常调用 (说得更加具体一点, 就是确保 Web 服务终结点能够正常交互) 的条件信息。服务的消费者在进行正常的服务调用之前, 需要保证满足这些必备的前提条件。对于服务提供者来说, 针对某个具体的应用场景会提供一个到多个不同的选择项。比如我们上面给出的例子, 对消息主体部分的保护具有两种可选的方式: 加密和签名。

这些单一的选择项被称为策略选择项 (Policy Alternative)。对于上面给出的策略表达式, (03) ~ (07) 和 (08) ~ (12) 定义两个策略选择项, 代表对消息主体进行签名还是加密。一个策略选择项由零到多个策略断言通过相应的策略操作符组合在一起。

(3) 策略断言 (Policy Assertion)

Web 服务实体的某个单一行为属性通过一个策略断言表示。而一个策略断言由一个必需的断言类型 (Assert Type) 和一组可选的断言参数 (Assert Parameter) 组成。断言类型通过一个有效名称 (Qualified Name, QName, 即命名空间和本地名称的组合) 表示。上面的策略表达式中定义了两个策略断言, 其断言类型分别为: `sp:SignedParts` ((04) ~ (06)) 和 `sp:EncryptedParts` ((09) ~ (11))。

一个最简单的策略断言可以仅仅由一个包含断言类型的空 XML 元素构成, 也可以为它添加一些属性 (Attribute) 和子元素, 我们把这些策略辅助描述信息称为断言参数。

一个策略断言可以很简单 (一个空 XML 元素), 也可以定义得很复杂 (一个结构复杂的 XML 元素), 这取决于具体的策略描述对象。一个比较极端的策略断言是将一个完整的策略表达式作为其子元素, 我们把这种情况称为策略断言嵌套 (Policy Assertion Nesting)。嵌套的策略断言的结构可以通过下面的 XML 表示。

```
(01) <Assertion ...>
(02)   ...
(03)   ( <wsp:Policy ...> ... </wsp:Policy> )?
(04)   ...
(05) </Assertion>
```

(4) 策略操作符 (Policy Operator)

一个策略选择项由零 (也就是说可以定义空的策略选择项) 到多个策略断言通过一定的规则构成, 策略断言的组合规则通过策略操作符来体现。策略操作符体现的是这样一种含义: 请求者采用怎样的方式去满足构成策略选择项的所有策略断言——需要满足所有的断言呢, 还是仅仅需要满足其中某一个。

WS-Policy 定义了两种主要的策略操作符 (实际上 Policy 本身就属于一个策略操作符 `wsp:Policy`): `ExactlyOne` 和 `All`。根据名称不难猜出, `ExactlyOne` 表示仅仅需要满足断言集

合的某一个元素即可，而 All 意味着必须满足断言集合中的所有元素。由于 ExactlyOne 表示的是“满足其中之一”的意思，这和策略和策略选择项之间体现的关系吻合，因此在标准形式的策略表达式中，所有的策略选择项均纳入到 ExactlyOne 操作符之中。

整个策略表达式的结构（策略、策略选择项和策略断言之间的关系）大体可以通过图 2-1 表示。关于 WS-Policy 中对策略表达式的规定，还有其他一些额外的内容，比如策略的识别（Policy Identifying）、策略简写形式（Compact Form）等，在这里就不再一一介绍了，有兴趣的读者可以下载 WS-Policy 1.5 的官方文档（<http://www.w3.org/TR/ws-policy/>）。

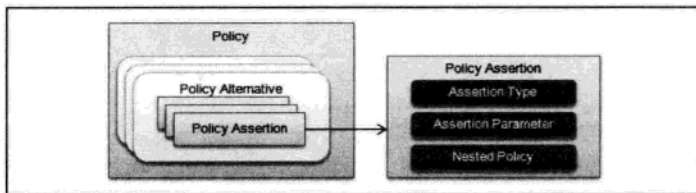


图 2-1 策略表达式结构图

2. WS-Transfer

在 Web 服务的世界中，很多资源（Resource）都可以通过 WS-Addressing 规定的寻址方式进行定位和引用。而 WS-Transfer 则为这些可寻址（Addressable）的 Web 服务资源的基本操作提供了统一规范。具体来说，WS-Transfer 提供了一种基于 SOAP 的标准方式去实现对这些资源的获取、更新、删除和创建。

WS-Transfer 的主要内容集中在对 4 个基本资源操作描述上面，即 Get、Put、Delete 和 Create，分别实现对资源的获取、更新、删除和创建。W3C 分别在 2006 年 3 月和 9 月先后推出了两个版本的 WS-Transfer。接下来的介绍完全基于最新版本的 WS-Transfer。

（1）资源的获取（Get）

请求者向目标地址发送 Get 请求以获取相应的资源，这样的请求消息具有如下的格式。其中<Action>报头的值必须是 <http://schemas.xmlsoap.org/ws/2004/09/transfer/Get>，而消息的主体部分为空。

```
<s:Envelope...>
  <s:Header ...
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/Get
    </wsa:Action>
    <wsa:MessageID>xs:anyURI</wsa:MessageID>
    <wsa:To>xs:anyURI</wsa:To>
    ...
  </s:Header>
  <s:Body ...>
    ...
  </s:Body>
</s:Envelope>
```

资源服务接受 Get 请求后, 将被请求的资源封装到 GetResponse 消息中并予以回复。GetResponse 消息具有如下的结构。<Action>报头的值为 http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse。代表请求资源的 XML 必须作为消息主体的第一个子元素。

```
<s:Envelope ...>
  <s:Header ...>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse
    </wsa:Action>
    <wsa:RelatesTo>xs:anyURI</wsa:RelatesTo>
    <wsa:To>xs:anyURI</wsa:To>
    ...
  </s:Header>
  <s:Body ...>
    xs:any
    ...
  </s:Body>
</s:Envelope>
```

(2) 资源的更新 (Put)

请求者向目标地址发送 Put 请求以更新相应的资源, 这样的请求消息具有如下的格式。Put 请求消息的<Action>报头为 http://schemas.xmlsoap.org/ws/2004/09/transfer/Put。表示更新资源的 XML 必须作为消息主体的第一个子元素。

```
<s:Envelope ...>
  <s:Header...>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/Put
    </wsa:Action>
    <wsa:MessageID>xs:anyURI</wsa:MessageID>
    <wsa:To>xs:anyURI</wsa:To>
    ...
  </s:Header>
  <s:Body...>
    xs:any
    ...
  </s:Body>
</s:Envelope>
```

资源服务接受 Put 请求后, 回复一个具有如下格式的 PutResponse 消息。<Action>报头值为 http://schemas.xmlsoap.org/ws/2004/09/transfer/PutResponse。如果资源服务完全采用请求者提供的资源对现有的目标资源进行更新, 那么回复消息的主体部分为空, 否则将更新后的资源以 XML 的形式置于 PutResponse 消息主体部分的第一个子元素中。

```
<s:Envelope ...>
  <s:Header ...>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/PutResponse
    </wsa:Action>
    <wsa:RelatesTo>xs:anyURI</wsa:RelatesTo>
    <wsa:To>xs:anyURI</wsa:To>
    ...
  </s:Header>
```



```
<s:Body ...>
  xs:any ?
</s:Body>
</s:Envelope>
```

(3) 资源的删除 (Delete)

请求者向目标地址发送 Delete 请求以删除相应的资源, 这样的请求消息具有如下的格式。Delete 消息的<Action>报头为 http://schemas.xmlsoap.org/ws/2004/09/transfer/Delete, 而主体部分为空。

```
<s:Envelope ...>
  <s:Header ...>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/Delete
    </wsa:Action>
    <wsa:MessageID>xs:anyURI</wsa:MessageID>
    <wsa:To>xs:anyURI</wsa:To>
    ...
  </s:Header>
  <s:Body .../>
</s:Envelope>
```

资源服务接受 Delete 请求后, 回复一个具有如下结构的 DeleteResponse 消息。消息的<Action>报头为 http://schemas.xmlsoap.org/ws/2004/09/transfer/DeleteResponse, 而主体部分为空。

```
<s:Envelope ...>
  <s:Header ...>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/DeleteResponse
    </wsa:Action>
    <wsa:RelatesTo>xs:anyURI</wsa:RelatesTo>
    <wsa:To>xs:anyURI</wsa:To>
    ...
  </s:Header>
  <s:Body ...> ... </s:Body>
</s:Envelope>
```

(4) 资源的创建 (Create)

请求者向目标地址发送 Create 请求以创建新的资源。Create 请求消息具有如下的格式。消息的<Action>报头为 http://schemas.xmlsoap.org/ws/2004/09/transfer/Create。新的资源内容以 XML 的形式作为消息主体部分的第一个子元素。

```
<s:Envelope ...>
  <s:Header ...>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/Create
    </wsa:Action>
    <wsa:MessageID>xs:anyURI</wsa:MessageID>
    <wsa:To>xs:anyURI</wsa:To>
    ...
  </s:Header>
  <s:Body ...>
```

```

    xs:any
    ...
  </s:Body>
</s:Envelope>

```

资源服务接受 Create 请求后, 回复一个具有如下格式的 CreateResponse 消息。消息的 <Action> 报头为 `http://schemas.xmlsoap.org/ws/2004/09/transfer/CreateResponse`。新创建资源的终结点引用 (Endpoint Reference) 作为回复消息主体部分的第一个子元素。如果最终被更新的资源内容和请求者提供的不一致, 被更新的资源内容需要作为回复消息主体部分的第二个子元素返回。

```

<s:Envelope ...>
  <s:Header ...>
    <wsa:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/CreateResponse
    </wsa:Action>
    <wsa:RelatesTo>xs:anyURI</wsa:RelatesTo>
    <wsa:To>xs:anyURI</wsa:To>
    ...
  </s:Header>
  <s:Body ...>
    <wxf:ResourceCreated>endpoint-reference</wxf:ResourceCreated>
    xs:any?
  </s:Body>
</s:Envelope>

```

3. WSDL

WSDL 全称为 Web 服务描述语言 (Web Service Description Language), 是采用 XML 的形式对 Web 服务的描述。WSDL 将一个 Web 服务定义成一组终结点的集合, 而每一个终结点包含一系列基于消息 (Message) 的操作 (Operation)。这些抽象的操作和消息最终和相应的协议及消息格式绑定。

虽然 W3C 在 2007 年 6 月就正式出台了 WSDL 2.0 版本, 并将其作为官方推荐版本, 但是该版本并没有得到广泛的推广。如今, WCF 完全支持的还是 WSDL 1.1 版本, 所以接下来将针对这个版本对 WSDL 进行简单的介绍。根据 WSDL 描述对象的性质, 大体可以将所有 WSDL 的元素划分为以下两类。

- 抽象元素: 比如通过 XSD 表示的数据类型, 用于承载数据信息的消息, 通过对关联的消息按照某种消息交换模式组合而成的操作等。
- 具体元素: 比如将相应的操作和具体的网络协议和消息格式进行绑定等。

为了有效地了解 WSDL 的结构, 来看看如下一段直接从官方文档上复制出来的 WSDL 文档。

```

<definitions name="StockQuote"
  targetNamespace="http://example.com/stockquote.wsdl"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xsd="http://example.com/stockquote.xsd"

```

```

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
  <schema targetNamespace="http://example.com/stockquote.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="TradePriceRequest">
      <complexType>
        <all>
          <element name="tickerSymbol" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="TradePrice">
      <complexType>
        <all>
          <element name="price" type="float"/>
        </all>
      </complexType>
    </element>
  </schema>
</types>
<message name="GetLastTradePriceInput">
  <part name="body" element="xsd1:TradePriceRequest"/>
</message>
<message name="GetLastTradePriceOutput">
  <part name="body" element="xsd1:TradePrice"/>
</message>
<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetLastTradePrice">
    <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="StockQuoteService">
  <documentation>My first service</documentation>
  <port name="StockQuotePort" binding="tns:StockQuoteBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
</service>
</definitions>

```

可以看出它由 5 个子元素构成: Type、Message、PortType、Binding 和 Service。这五大元素构成了一个完整的 WSDL 文档。

(1) Type (通过 XSD 表示的数据类型)

WSDL 直接采用 XSD 作为数据类型定义的语言。上面的 WSDL 文档通过 XSD 定义了 TradePriceRequest 和 TradePrice 两个 XML 元素。不难看出这两个类型分别是字符串和浮点数类型。

```
<types>
  <schema targetNamespace="http://example.com/stockquote.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="TradePriceRequest">
      <complexType>
        <all>
          <element name="tickerSymbol" type="string"/>
        </all>
      </complexType>
    </element>
    <element name="TradePrice">
      <complexType>
        <all>
          <element name="price" type="float"/>
        </all>
      </complexType>
    </element>
  </schema>
</types>
```

(2) Message (通信数据的载体)

Web 服务采用基于消息的通信方式, 所以消息是通信数据的载体。WSDL 的 <message> 元素用于定义作为服务操作输入和输出的消息格式。WSDL 的消息是一个具有唯一标识 (通过 Name 属性) 的 XML 元素, 通常利用 Types 节点中定义的数据类型来描述其结构。上面的 WSDL 定义了 GetLastTradePriceInput 和 GetLastTradePriceOutput 两个消息。

```
<message name="GetLastTradePriceInput">
  <part name="body" element="xsd1:TradePriceRequest"/>
</message>
<message name="GetLastTradePriceOutput">
  <part name="body" element="xsd1:TradePrice"/>
</message>
```

(3) PortType (服务操作的集合)

一个服务逻辑上由一到多个关联的操作组成, 而操作体现在按照某种模式进行的消息交换。WSDL 的 PortType 表示的是服务操作的集合, 反映在 XML 结构上, 就是一组 <operation> 元素的集合。每一个 <operation> 元素代表一个单一的操作, 它通过一个或两个消息组合而成。消息的不同组合方式反映了操作采用的不同消息交换模式。上面给出的 WSDL 通过如下的 XML 片段定义了一个仅包含一个操作的 PortType。

```
<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"/>
  </operation>
</portType>
```

当 WCF 服务通过 WSDL 的形式发布出来后, 服务契约对应的部分就是 PortType。WCF 支持三种典型的消息交换模式, 即单工 (One-way)、请求-回复 (Request-Reply) 和双工 (Duplex)。而双工模式实际上是由前面两种模式组合而成的, 单工 (One-way) 和请求-回复 (Request-Reply) 模式才是基本消息交换模式。除了这两种基本模式, WSDL 还对另外两种消息交换模式提供支持, 即恳请-回复 (Solicit-Response) 和通知 (Notification) 模式。

PortType 中的每一个操作均由输入 (Input) 和输出 (Output) 消息的不同组合方式定义, 而这种对输入、输出消息的不同组合就是对某种消息交换模式的反映。接下来站在服务端终结点的角度, 来介绍上述 4 种消息交换模式。

单工消息交换模式下, 终结点仅接收来自客户端的请求。所以单工操作仅包含一个 <input> 消息, 在 WSDL 中的表示如下。

```
<wsdl:definitions ... >
  <wsdl:portType ... >*
    <wsdl:operation name="nmtoken">
      <wsdl:input name="nmtoken"? message="qname"/>
    </wsdl:operation>
  </wsdl:portType >
</wsdl:definitions>
```

请求-回复消息交换模式下, 终结点接收来自客户端的请求, 并向对方发送回复消息。请求-回复操作通过输入 <input> 和 <output> 消息的有序组合表示, 在 WSDL 中的表示如下。

```
<wsdl:definitions ...>
  <wsdl:portType ...>*
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:input name="nmtoken"? message="qname"/>
      <wsdl:output name="nmtoken"? message="qname"/>
      <wsdl:fault name="nmtoken" message="qname"/>*
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

恳请-回复模式和请求-回复模式正好相反。在这种模式下, 终结点先向客户端发送请求, 并接收来自客户端的回复。恳请-回复操作由 <output> 消息和 <input> 消息的有序组合表示, 在 WSDL 中的表示如下。

```
<wsdl:definitions ...>
  <wsdl:portType ...>*
    <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
      <wsdl:output name="nmtoken"? message="qname"/>
      <wsdl:input name="nmtoken"? message="qname"/>
      <wsdl:fault name="nmtoken" message="qname"/>*
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

在通知消息交换模式下, 终结点仅向客户端发送请求, 这和单工模式正好相反。通知操作由单一的 <output> 消息组成, 在 WSDL 中的表示如下。

```

<wsdl:definitions ...>
  <wsdl:portType ...> *
    <wsdl:operation name="nmtoken">
      <wsdl:output name="nmtoken"? message="qname"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>

```

(4) Bindings (消息、操作与协议、格式的绑定)

上面介绍 WSDL 的三个元素时主要从抽象的角度对数据类型、消息和操作进行描述。要创建服务于具体消息交换场景的终结点，还需要将这些抽象的描述和具体的消息格式 (Format) 和网络协议 (比如 SOAP、HTTP-GET 和 MIME 等) 进行绑定。

很有必要强调“终结点”。本节提到的终结点在大部分场景中都是指与技术无关的、用于进行消息交换的“端口”。而 WCF 中提到的终结点，可以看成是这样一个通用的终结点在具体技术平台中的实现。WCF 的终结点由地址、绑定和契约构成。结合 WSDL 不难看出，Type、Message 和 PortType 是对契约的描述，而绑定实现了抽象的描述和具体的协议 (网络传输协议、SOAP 和 WS-*规范等) 之间的绑定。所以 WCF 中的绑定和 WSDL 中的 Bindings 元素是对等的。

在 WSDL 中，可以通过很多绑定扩展实现与某种协议的绑定，最常见的是基于 SOAP 1.1 和 SOAP 1.2 的绑定。上面给出的 WSDL 中定义了一个典型的基于 SOAP 1.1 的绑定。

```

<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetLastTradePrice">
    <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

```

以上面这个 SOAP 绑定为例，一个绑定通过 name 属性 (name="StockQuoteSoapBinding") 定义其唯一标识，并通过 type 属性和定义一个 PortType (type="tns:StockQuotePortType") 进行关联。该 SOAP 绑定定义了消息采用的风格 (style="document"，另外一个选项是“rpc”) 和传输协议 (transport="http://schemas.xmlsoap.org/soap/http")。在操作级别，定义了操作反映在 <Action> 报头的值 (soapAction="http://example.com/GetLastTradePrice")，以及输入和输出消息的主体部分采用的编码方式 (use="literal")。

(5) Service (相关终结点的集合)

由于一个 Web 服务最终以终结点的方式暴露出来，因此 WSDL 最终体现在对终结点集合的描述。这里介绍的 WSDL 最后一个元素 <Service/> 本质上就是对基于该 Web 服务的一组

相关终结点的定义。我们照例将上面给出的 WSDL 的 Service 相关部分提取出来，根据具体的例子分析 Service 节点应有的结构。

```
<service name="StockQuoteService">
  <documentation>My first service</documentation>
  <port name="StockQuotePort" binding="tns:StockQuoteBinding">
    <soap:address location="http://example.com/stockquote"/>
  </port>
</service>
```

<service>通过 name 属性 (name="StockQuoteService") 定义了服务的唯一标识。该节点具有一个可选的<documentation>元素，可以通过单纯的文本或者 XML 为该服务定义一些可读的说明性的描述。<Service>节点最重要的是一组代表着一个终结点的<port>元素集合。<port>元素通过 name 和 binding 属性分别定义终结点的名称和引用相应的绑定。通过相应的绑定扩展定义终结点的地址。在这里终结点的地址通过 SOAP 绑定定义 (<soap:address location="http://example.com/stockquote"/>)。

图 2-2 反映了 WSDL 5 个元素之间的关系。对于表示一个终结点对象的<port>元素来说，它具有一个地址 (Address)，关联着一个绑定 (Binding)，而绑定对象关联着一个 PortType。一个 PortType 实际上对应着 WCF 中的契约 (Contract)。所以 WCF 下的终结点由地址、绑定和契约三要素组成在这里也得到了进一步的反映。实际上 WCF 本身就是按照 WS 开放标准设计的。

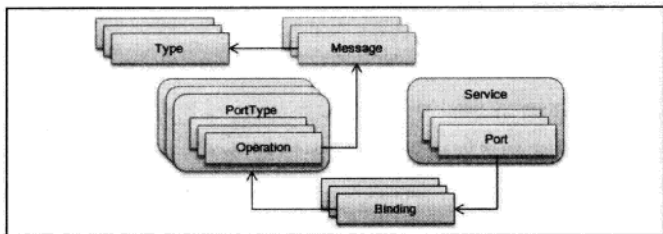


图 2-2 WSDL 5 个元素之间的引用关系

4. WS-MEX

WCF 的元数据架构体系构建在一个开放的标准之上，这个标准就是 WS-Metadata Exchange，简称 WS-MEX。WS-MEX 是 WS-*大家庭中的一名重要成员，最新的版本是 1.1。WS-MEX 在下面几个方面制定了相应的规范。

- 如何将基于 Web 终结点的元数据表示成一个 WS-Transfer 资源？
- 如何将元数据内嵌于 WS-Addressing 的终结点引用 (Endpoint Reference) 中？
- 如何获取某个 Web 服务终结点的元数据？

WS-MEX 的主要目的在于规范元数据的获取，它提供了如下两种不同的方式去获取 Web 服务终结点的元数据，即 WS-Transfer Get 和 Get Metadata。

(1) WS-Transfer Get

通过前面的介绍,我们知道了 WS-Transfer 旨在规范如何获取、更新、删除和创建 Web 服务资源。元数据本身就可以作为一种典型的 Web 服务资源,那么采用 WS-Transfer 无疑是一种最直接的选择。

元数据的提供者将元数据作为一种 Web 服务资源通过一个基于 WS-Transfer 的终结点暴露出来。请求者向该终结点发送 WS-Transfer Get 请求,以回复消息的形式获得所需的元数据。下面就是一个典型的基于 SOAP 1.1 的 WS-Transfer Get 请求消息,请求的目标地址就是发布元数据资源的终结点。

```
<s11:Envelope xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsa10="http://www.w3.org/2005/08/addressing">
  <s11:Header>
    <wsa10:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/Get
    </wsa10:Action>
    <wsa10:To>
      http://services.example.org/stockquote/metadata</wsa10:To>
    <wsa10:ReplyTo>
      <wsa10:Address>
        http://client.example.org</wsa10:Address>
      </wsa10:ReplyTo>
    <wsa10:MessageID>
      urn:uuid:1cecl21a-82fe-41da-87e1-3b23f254f128
    </wsa10:MessageID>
  </s11:Header>
  <s11:Body />
</s11:Envelope>
```

针对该 WS-Transfer Get 元数据请求,可能会得到如下一个标准的 WS-Transfer GetResponse 消息,而请求的元数据被置于 SOAP 消息的主体部分。

```
<s11:Envelope xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsa10="http://www.w3.org/2005/08/addressing"
  xmlns:mex="http://schemas.xmlsoap.org/ws/2004/09/mex"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <s11:Header>
    <wsa10:Action>
      http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse
    </wsa10:Action>
    <wsa10:To>
      http://client.example.org
    </wsa10:To>
    <wsa10:RelatesTo>
      urn:uuid:1cecl21a-82fe-41da-87e1-3b23f254f128
    </wsa10:RelatesTo>
  </s11:Header>
  <s11:Body>
    <mex:Metadata>
      <mex:MetadataSection Dialect="http://schemas.xmlsoap.org/wsdl/">
        <wsdl:definitions name="StockQuote" ...>
          ...
        </wsdl:definitions>
      </mex:MetadataSection>
    </mex:Metadata>
  </s11:Body>
```



```

</mex:MetadataSection>
<mex:MetadataSection
  Dialect="http://www.w3.org/2001/XMLSchema"
  Identifier="http://services.example.org/stockquote/schemas">
  <mex:Location>
    http://services.example.org/stockquote/schemas
  </mex:Location>
</mex:MetadataSection>
<mex:MetadataSection
  Dialect="http://schemas.xmlsoap.org/ws/2004/09/policy"
  Identifier="http://services.example.org/stockquote/policy">
  <mex:MetadataReference>
    <wsa10:Address>
      http://services.example.org/stockquote/policy
    </wsa10:Address>
  </mex:MetadataReference>
</mex:MetadataSection>
</mex:Metadata>
</s11:Body>
</s11:Envelope>

```

从上面给出的 WS-Transfer GetResponse 消息中不难发现获取到的元数据包含在 <Metadata> 节点中。<Metadata> 节点实际上是一个 <MetadataSection> 元素的集合，具体的元数据就定义在相应的 <MetadataSection> 节点下。元数据的表现形式通过 Dialect 属性定义，被称为元数据方言。

在本章开始的时候，我们就谈到 Web 服务终结点元数据具有三种典型的表现形式：WSDL、XSD 和 WS-Policy。再看看上面给出的包含元数据的 SOAP 消息中，<Metadata> 节点下三个 <MetadataSection> 就分别对应这三种形式的元数据。

- 第一个 <MetadataSection> 通过内联的方式直接嵌入一个 WSDL 文档。
- 第二个 <MetadataSection> 以地址的方式指定了一个 XML Schema。
- 第三个 <MetadataSection> 以终结点引用的方式指定了一个 WS-Policy 策略。

<MetadataSection> 的 Dialect 以一个 URI 的形式指明了元数据体现的形式，即元数据方言 (Dialect)。图 2-3 展现了 Metadata、MetadataSection 及这三种典型元数据方言之间的关系。在 WS-MEX 中为以下 5 种方言定义了相应的 URI。

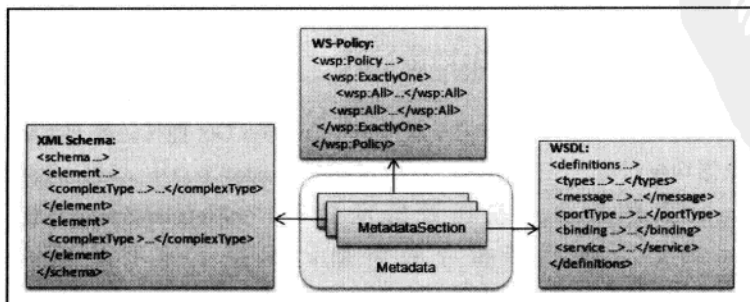


图 2-3 Metadata、MetadataSection 与三种典型的元数据方言之间的关系

- XML Schema: <http://www.w3.org/2001/XMLSchema>
- WSDL: <http://schemas.xmlsoap.org/wsdl/>
- WS-Policy: <http://schemas.xmlsoap.org/ws/2004/09/policy>
- WS-Policy Attachment: <http://schemas.xmlsoap.org/ws/2004/09/policy/attachment>
- MEX: <http://schemas.xmlsoap.org/ws/2004/09/mex>

(2) Get Metadata

通过 WS-Transfer Get 的方式获取 Web 服务元数据的前提是直接将元数据本身作为一个独立的可被寻址的 Web 服务资源。但是对于某些特殊的场景,这种方式不太适用。比如多个元数据资源关联到同一个元数据终结点,希望通过向该终结点发送请求获取所有相关的元数据。此外,并不是在任何情况下都能将终结点的元数据作为一个可以被寻址的 Web 服务资源。为了在这些场景中解决元数据的获取,WS-MEX 提出了另一种替换的元数据获取方式: Get Metadata。Get Metadata 操作请求的 SOAP 消息具有如下的结构要求(? 表示 0 或 1 个前置元素):

```
[action]
http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadata/Request
[Body]
<mex:GetMetadata ...>
  (<mex:Dialect>xs:anyURI</mex:Dialect>
  (<mex:Identifier>xs:anyURI</mex:Identifier>)?
  )?
</mex:GetMetadata>
```

当服务终结点接受了 Get Metadata 请求后,生成相应的回复消息并将元数据置于消息的主体部分。Get Metadata 回复消息具有如下的结构。

```
[action]
http://schemas.xmlsoap.org/ws/2004/09/mex/GetMetadata/Response
[Body]
<mex:Metadata ...>
  ...
</mex:Metadata>
```

2.1.2 MetadataSection 与 MetadataSet

通过对 WS-MEX 的介绍,我们知道不论是采用 WS-Transfer Get 操作还是 Get Metadata 操作,获取到的元数据均被封装到回复消息主体部分的<Metadata>节点中。<Metadata>是一组<MetadataSection>元素的集合。在托管的世界里,<MetadataSection>元素和<MetadataSection>元素集合对应着相应的类型,即接下来要着重介绍的 System.ServiceModel.Description.MetadataSection 和 System.ServiceModel.Description.MetadataSet。

1. MetadataSection

MetadataSection 用于定义基于某种方言 (Dialect) 的元数据。WS-MEX 中包含元数据 SOAP 消息主体的 <MetadataSection> 节点的内容通过 **MetadataSection** 对象来表示。现在不妨看看 **MetadataSection** 的定义。

```
[XmlRoot (ElementName="MetadataSection",
    Namespace="http://schemas.xmlsoap.org/ws/2004/09/mex")]
public class MetadataSection
{
    //其他成员
    public MetadataSection();
    public MetadataSection(string dialect, string identifier, object metadata);

    [XmlAnyAttribute]
    public Collection<XmlAttribute> Attributes { get; }
    [XmlAttribute]
    public string Dialect { get; set; }
    [XmlAttribute]
    public string Identifier { get; set; }
    [XmlElement("Location", typeof(MetadataLocation),
        Namespace="http://schemas.xmlsoap.org/ws/2004/09/mex")]
    [XmlElement("MetadataReference", typeof(MetadataReference),
        Namespace="http://schemas.xmlsoap.org/ws/2004/09/mex")]
    [XmlElement("Metadata", typeof(MetadataSet),
        Namespace="http://schemas.xmlsoap.org/ws/2004/09/mex")]
    [XmlElement("schema", typeof(XmlSchema),
        Namespace = "http://www.w3.org/2001/XMLSchema")]
    [XmlElement("definitions", typeof(ServiceDescription),
        Namespace = "http://schemas.xmlsoap.org/wsdl/")]
    [XmlAnyElement]
    public object Metadata { get; set; }

    //4 种预定义元数据方言 (Dialect)
    //MEX: http://schemas.xmlsoap.org/ws/2004/09/mex
    public static string MetadataExchangeDialect { get; }
    //WS-Policy: http://schemas.xmlsoap.org/ws/2004/09/policy
    public static string PolicyDialect { get; }
    //WSDL: http://schemas.xmlsoap.org/wsdl/
    public static string ServiceDescriptionDialect { get; }
    //XML Schema: http://www.w3.org/2001/XMLSchema
    public static string XmlSchemaDialect { get; }
}
```

单看 **MetadataSection** 的定义, 你可能觉得没有太多值得关注的地方。如果结合 WS-MEX 规范, 就有很多值得玩味的地方了:

- 类型上应用了一个 **XmlRootAttribute** 特性, 并定义名称和命名空间分别为 **MetadataSection** 和 **http://schemas.xmlsoap.org/ws/2004/09/mex**。这和 WS-MEX 1.1 完全吻合。
- 属性 **Dialect** 表述元数据方言。在 WS-MEX 中定义了 5 种预定义元数据方言 (MEX、XML Schema、WSDL、WS-Policy 和 WS-Policy Attachment), 除了 WS-Policy Attachment, **MetadataSection** 为前面 4 种定义了相应的静态只读属性。

属性 Identifier 表示元数据的标识符, 这是一个以 URI 形式表示的字符串。Identifier 和 Dialect 最终被序列化后生成<MetadataSection>元素相应的属性 (Attribute)。MetadataSection 还定义了类型为 Collection<XmlAttribute>的 Attributes 属性, 可以自定义任意的 XML 属性, 最终将会作为<MetadataSection>元素的属性。

元数据的内容通常包含在属性 Metadata 中。当整个 MetadataSection 被序列化后, 该属性的值将会被序列化成 XML 元素, 其元素的名称和命名空间根据具体的类型决定。从应用在该属性上的一系列 XmlElementAttribute 特性可以看出 MetadataSection 为以下几种特殊的类型定义了相应的名称和命名空间。

(1) MetadataLocation

WS-MEX 1.1 规定了可以采用元数据文档地址的 URI 来替代相应元数据的内容, 而 MetadataLocation 就表示这个以 URI 形式体现元数据文档的地址。MetadataLocation 定义在 System.ServiceModel.Description 命名空间下, 具有如下的定义。

```
[XmlRoot(ElementName="Location",
          Namespace="http://schemas.xmlsoap.org/ws/2004/09/mex")]
public class MetadataLocation
{
    public MetadataLocation();
    public MetadataLocation(string location);

    [XmlText]
    public string Location { get; set; }
}
```

(2) MetadataReference

如果元数据成为一种可被寻址的资源, 就可以通过终结点引用 (Endpoint Reference) 的方式来定位该资源。WS-MEX 1.1 规定了可以采用元数据终结点引用来替代相应元数据的内容。元数据终结点引用可以通过 MetadataReference 来表示, 该类型定义于 System.ServiceModel.Description 命名空间下, 具有如下的定义。

```
[XmlRoot(ElementName = "MetadataReference",
          Namespace = "http://schemas.xmlsoap.org/ws/2004/09/mex")]
public class MetadataReference : IXmlSerializable
{
    public MetadataReference();
    public MetadataReference(EndpointAddress address, AddressingVersion
        addressVersion);
    public EndpointAddress Address { get; set; }
    public AddressingVersion AddressVersion { get; set; }
}
```

(3) MetadataSet

MetadataSet 就是我们即将介绍的用于表示 MetadataSection 的集合, 将 MetadataSet 作为 MetadataSection 的元数据, 意味着元数据可以以一种嵌套的形式来表示。

(4) XmlSchema

如果元数据的类型为 XmlSchema, 即表示以 XML Schema 方言 (Dialect) 表示的元数据。

(5) ServiceDescription

关于这里的 ServiceDescription, 指的是 System.Web.Services.Description.ServiceDescription, 而不是 System.ServiceModel.Description.ServiceDescription。后者是我们熟悉的对 WCF 服务的描述, 前者实际上是对一个 WSDL 文档的描述。如果元数据的类型为 ServiceDescription, 即表示以 WSDL 方言 (Dialect) 表示的元数据。

MetadataSection 还定义了如下三个静态方法帮助快速创建基于 WS-Policy 策略、XML Schema 和 WSDL 元数据方言的 MetadataSection 对象。

```
[XmlRoot(ElementName="MetadataSection",
          Namespace="http://schemas.xmlsoap.org/ws/2004/09/mex")]
public class MetadataSection
{
    //其他成员
    public static MetadataSection CreateFromPolicy(XmlElement policy, string
    identifier);
    public static MetadataSection CreateFromSchema(XmlSchema schema);
    public static MetadataSection CreateFromServiceDescription(ServiceDescription
    serviceDescription);
}
```

2. MetadataSet

MetadataSet 是对 WS-MEX 中置于 SOAP 消息主体部分的整个元数据内容的描述。既然 <Metadata> 节点是一组 <MetadataSection> 元素的集合, MetadataSet 相应地也就是一组 MetadataSection 对象的集合, 这可以从 MetadataSet 的定义看出来。

```
[XmlRoot("Metadata", Namespace="http://schemas.xmlsoap.org/ws/2004/09/mex")]
public class MetadataSet : IXmlSerializable
{
    //其他成员
    [XmlElement("MetadataSection",
               Namespace="http://schemas.xmlsoap.org/ws/2004/09/mex")]
    public Collection<MetadataSection> MetadataSections { get; }
}
```

2.1.3 WCF 元数据架构模型

WCF 通过终结点的形式将某个服务暴露出来, 而元数据的目的在于帮助服务的消费者有效地与该终结点进行交互, 以实现对该服务的正常调用。元数据帮助像 SvcUtil.exe 这样的代码生成工具有效地生成客户端代码和配置。WCF 在内部构建了一个完整的元数据架构体系, 很好地实现了元数据的导出、发布、获取和导入, 这个框架体系对元数据的处理大体如图 2-4 所示。

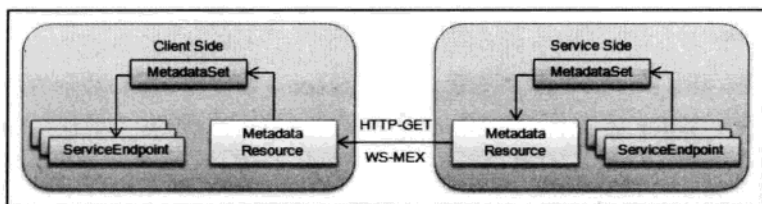


图 2-4 WCF 元数据架构体系对元数据的处理

从图 2-4 可以看出，整个元数据框架体系大体分成服务端体系和客户端体系。服务端负责元数据的导出和发布，而客户端实现元数据的获取与导入。元数据的导出、发布、获取和导入这 4 个基本操作在整个框架体系中分别实现以下的功能。

- 元数据导出 (Exporting)：将 WCF 服务相关的终结点列表转换成 `MetadataSet` 对象。元数据的导出通过 `System.ServiceModel.Description.MetadataExporter` 实现。
- 元数据发布 (Publishing)：将导出的 `MetadataSet` 对象转换成可被寻址的元数据资源通过相关的协议发布出来，`WS-MEX` 和 `HTTP-GET` 是两种常见的协议。元数据的发布通过 `System.ServiceModel.Description.ServiceMetadataBehavior` 服务行为实现。
- 元数据获取 (Retrieving)：通过相应的协议 (`WS-MEX` 或者 `HTTP-GET`) 获取发布出来的元数据资源，并转换成 `MetadataSet` 对象。元数据的获取通过 `System.ServiceModel.Description.MetadataExchangeClient` 实现。
- 元数据导入 (Importing)：将获取元数据资源生成的 `MetadataSet` 对象最终转换终结点对象。元数据导入通过 `System.ServiceModel.Description.MetadataImporter` 实现。

接下来将针对上述的 4 个元数据基本操作对 WCF 的元数据框架的实现原理进行深入的剖析。先来看看元数据是如何被导出的。

2.2 元数据的导出

元数据的导出就是实现从 `ServiceEndpoint` 对象向 `MetadataSet` 对象转换的过程。在 WCF 元数据框架体系中，元数据的导出工作由 `MetadataExporter` 实现。`MetadataExporter` 是一个抽象类型，定义了导出元数据的基本功能。WCF 定义一个具体的 `Wsdlexporter`，将基于某个终结点的元数据导出生成基于 WSDL 的 `MetadataSet`。下面先来认识 `MetadataExporter` 和 `Wsdlexporter`。

2.2.1 MetadataExporter 和 Wsdlexporter

下面的代码片段给出了抽象类型 `MetadataExporter` 的定义。它定义了三个与元数据导出相关的方法，其中 `ExportContract` 仅仅导出基于某个服务契约相关的元数据，而 `Export`

Endpoint 则导出某个终结点相关的所有元数据。这两个方法并不直接返回用于承载元数据信息的 MetadataSet 对象，而是将导出的元数据暂存于元数据转换的上下文中，最终通过 GetGeneratedMetadata 方法从该上下文中将导出的元数据提取出来。

```
public abstract class MetadataExporter
{
    public abstract void ExportContract(ContractDescription contract);
    public abstract void ExportEndpoint(ServiceEndpoint endpoint);
    public abstract MetadataSet GetGeneratedMetadata();

    public Collection<MetadataConversionError> Errors { get; }
    public PolicyVersion PolicyVersion { get; set; }
    public Dictionary<object, object> State { get; }
}
```

MetadataExporter 还定义了三个属性 Errors、PolicyVersion 和 State。Errors 是一个 MetadataConversionError 对象的集合，包含了在进行元数据导出的过程中出现的错误或者警告消息。字典类型的 State 可以作为一个存储元数据导出过程中动态使用到的对象的容器。而类型为 System.ServiceModel.Description.PolicyVersion 的 PolicyVersion 属性代表元数据基于的 WS-Policy 规范的版本。

PolicyVersion 的定义如下。由于定义的构造函数是私有的，因此不能直接利用 new 操作符创建该对象，只能通过定义在 PolicyVersion 中的两个静态只读属性 Policy12 和 Policy15 得到代表 WS-Policy 1.2 和 WS-Policy 1.5 的 PolicyVersion 对象。静态属性 Default 代表默认的 WS-Policy 版本，目前为 WS-Policy 1.2。属性 Namespace 表示相应 WS-Policy 版本的命名空间。

```
public sealed class PolicyVersion
{
    //其他成员
    private PolicyVersion(string policyNamespace);
    public static PolicyVersion Default { get; }
    public string Namespace { get; }
    public static PolicyVersion Policy12 { get; }
    public static PolicyVersion Policy15 { get; }
}
```

WCF 定义了一个具体的 MetadataExporter 类型用于将终结点导出为基于 WSDL 的 MetadataSet，即 WsdlExporter。

1. WsdlExporter

元数据具有三种主要的表现形式：XML Schema、WS-Policy 策略和 WSDL，而且 WSDL 可以直接采用 XML Schema 表示 Web 服务使用到的数据类型和消息结构，采用基于 WS-Policy 的策略断言定义其绑定行为，一个 WSDL 文档基本上可以用于表示 Web 服务的所有信息。正是因为 WSDL 是目前描述 Web 服务的最好的语言，所以建立 WCF 终结点与 WSDL 元素之间的匹配关系，以及基于该匹配关系的元数据导入和导出的实现，是 WCF 元

数据框架体系的一个最为重要的目标。

在 2.1 节对 WSDL 的介绍中, 已经谈过了 WCF 下终结点三要素 (地址、绑定和契约) 与组成一份完整 WSDL 文档 (基于 WSDL 1.1) 的 5 个元素之间的匹配关系, 现在进行简单的总结。组成 WSDL 的 5 个元素 (Service、Binding、PortType、Message 和 Type) 与终结点三要素之间的匹配关系大体上可以通过图 2-5 来体现, 其中 WSDL 元素之间的箭头代表引用关系, WSDL 和 ServicePoint 之间的箭头表示匹配关系。

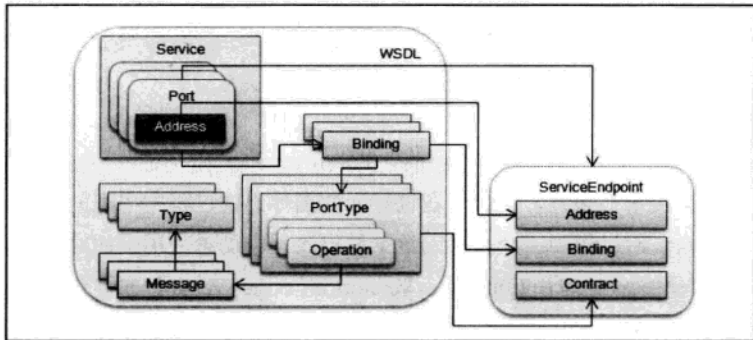


图 2-5 WSDL 各元素和终结点三要素之间的匹配关系

从图 2-5 不难看出, WSDL 中 Service 元素的一个 Port 实际上就代表着整个 ServiceEndpoint 对象, 而 Port 下的 Address 元素即终结点的地址。WSDL 中的 Binding 元素和终结点的 Binding 表示相同的内容。而终结点的契约则和一个 PortType 元素相匹配。

既然 WSDL 和 ServiceEndpoint 之间存在如此清晰的匹配关系, 那么直接将一个 ServiceEndpoint 对象导出成一个基于 WSDL 的 MetadataSet 就不会是一件很难的事情。Wsdlexporter 就是为实现这样的目标而设计的。

从下面给出的关于 MetadataExporter 的定义代码中, 可以看到 Wsdlexporter 直接继承 MetadataExporter。除了重写定义在 MetadataExporter 三个抽象方法之外, 还定义了一个 ExportEndpoints 方法帮助我们将一个包含多个终结点的服务作为一个整体导出, 因为一个 WSDL 本身就是对一个完整的 Web 服务的描述。

```
public class Wsdlexporter : MetadataExporter
{
    //其他成员
    public override void ExportContract(ContractDescription contract);
    public override void ExportEndpoint(ServiceEndpoint endpoint);
    public void ExportEndpoints(IEnumerable<ServiceEndpoint> endpoints,
        XmlQualifiedName wsdlServiceQName);
    public override MetadataSet GetGeneratedMetadata();

    public ServiceDescriptionCollection GeneratedWsdldocuments { get; }
    public XmlSchemaSet GeneratedXmlSchemas { get; }
}
```


Wsdlexporter 还定义了 GeneratedWsdldocuments 和 GeneratedXmlSchemas 两个只读属性。前者以 System.Web.Services.Description.ServiceDescription 集合的形式返回导出生成的 WSDL 文档, 后者则返回导出生成作为描述数据类型和消息结构的 XML Schema。

2. 实例演示: 如何通过 Wsdlexporter 导出元数据 (S201)

为了让读者更加深刻地认识 Wsdlexporter, 下面通过一个控制台应用实例演示如何利用它来导出服务的元数据。我们先演示如何利用 Wsdlexporter 导出一个单一的终结点。为此定义了一个处理订单的服务契约, 契约接口 IOrderService 和使用到的数据类型 (Order 和 OrderDetail) 定义如下。

```
using System.Collections.ObjectModel;
using System.Runtime.Serialization;
using System.ServiceModel;
namespace Artech.MetadataExporting
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface IOrderService
    {
        [OperationContract]
        void ProcessOrder(Order order);
    }
    [DataContract(Namespace = "http://www.artech.com/types/")]
    public class Order
    {
        [DataMember]
        public string OrderId { get; set; }
        [DataMember]
        public string CustomerId { get; set; }
        [DataMember]
        public Collection<OrderDetail> Details { get; set; }
    }
    [DataContract(Namespace = "http://www.artech.com/types/")]
    public class OrderDetail
    {
        [DataMember]
        public string OrderId { get; set; }
        [DataMember]
        public string ProductId { get; set; }
        [DataMember]
        public int Quantity { get; set; }
    }
}
```

接下来通过下面的代码创建两个 ServiceEndpoint 对象和一个表示服务有效名称 (QName) 的 XmlQualifiedName 对象, 传入 Wsdlexporter 的 ExportEndpoints 方法。然后通过调用 GetGeneratedMetadata 方法获取包含所有导出元数据的 MetadataSet 对象, 并将其写入一个 XML 文件中, 通过调用 Process 的静态 Start 方法打开该 XML 文件。

```
using System.Diagnostics;
using System.ServiceModel;
using System.ServiceModel.Description;
```

```

using System.Text;
using System.Xml;
namespace Artech.MetadataExporting
{
    class Program
    {
        static void Main(string[] args)
        {
            ContractDescription contract =
                ContractDescription.GetContract(typeof(IOrderService));
            EndpointAddress address1 = new
                EndpointAddress("http://127.0.0.1/orderservice");
            EndpointAddress address2 = new
                EndpointAddress("net.tcp://127.0.0.1/orderservice");
            ServiceEndpoint endpoint1 = new ServiceEndpoint(contract, new
                WS2007HttpBinding(), address1);
            ServiceEndpoint endpoint2 = new ServiceEndpoint(contract, new
                NetTcpBinding(), address2);
            XmlQualifiedName serviceName = new XmlQualifiedName("OrderService",
                "http://www.artech.com/services/");
            Wsdlexporter exporter = new Wsdlexporter();
            exporter.ExportEndpoints(new ServiceEndpoint[] { endpoint1, endpoint2 },
                serviceName);
            MetadataSet metadata = exporter.GetGeneratedMetadata();
            using (XmlWriter writer = new XmlTextWriter("metadata.xml",
                Encoding.UTF8))
            {
                metadata.WriteTo(writer);
            }
            Process.Start("metadata.xml");
        }
    }
}

```

由于本机采用 IE 作为开启 XML 文件默认的应用程序, 因此当上面的代码成功执行后, 包含有元数据的 XML 文件会通过 IE 打开。图 2-6 是运行后的截图, 从图中可以看出导出的元数据由 6 个 MetadataSection 构成。

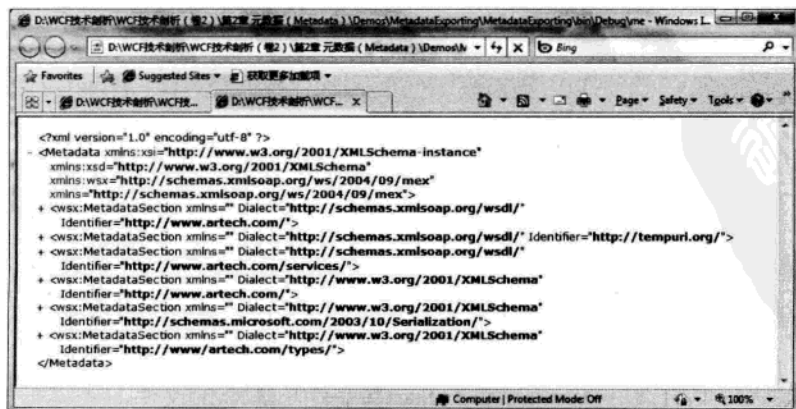


图 2-6 通过 IE 查看导出的元数据

所有 MetadataSection 的元数据方言 (Dialect) 集中在 WSDL 和 XML Schema 两种方法。其中基于 XML Schema 方言的 MetadataSection 描述了:

- 通过数据契约定义的 Order 和 OrderDetail 类型;
- ProcessOrder 操作的输入消息和输出消息的 XSD;
- 所有 CLR 基元类型 (Primary Type, 比如 int、double 和 DateTime 等) 的 XSD。

而所有基于 WSDL 方言的 MetadataSection 共同构建了一份反映服务的 WSDL 文档。该 WSDL 除了包含 WSDL 基本的 5 个元素之外, 还包含通过 WS2007HttpBinding 和 NetTcpBinding 导出的一些 WS-Policy 策略断言。由于篇幅所限, 不可能将整个 XML 展示出来, 对此感兴趣的读者可以下载本实例的源代码。

2.2.2 WSDL 导出扩展和策略导出扩展

通过上面对 WSDL 元素和终结点三要素的之间的匹配关系的介绍, 我们知道了 WSDL 的 Binding 元素来源于终结点的绑定对象。那么这些基于 Binding 的元数据及相应的策略断言是如何被写入 WSDL 的呢? 要回答这个问题, 就需要涉及两个扩展, 即 WSDL 导出扩展 (WSDL Export Extension) 和策略导出扩展 (Policy Export Extension)。

1. WSDL 导出扩展

终结点的绑定本质上就是相关的绑定元素 (BindingElement) 的有序组合, 所以基于绑定的 WSDL 导出扩展通过绑定元素的形式实现。对于需要向最终导出的 WSDL 添加与 Binding 相关的元数据的绑定元素, 必须实现 System.ServiceModel.Description.IwsdlExportExtension 接口。

WSDL 导出扩展并不限于被相应的绑定元素用于添加 Binding 相关的元数据, 也可以采用终结点行为、契约行为和操作行为作为 WSDL。下面的代码片段给出了 IwsdlExportExtension 接口的定义。IWsdlexportExtension 定义了 ExportContract 和 ExportEndpoint 两个方法, 分别对应于 Wsdlexporter 的同名方法。

```
public interface IWsdlexportExtension
{
    void ExportContract(Wsdlexporter exporter, WsdlexportContractConversionContext context);
    void ExportEndpoint(Wsdlexporter exporter, WsdlexportEndpointConversionContext context);
}
```

从 Wsdlexporter 的定义可以知道, 无论是调用 ExportContract 还是 ExportEndpoint 方法, 并不会直接将导出的元数据返回, 最终导出的元数据是通过于另外一个额外的方法 GetGeneratedMetadata 获得的。实际上当调用 Wsdlexporter 的 ExportContract 或者

ExportEndpoint 方法的时候, 会将导出的元数据暂存在一个基于 WsdlExporter 对象的上下文 (Context) 之中。对于 ExportContract 方法, 这个上下文对象是分别作为 IWsdlExportExtension 接口两个方法参数类型的 System.ServiceModel.Description.WsdlContractConversionContext, 而对于 ExportEndpoint 方法则是 System.ServiceModel.Description.WsdlEndpointConversionContext。

WCF 在针对某个服务的元数据导出过程中, 会先创建一个 WsdlExporter 和两个上下文对象 (WsdlContractConversionContext 和 WsdlEndpointConversionContext)。然后调用 ExportContract 和 ExportEndpoint 方法。

在执行这两个方法的最后阶段, 会遍历所有实现了 IWsdlExportExtension 接口的 WSDL 导出扩展元素 (对于 ExportContract 方法, 即所有实现了 IWsdlExportExtension 接口的三种行为对象。对于 ExportEndpoint 方法, 则同时包括实现了 IWsdlExportExtension 接口的行为对象和绑定元素), 并将 WsdlExporter 对象本身和相应的上下文对象 (WsdlContractConversionContext 或者 WsdlEndpointConversionContext) 作为参数执行 WSDL 导出扩展的 ExportContract 方法或者 ExportEndpoint 方法, 最终实现了将定制的元数据写入最终的 WSDL 的目的。

WSDL 导出扩展并不包含对 WS-Policy 策略断言的导出, 对其的实现定义在策略导出扩展中。

2. 策略导出扩展

WS 规范最终都是通过绑定实现的。WCF 通过定义相应的绑定元素对某个 WS 规范提供支持。终结点的绑定包含了很多基于相应 WS 规范的 WS-Policy 策略断言, 需要作为元数据导出到最终生成的 WSDL 中。比如对于 WSHttpBinding, 如果采用基于消息的安全模式, 需要导出基于 WS-Security 相关的策略断言。如果开启了可靠会话 (Reliable Sessions), 需要导出基于 WS-RM (WS-Reliable Messaging) 相关的策略断言。在 WCF 元数据结构体系中, 通过策略导出扩展实现对 WS-Policy 策略断言的导出。

所有需要实现 WS-Policy 策略断言导出的绑定元素必须实现具有如下定义的 System.ServiceModel.Description.IPolicyExportExtension 接口。该接口仅仅定义了唯一的 ExportPolicy 方法, 用于导出相关的策略断言。

```
public interface IPolicyExportExtension
{
    void ExportPolicy(MetadataExporter exporter, PolicyConversionContext context);
}
```

在 WsdlExporter 执行 ExportEndpoint 方法的最后阶段, 会创建 System.ServiceModel.Description.PolicyConversionContext 上下文对象。遍历所有实现了 IPolicyExportExtension 接口的所有绑定元素, 并将 WsdlExporter 对象本身和该 PolicyConversionContext 对象作为参数调用这些绑定元素的 ExportPolicy 方法。

这些作为策略导出扩展的绑定元素将相应的基于 WS-Policy 策略的元数据导出到 PolicyConversionContext 对象中。待所有绑定元素执行完毕，再将暂存于 PolicyConversionContext 的策略元数据附加到上面提到的 WsdlEndpointConversionContext 对象上，那么最后导出的元数据就包含了相应的 WS-Policy 策略断言。

2.3 元数据的发布

对于 WCF 服务端元数据架构体系来说，通过 MetadataExporter 将服务的终结点导出成 MetadataSet，仅仅是完成了一半的工作。被成功导出的以 MetadataSet 对象表示的元数据最终需要作为可被访问的网络资源发布出来。元数据的发布最终是通过 System.ServiceModel.Description.ServiceMetadataBehavior 这样一个服务行为实现的，下面先来认识一下 ServiceMetadataBehavior。

2.3.1 元数据发布的实现者：ServiceMetadataBehavior

ServiceMetadataBehavior 是一个实现了 IServiceBehavior 接口的服务行为，它实现了基于如下两种协议的元数据发布模式。

- HTTP-GET：服务的消费者向元数据目标地址发送 HTTP-Get 请求，并以查询字符串（QueryString）的形式表示相应的查询参数。获取到的元数据最终以 HTTP 回复的形式返回。
- WS-MEX：元数据提供者按照 WS-MEX 规范创建终结点发布元数据，元数据消费者创建同样基于 WS-MEX 的终结点与之交互，并最终通过 SOAP 的形式获取元数据。

下面的代码片段给出了 ServiceMetadataBehavior 这个服务行为的基本定义，它实现 IServiceBehavior 接口，针对服务元数据的发布则实现在 ApplyDispatchBehavior 方法中。

```
public class ServiceMetadataBehavior : IServiceBehavior
{
    //其他成员
    public Uri ExternalMetadataLocation { get; set; }

    //HTTP
    public bool HttpGetEnabled { get; set; }
    public Uri HttpGetUrl { get; set; }
    public Binding HttpGetBinding { get; set; }

    //HTTPS
    public bool HttpsGetEnabled { get; set; }
    public Binding HttpsGetBinding { get; set; }
    public Uri HttpsGetUrl { get; set; }

    public MetadataExporter MetadataExporter { get; set; }
}
```

`ServiceMetadataBehavior` 定义了一系列的属性用于控制具体的元数据发布行为, 其中绝大部分是基于 HTTP-GET 发布模式的。`ExternalMetadataLocation` 表示返回给客户端的一个外部元数据地址, 可以是绝对地址, 也可以是基于 `HttpGetUrl` 或者 `HttpsGetUrl` 表述的相对地址。

基于 HTTP-GET 的元数据发布同时支持 HTTP 和 HTTPS 两种形式, `Http(s)GetEnabled` 属性是控制是否允许基于 HTTP(s) 进行元数据发布的开关, `Http(s)GetUrl` 和 `Http(s)GetBinding` 则指定了采用的地址和绑定。`MetadataExporter` 属性表示的 `MetadataExporter` 对象用于进行元数据的导出, 默认为 `WsdlExporter`。

可以通过配置的方式来设置除 `MetadataExporter` 之外的所有 `ServiceMetadataBehavior` 属性。`WCF` 还提供了一些额外的配型项供你更好地控制元数据的发布行为。对于 `WCF` 的开发者来说, 当没有一份完备的文档指导进行基于服务行为或者终结点行为的配置时, 可以查看该行为对应的类型为 `BehaviorExtensionElement` 的配置元素定义获取与该行为相关的所有配置信息。`ServiceMetadataBehavior` 相关的配置项全部定义在 `System.ServiceModel.Configuration.ServiceMetadataPublishingElement` 中, 下面给出了 `ServiceMetadataPublishingElement` 的定义。

```
public sealed class ServiceMetadataPublishingElement :
BehaviorExtensionElement
{
    //其他成员
    public override Type BehaviorType { get; }
    [ConfigurationProperty("externalMetadataLocation")]
    public Uri ExternalMetadataLocation { get; set; }

    //HTTP
    [ConfigurationProperty("httpGetEnabled", DefaultValue = false)]
    public bool HttpGetEnabled { get; set; }
    [ConfigurationProperty("httpGetUrl")]
    public Uri HttpGetUrl { get; set; }
    [ConfigurationProperty("httpGetBinding", DefaultValue = "")]
    public string HttpGetBinding { get; set; }
    [ConfigurationProperty("httpGetBindingConfiguration", DefaultValue = "")]
    public string HttpGetBindingConfiguration { get; set; }

    //HTTPS
    [ConfigurationProperty("httpsGetEnabled", DefaultValue = false)]
    public bool HttpsGetEnabled { get; set; }
    [ConfigurationProperty("httpsGetUrl")]
    public Uri HttpsGetUrl { get; set; }
    [ConfigurationProperty("httpsGetBinding", DefaultValue = "")]
    public string HttpsGetBinding { get; set; }
    [ConfigurationProperty("httpsGetBindingConfiguration", DefaultValue = "")]
    public string HttpsGetBindingConfiguration { get; set; }

    [ConfigurationProperty("policyVersion", DefaultValue = "Default")]
    public PolicyVersion PolicyVersion { get; set; }
}
```

通过对 `ServiceMetadataPublishingElement` 的定义可以看出, 不但可以通过配置指定服务行为为 `ServiceMetadataBehavior` 除 `MetadataExporter` 之外的所有属性, 还可以通过 `policyVersion` 配置项指定具体采用的 WS-Policy 的版本。下面是一个 `ServiceMetadataBehavior` 配置的例子, 在

这段配置中我们同时开启基于 HTTP 和 HTTPS 的元数据发布方式,并采用 WS-Policy 1.5 版本。

```
<configuration>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration="metadataPublishBehavior"...>
        ...
      </service>
    </services>

    <behaviors>
      <serviceBehaviors>
        <behavior name="metadataPublishBehavior">
          <serviceMetadata
            externalMetadataLocation="http://127.0.1/mex/calculatorService"
            httpGetEnabled="true"
            httpGetUrl="http://127.0.0.1/calculatorService/mex"
            httpsGetEnabled="true"
            httpsGetUrl="https://127.0.0.1/calculatorService/mex"
            policyVersion="Policy15" />
          </behavior>
        </serviceBehaviors>
      </behaviors>
    </system.serviceModel>
  </configuration>
```

如果希望通过 WS-MEX 的方式进行元数据的发布,需要为服务添加一个基于 WS-MEX 的终结点。基于 WS-MEX 的终结点和一般意义上的终结点一样由地址、绑定和契约三部分组成。地址表示发布元数据的目标地址。因为需要按照 WS-MEX 规范进行消息的交换,所以对绑定和契约具有特殊的要求。在具体对 MEX 终结点进行详细介绍之前,不妨先来看看如何通过配置的方式为服务添加 MEX 终结点。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.Services.CalculatorService">
        <endpoint address="http://127.0.0.1:3721/calculatorService"
          binding="basicHttpBinding" contract="Artech.
            Contracts.ICalculator" />
        <endpoint address="http://127.0.0.1:3721/calculatorService/mex"
          binding="mexHttpBinding" contract="IMetadataExchange"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

2.3.2 MEX 终结点有何不同

我们通过为服务添加基于 WS-MEX 的终结点(以下简称 MEX 终结点)实现基于 WS-MEX 的元数据发布方式。总的来说, MEX 终结点和一般的终结点并没有本质的不同。但是为了支持 WS-MEX 规定的消息交换模式和请求/回复消息的结构,对契约和绑定具有一些特殊的要求。下面先来看看 MEX 终结点的契约。

1. MEX 终结点的契约 (IMetadataExchange)

从上面给出的基于 MEX 终结点的配置中, 可以看到该终结点的契约被配置成 `IMetadataExchange`。实际上 `IMetadataExchange` 是 WCF 内部定义的一个特殊服务契约接口, 定义在 `System.ServiceModel.Description` 命名空间下。从下面给出的关于 `IMetadataExchange` 的定义可以看出, 该契约接口仅仅包含分别以同步和异步方式定义的 `Get` 方法。

```
[ServiceContract(ConfigurationName = "IMetadataExchange",
    Name = "IMetadataExchange",
    Namespace = "http://schemas.microsoft.com/2006/04/mex")]
public interface IMetadataExchange
{
    [OperationContract(
        Action = "http://schemas.xmlsoap.org/ws/2004/09/transfer/Get",
        ReplyAction = "http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse",
        AsyncPattern = true)]
    IAsyncResult BeginGet(Message request, AsyncCallback callback, object state);
    Message EndGet(IAsyncResult result);

    [OperationContract(
        Action = "http://schemas.xmlsoap.org/ws/2004/09/transfer/Get",
        ReplyAction = "http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse")]
    Message Get(Message request);
}
```

通过 `ServiceContractAttribute` 特性将 `ConfigurationName` 属性设定成“`ImetadataExchange`”, 这也是为何在进行 MEX 终结点的配置的时候, 契约可以直接配置成“`ImetadataExchange`”, 而不是采用接口类型的全名“`System.ServiceModel.Description.ImetadataExchange`”的原因。

再看看 `Get` 操作, 通过 `OperationContractAttribute` 特性将 `Action` 和 `ReplyAction` 设置成了 `http://schemas.xmlsoap.org/ws/2004/09/transfer/Get` 和 `http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse`, 如果读者读 2.1 节对 WS-Transfer 相关介绍还有印象, 应该知道它们正是 WS-Transfer `Get` 请求和回复消息对应的 `<Action>` 报头的值。

在介绍 WS-MEX 的时候, 我们提到过 WS-MEX 支持两种形式的元数据获取方式: WS-Transfer `Get` 操作请求和 `Get Metadata` 操作请求。从这里可以看出, WCF 采用的是基于 WS-Transfer `Get` 操作的元数据请求方式。

2. MEX 终结点的绑定

WCF 专门为 MEX 终结点定制了一系列的绑定, 以实现对不同的网络传输协议 (HTTP、HTTPS、TCP 和 Named Pipe) 的支持。不过一般不会直接通过 `new` 操作符来创建这些 MEX 绑定, 而是通过静态类型 `System.ServiceModel.Description.MetadataExchangeBindings` 相应的 `CreateMexXxxBinding` 方法来创建相应的绑定。从下面给出的 `MetadataExchangeBindings` 的定义可以看出, 它定义了 4 个 `CreateMexXxxBinding` 方法, 分别用于创建上述 4 种类型的 MEX 绑定。


```

public static class MetadataExchangeBindings
{
    //其他成员
    public static Binding CreateMexHttpBinding();
    public static Binding CreateMexHttpsBinding();
    public static Binding CreateMexTcpBinding();
    public static Binding CreateMexNamedPipeBinding();
}

```

如果采用编程的方式为服务添加 MEX 终结点,那么可以直接借助 `MetadataExchangeBindings` 创建相应的 MEX 绑定。如果采用配置的方式,仅仅将终结点的 binding 分别配置成 `mexHttpBinding`、`mexHttpsBinding`、`mexTcpBinding` 和 `mexNamedPipeBinding` 即可。在下面的这段配置中,分别定义了基于这 4 种绑定的 MEX 终结点。

```

<configuration>
  <system.serviceModel>
    <services>
      <service ...>
        ...
        <endpoint address="http://127.0.0.1/calculator/mex"
                  binding="mexHttpBinding"
                  contract="IMetadataExchange" />
        <endpoint address="https://127.0.0.1/calculator/mex"
                  binding="mexHttpsBinding"
                  contract="IMetadataExchange" />
        <endpoint address="net.tcp://127.0.0.1/calculator/mex"
                  binding="mexTcpBinding"
                  contract="IMetadataExchange" />
        <endpoint address="net.pipe://127.0.0.1/calculator/mex"
                  binding="mexNamedPipeBinding"
                  contract="IMetadataExchange" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

2.3.3 ServiceMetadataBehavior 是如何实现元数据发布的

通过上面的介绍我们知道了如何通过编程或者配置的方式将 `ServiceMetadataBehavior` 这样一个服务形式应用到相应的服务上面,从而实现基于 HTTP-GET 或者 WS-MEX 的元数据发布。那么在 WCF 内部具体的实现原理又是怎样的呢?

1. 从 WCF 分发体系谈起

如果读者想对 WCF 内部的元数据发布机制的实现原理有一个全面而深入的了解,必须对 WCF 服务端的分发体系有一个清晰的认识。在这里先对该分发体系做一个概括性的介绍,如果希望对此做进一步的了解可以参考本书的第 9 章“扩展 (Extension)”。WCF 整个分发体系在进行服务寄宿时被构建,该体系的基本结构基本上可以通过图 2-7 体现。

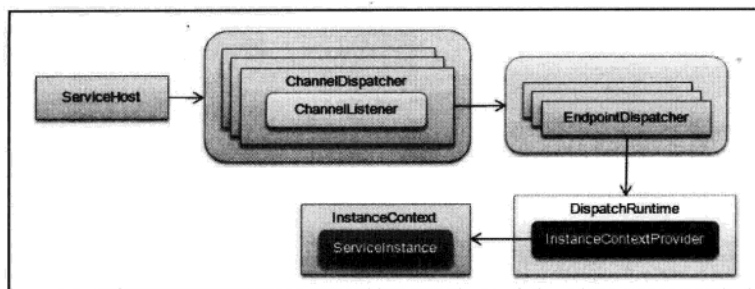


图 2-7 WCF 服务端分发体系

当我们创建 `ServiceHost` 对象成功寄宿某个服务后，WCF 会根据监听地址的不同为该 `ServiceHost` 对象创建一到多个信道分发器（`ChannelDispatcher`）对象。每个信道分发器都拥有各自的信道监听器（`ChannelListener`），它们绑定到相应的监听地址监听来自外界的请求。对于每一个信道分发器对象，具有一到多个终结点分发器（`EndpointDispatcher`）对象与之匹配，每一个终结点分发器对应着某个终结点。

而对于每一个终结点分发器，在初始化的时候会为之创建一个分发运行时（`DispatchRuntime`）。分发运行时拥有一系列用于处理请求、激活对象和执行方法等操作的运行时对象，在这里主要关注一个称为实例上下文提供者（`InstanceContextProvider`）的对象用于提供封装有相应服务实例的实例上下文（`InstanceContext`）。

2. 基于 WS-MEX 模式下的元数据发布是如何实现的

现在再把话题移到元数据发布上来，先来谈谈基于 `WS-MEX` 协议的元数据发布方式。在这种元数据发布模式下，服务端通过 `MEX` 终结点发布元数据，客户端创建相应的 `MEX` 终结点获取元数据，这和一般意义上的服务调用并没有本质的不同。完全可以将元数据的获取当成一个针对某个服务操作的调用，而该服务操作的返回值就是目标服务的元数据。

如果我们通过编程或者配置的方式为某个服务添加了一个 `MEX` 终结点，当服务被成功寄宿后，WCF 会为之创建一个信道分发器。该信道分发器拥有一个用于监听元数据请求的信道监听器，监听的地址就是数据发布的地址。基于该 `MEX` 终结点的终结点分发器对象连同其分发运行时也会被创建，并与该信道分发器关联在一起。

与普通终结点关联的分发运行时一样，`MEX` 终结点的分发运行时同样拥有相同的运行时对象集合。但是由于并没有一个真正用于提供元数据的服务被寄宿，分发运行时的实例上下文提供者（默认是 `PerSessionInstanceContextProvider`）是获取不到包含有真正服务实例的实例上下文的。

如果我们能够定制 `MEX` 终结点的分发运行时的实例上下文提供者，而它提供的实例上下文封装一个发布元数据的服务实例，那么元数据的发布就迎刃而解了。实际上，`ServiceMetadataBehavior` 服务行为内部就是这么做的，而这个用于提供元数据的服务类型就

是如下实现了 `IMetadataExchange` 契约接口的内部类 `WSMexImpl`。

```
internal class WSMexImpl : IMetadataExchange
{
    //其他成员
    public IAsyncResult BeginGet(Message request, AsyncCallback callback,
        object state);
    public Message EndGet(IAsyncResult result);
    private MetadataSet GatherMetadata(string dialect, string identifier);
    public Message Get(Message request);
}
```

当 `ServiceMetadataBehavior` 的 `ApplyDispatchBehavior` 方法被执行的时候, `WSMexImpl` 对象被创建出来并被封装在一个 `InstanceContext` 对象中。而该 `InstanceContext` 对象作为 MEX 终结点分发运行时的单例实例上下文 (`SingletonInstanceContext`)。然后创建一个单例实例上下文提供者 (`SingletonInstanceContextProvider`) 作为该分发运行时的实例上下文提供者。那么 MEX 终结点的分发运行时就能根据元数据请求消息激活 `WSMexImpl` 服务了。

上述的这些内容虽然不算复杂,但是要求读者对 WCF 的实例上下文机制有清晰的认识。对此不太熟悉的读者,可以参阅上册的第 9 章“实例化与会话 (Instancing and Session)”。为了加深读者对基于 WS-MEX 元数据发布机制的理解,接下来进行一个简单的实例演示。

3. 实例演示: 模拟 `ServiceMetadataBehavior` 实现基于 WS-MEX 元数据发布 (S202)

接下来会基于上面介绍的 `ServiceMetadataBehavior` 服务行为发布元数据的实现原理,创建一个自定义服务行为实现基于 WS-MEX 的元数据发布。首先来编写一些辅助性质的代码。

由于在本例中需要创建一些与分发运行时相关的运行时对象,而且很多对象属于内部类型,所以需要通过反射的机制来创建它们。此外为某些对象的一些私有或者内部属性赋值也需要利用反射,为此写了如下一个 `Utility` 类。

```
using System;
using System.Globalization;
using System.Reflection;
namespace Artech.ServiceMetadataBehaviorSimulator
{
    public static class Utility
    {
        public static T CreateInstance<T>(string typeQname, Type[] parameterTypes,
            object[] parameters) where T : class
        {
            Type type = Type.GetType(typeQname);
            BindingFlags bindingFlags = BindingFlags.NonPublic | BindingFlags.Public
                | BindingFlags.Instance | BindingFlags.Static;
            ConstructorInfo constructorInfo = type.GetConstructor(bindingFlags,
                Type.DefaultBinder, parameterTypes, null);
            return Activator.CreateInstance(type, bindingFlags, Type.DefaultBinder,
                parameters, CultureInfo.InvariantCulture) as T;
        }

        public static void SetPropertyValue(object target, string propertyName,
```

```

        object propertyValue)
    {
        BindingFlags bindingFlags = BindingFlags.NonPublic |
            BindingFlags.SetProperty | BindingFlags.Instance;
        PropertyInfo propertyInfo = target.GetType().GetProperty(propertyName,
            bindingFlags);
        propertyInfo.SetValue(target, propertyValue, null);
    }
}

```

接下来仿照 `IMetadataExchange` 接口定义了如下一个 `IMetadataProvisionService` 接口。为了简化,省略了异步模式定义的 `Get` 操作(`BeginGet/EndGet`)。`Get` 操作的 `Action` 和 `ReplyAction` 同样基于 `WS-Transfer` 规范定义。通过 `ServiceContractAttribute` 特性将契约的 `Name` 和 `ConfigurationName` 设定成 `IMetadataProvisionService`。

```

using System.ServiceModel;
using System.ServiceModel.Channels;
namespace Artech.ServiceMetadataBehaviorSimulator
{
    [ServiceContract(ConfigurationName = "IMetadataProvisionService",
        Name = "IMetadataProvisionService",
        Namespace = "http://schemas.microsoft.com/2006/04/mex")]
    public interface IMetadataProvisionService
    {
        [OperationContract(
            Action = "http://schemas.xmlsoap.org/ws/2004/09/transfer/Get",
            ReplyAction = "http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse")]
        Message Get(Message request);
    }
}

```

由于 `Get` 操作返回的是封装有元数据(以 `MetadataSet` 的形式)的消息对象,因此定义了如下一个消息契约(`Message Contract`) `MetadataMessage`。`MetadataMessage` 通过 `MessageBodyMemberAttribute` 特性直接将类型为 `MetadataSet` 的属性定义成消息主体成员,并按照 `WS-MEX` 规范设置该成员的名称和命名空间。

```

using System.ServiceModel;
using System.ServiceModel.Description;
namespace Artech.ServiceMetadataBehaviorSimulator
{
    [MessageContract(IsWrapped = false)]
    public class MetadataMessage
    {
        public MetadataMessage(MetadataSet metadata)
        {
            this.Metadata = metadata;
        }
        [MessageBodyMember(Name = "Metadata",
            Namespace = "http://schemas.xmlsoap.org/ws/2004/09/mex")]
        public MetadataSet Metadata { get; set; }
    }
}

```

接下来创建如下一个真正用于提供元数据的服务类 `MetadataProvisionService`。`Metadata`

ProvisionService 实现了上面定义的服务契约接口 IMetadataProvisionService，具有一个 MetadataSet 类型的属性成员 Metadata。在 Get 方法中，通过 Metadata 属性表述的 MetadataSet 创建 MetadataMessage 对象，并将其转换成 Message 对象返回。最终返回的消息具有 WS-Transfer 规定的<Action>报头（http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse）。

```
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
namespace Artech.ServiceMetadataBehaviorSimulator
{
    public class MetadataProvisionService : IMetadataProvisionService
    {
        public MetadataSet Metadata{ get; private set; }
        public MetadataProvisionService(MetadataSet metadata)
        {
            this.Metadata = metadata;
        }
        public Message Get(Message request)
        {
            MetadataMessage message = new MetadataMessage(this.Metadata);
            string action =
                "http://schemas.xmlsoap.org/ws/2004/09/transfer/GetResponse";
            TypedMessageConverter converter =
                TypedMessageConverter.Create(typeof(MetadataMessage), action);
            return converter.ToMessage(message, request.Version);
        }
    }
}
```

最后就可以创建用于实现元数据发布的服务行为了。在这里使用了与 ServiceMetadataBehavior 相同的名字并将其定义成特性（我们就可以直接通过特性的方式应用到服务类型上）。元数据的发布体现在 ApplyDispatchBehavior 方法中，该方法先后执行以下两组操作。

- 导出元数据：直接通过 WsdlExporter 将服务相关的所有终结点导出生成 MetadataSet。需要注意的是，在进行终结点收集的时候，需要过滤到 MEX 终结点。元数据导出的所有操作实现在 GetExportedMetadata 方法中。
- 定制 MEX 终结点的分发运行时：在所有的终结点分发器列表中，筛选出基于 MEX 终结点的终结点分发器，然后定制它们的分发运行时。即创建 SingletonInstanceContext Provider 作为实例上下文提供者。根据导出的 MetadataSet 创建 MetadataProvisionService 对象，并将其封装在 InstanceContext 中，最后将该 InstanceContext 直接设定为分发运行时的单例实例上下文。

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using System.Xml;
namespace Artech.ServiceMetadataBehaviorSimulator
{
```

```

[AttributeUsage(AttributeTargets.Class)]
public class ServiceMetadataBehaviorAttribute : Attribute, IServiceBehavior
{
    private const string MexContractName = "IMetadataProvisionService";
    private const string MexContractNamespace =
        "http://schemas.microsoft.com/2006/04/mex";
    private const string SingletonInstanceContextProviderType =
        "System.ServiceModel.Dispatcher.SingletonInstanceContextProvider, System.Se
        rvicemodel, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089";
    public void AddBindingParameters(ServiceDescription serviceDescription,
        ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
        BindingParameterCollection bindingParameters) {}

    public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
        ServiceHostBase serviceHostBase)
    {
        MetadataSet metadata = GetExportedMetadata(serviceDescription);
        CustomizeMexEndpoints(serviceDescription, serviceHostBase, metadata);
    }
    private static MetadataSet GetExportedMetadata(ServiceDescription
        serviceDescription)
    {
        Collection<ServiceEndpoint> endpoints = new
            Collection<ServiceEndpoint>();
        foreach (var endpoint in serviceDescription.Endpoints)
        {
            if (endpoint.Contract.ContractType ==
                typeof(IMetadataProvisionService))
            {
                continue;
            }
            ServiceEndpoint newEndpoint = new ServiceEndpoint(endpoint.Contract,
                endpoint.Binding, endpoint.Address);
            newEndpoint.Name = endpoint.Name;
            foreach (var behavior in endpoint.Behaviors)
            {
                newEndpoint.Behaviors.Add(behavior);
            }
            endpoints.Add(newEndpoint);
        }
        WsdlExporter exporter = new WsdlExporter();
        XmlQualifiedName wsdlServiceQName = new
            XmlQualifiedName(serviceDescription.Name,
                serviceDescription.Namespace);
        exporter.ExportEndpoints(endpoints, wsdlServiceQName);
        MetadataSet metadata = exporter.GetGeneratedMetadata();
        return metadata;
    }
    private static void CustomizeMexEndpoints(ServiceDescription serviceDescription,
        ServiceHostBase host, MetadataSet metadata)
    {
        foreach (ChannelDispatcher channelDispatcher in host.ChannelDispatchers)
        {
            foreach (EndpointDispatcher endpoint in channelDispatcher.
                Endpoints)
            {
                if (endpoint.ContractName == MexContractName &&

```



```

    public Message Get(Message request)
    {
        throw new NotImplementedException();
    }
}

```

那么在进行服务寄宿的时候,就可以采用如下的方式添加 MEX 终结点了。可以看到这与 WCF 本身支持的 MEX 终结点的配置基本上是一样的,唯一不同的是这里的契约是我们自定义 `IMetadataProvisionService`, 而不是 `IMetadataExchange`。

```

<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService">
        <endpoint
          address="http://127.0.0.1:3721/calculatorservice"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.
            ICalculator"/>
        <endpoint address="http://127.0.0.1:9999/calculatorservice/mex"
          binding="mexHttpBinding"
          contract="IMetadataProvisionService"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

在客户端就可以采用与一般服务调用完全一样的方式获取服务的元数据了,下面是客户端的配置。需要注意的是这里配置的终结点并不是调用 `Calculatorservice` 的终结点,而是为了获取元数据的 MEX 终结点。地址是服务端 MEX 终结点的地址,契约是 `IMetadataProvisionService`, 采用的绑定是标准的基于 HTTP 的 MEX 绑定。

```

<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="mex"
        address="http://127.0.0.1:9999/calculatorservice/mex"
        binding="mexHttpBinding"
        contract="IMetadataProvisionService"/>
    </client>
  </system.serviceModel>
</configuration>

```

下面是基于 `ChannelFactory<TChannel>` 创建服务代理的客户端代码,可以看到与一般的服务调用并无二致。获取的元数据最终被写入一个 XML 文件并被打开。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel.Channels;
using System.ServiceModel;
using Artech.ServiceMetadataBehaviorSimulator;
using System.ServiceModel.Description;

```



```

using System.Xml;
using System.Diagnostics;
namespace Artech.WcfServices.Client
{
    class Program
    {
        static void Main(string[] args)
        {
            using (ChannelFactory<IMetadataProvisionService> channelFactory = new
                ChannelFactory<IMetadataProvisionService>("mex"))
            {
                IMetadataProvisionService proxy = channelFactory.CreateChannel();
                string action = "http://schemas.xmlsoap.org/ws/2004/09/
                    transfer/Get";
                Message request = Message.CreateMessage(MessageVersion.Default,
                    action);
                Message reply = proxy.Get(request);
                MetadataSet metadata = reply.GetBody<MetadataSet>();
                using (XmlWriter writer = new XmlTextWriter("metadata.xml",
                    Encoding.UTF8))
                {
                    metadata.WriteTo(writer);
                }
                Process.Start("metadata.xml");
            }
        }
    }
}

```

上面的应用如果正常执行，包含所有元数据信息的 XML 文件将会通过 IE（假设使用 IE 作为开启 XML 文件的默认应用程序）开启，图 2-8 是运行后的截图。

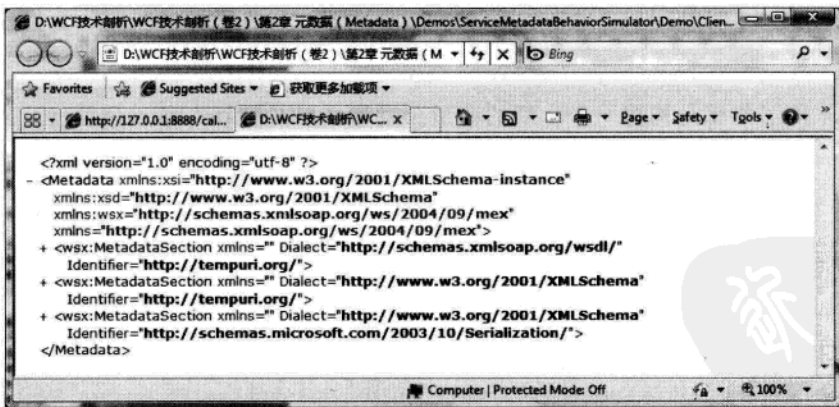


图 2-8 获取的元数据在 IE 中的显示

4. 基于 HTTP-GET 模式下的元数据发布是如何实现的 (S203)

基于 HTTP-GET 的元数据发布方式与基于 WS-MEX 原理类似，但是 ServiceMetadataBehavior 需要做更多的工作。原因很简单，由于在 WS-MEX 模式下，我们为寄宿的服务添加了相应的 MEX 终结点，因此当服务被成功寄宿后，WCF 已经为元数据的消

息交换建立了如图 2-7 所示的分发体系, 我们需要做的仅仅是对 MEX 终结点的分发运行时进行相应的定制而已。

但是如果采用 HTTP-GET 模式, 实际上需要从信道分发器开始, 重新构建整个分发体系。接下来, 在上面实例的基础上, 对 ServiceMetadataBehaviorAttribute 进行进一步的完善, 使之同时对两种模式的元数据发布提供支持。

首先需要定义一个新的服务契约接口 IHttpGetMetadata, Get 操作处理任何形式的消息请求。因为它的输入参数和返回类型均为 Message, 并且 Action 和 ReplyAction 为*。

```
using System.ServiceModel;
using System.ServiceModel.Channels;
namespace Artech.ServiceMetadataBehaviorSimulator
{
    [ServiceContract(Name = "IHttpGetMetadata",
        Namespace = "http://www.artech.com/")]
    public interface IHttpGetMetadata
    {
        [OperationContract(Action = "*", ReplyAction = "*")]
        Message Get(Message msg);
    }
}
```

然后我们让前面定义的 MetadataProvisionService 实现 IHttpGetMetadata 接口, 在这里无须再写任何多余的代码, 因为 MetadataProvisionService 已经具有了一个 Get 方法。

```
public class MetadataProvisionService : IMetadataProvisionService,
    IHttpGetMetadata
{
    //省略成员
}
```

接下来的工作就是构建一个全新的信道分发器, 以及关联终结点分发器, 最后对 EndpointDispatcher 的分发策略进行定制。为此单独写了如下一个 CreateHttpGetChannel Dispatcher 方法。

```
[AttributeUsage(AttributeTargets.Class)]
public class ServiceMetadataBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    private const string SingletonInstanceContextProviderType =
        "System.ServiceModel.Dispatcher.SingletonInstanceContextProvider, System.Se
        rvicemodel, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089";
    private const string SyncMethodInvokerType =
        "System.ServiceModel.Dispatcher.SyncMethodInvoker, System.ServiceModel,
        Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089";
    private const string MessageOperationFormatterType =
        "System.ServiceModel.Dispatcher.MessageOperationFormatter, System.ServiceModel,
        Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089";

    private static void CreateHttpGetChannelDispatcher(ServiceHostBase host, Uri
        listenUri, MetadataSet metadata)
    {

```

```

//创建 Binding
TextMessageEncodingBindingElement messageEncodingElement = new
    TextMessageEncodingBindingElement() { MessageVersion =
        MessageVersion.None };
HttpTransportBindingElement transportElement = new
    HttpTransportBindingElement();
Utility.SetPropertyValue(transportElement, "Method", "GET");
Binding binding = new CustomBinding(messageEncodingElement,
    transportElement);

//创建 ChannelListener
IChannelListener listener =
    binding.BuildChannelListener<IReplyChannel>(listenUri, string.Empty,
        ListenUriMode.Explicit, new BindingParameterCollection());
ChannelDispatcher dispatcher = new ChannelDispatcher(listener,
    "ServiceMetadataBehaviorHttpGetBinding", binding) { MessageVersion =
        binding.MessageVersion };

//创建 EndpointDispatcher
EndpointDispatcher endpoint = new EndpointDispatcher(new
    EndpointAddress(listenUri), "IHttpGetMetadata",
    "http://www.artech.com/");

//创建 DispatchOperation, 并设置 DispatchMessageFormatter 和 Operation
Invoker
DispatchOperation operation = new DispatchOperation(endpoint.Dispatch
    Runtime, "Get", "*", "");
operation.Formatter =
    Utility.CreateInstance<IDispatchMessageFormatter>(
        MessageOperationFormatterType, Type.EmptyTypes, new object[0]);
MethodInfo method = typeof(IHttpGetMetadata).GetMethod("Get");
operation.Invoker = Utility.CreateInstance<IOperationInvoker>(<
    SyncMethodInvokerType, new Type[] { typeof(MethodInfo) }, new object[]
        { method });
endpoint.DispatchRuntime.Operations.Add(operation);

//设置 SingletonInstanceContext 和 InstanceContextProvider
MetadataProvisionService serviceInstance = new
    MetadataProvisionService(metadata);
endpoint.DispatchRuntime.SingletonInstanceContext =
    new InstanceContext(host, serviceInstance);
endpoint.DispatchRuntime.InstanceContextProvider =
    Utility.CreateInstance<IInstanceContextProvider>(<
        SingletonInstanceContextProviderType,
        new Type[] { typeof(DispatchRuntime) },
        new object[] { endpoint.DispatchRuntime });
dispatcher.Endpoints.Add(endpoint);

//设置 ContractFilter 和 AddressFilter
endpoint.ContractFilter = new MatchAllMessageFilter();
endpoint.AddressFilter = new MatchAllMessageFilter();

host.ChannelDispatchers.Add(dispatcher);
}
}

```

下面大体上介绍一下创建 ChannelDispatcher 的逻辑。首先创建绑定对象, 该绑定由两个绑定元素构成 (TextMessageEncodingBindingElement 和 HttpTransportBindingElement)。因为

元数据请求消息就是单纯的 HTTP-GET 请求消息，并不是一个 SOAP，所以需要 will 将 `HttpTransportBindingElement` 的消息版本设为 `None`，并将 `Method` 属性（这是一个内部属性）设为 `GET`。

然后利用创建的绑定对象创建信道监听器，并借此创建信道分发器。当信道分发器成功创建后，开始创建终结点分发器对象，并定制它的分发运行时。这其中包括创建分发操作（`DispatchOperation`）及相关的消息格式化器和操作执行器。然后是我们熟悉的对实例上下文提供者和单例实例上下文的设定。最后需要设置终结点分发器的两个消息筛选器（契约筛选器和地址筛选器），在这里将它们设置成 `MatchAllMessageFilter` 类型，使之能够匹配所有的请求消息。

待分发运行时被成功定制后，将创建的终结点分发器添加到信道分发器的 `Endpoints` 列表中。最终再将信道分发器添加到 `ServiceHost` 的 `ChannelDispatchers` 列表中。

`CreateHttpGetChannelDispatcher` 方法被创建之后，需要在 `ServiceMetadataBehaviorAttribute` 添加两个属性，即 `HttpGetEnabled` 和 `HttpGetUrl`。前者表示是否采用基于 HTTP-GET 的元数据发布模式，后者指定元数据发布的地址。如果 `HttpGetEnabled` 属性为 `True`，则在 `ApplyDispatchBehavior` 方法中调用 `CreateHttpGetChannelDispatcher` 方法为当前 `ServiceHost` 创建用于处理 HTTP-GET 形式的元数据请求。

```
[AttributeUsage(AttributeTargets.Class)]
public class ServiceMetadataBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public bool HttpGetEnabled { get; set; }
    public string HttpGetUrl { get; set; }
    public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
        ServiceHostBase serviceHostBase)
    {
        MetadataSet metadata = GetExportedMetadata(serviceDescription);
        CustomizeMexEndpoints(serviceDescription, serviceHostBase, metadata);
        if (this.HttpGetEnabled)
        {
            CreateHttpGetChannelDispatcher(serviceHostBase,
                new Uri(this.HttpGetUrl), metadata);
        }
    }
}
```

现在就可以通过下面的方式将 `ServiceMetadataBehaviorAttribute` 应用到我们的 `CalculatorService`，并通过 `HttpGetUrl` 属性指定原数据发布的目标地址。

```
[ServiceMetadataBehavior(HttpGetEnabled = true, HttpGetUrl =
    "http://127.0.0.1:9999/calculatorservice/mex")]
public class CalculatorService : ICalculator, IMetadataProvisionService
{
    //省略成员
}
```

如果 CalculatorService 被成功寄宿, 直接通过浏览器访问元数据发布的地址 (http://127.0.0.1:9999/calculatorservice/mex), 可以看到与图 2-8 一样的结果。

2.4 元数据的获取和导入

服务的元数据以 MetadataSet 的形式被 MetadataExporter 导出, 并最终通过 WS-MEX 或者 HTTP-GET 协议发布出来, 那么服务的消费者就可以采用相应的协议获取元数据。元数据的发布方式决定了其对应的获取方式, 也就是说如果元数据通过 MEX 终结点的形式发布, 我们就可以创建相匹配的 MEX 终结点, 通过向元数据资源的目标地址发送请求的方式获取元数据。如果采用 HTTP-GET 的形式发布元数据, 那么我们仅仅需要对目标地址发送 Method 为 GET 的 HTTP 请求, 即可获取相应的元数据资源。

WCF 客户端元数据架构体系为我们创建了一系列的组件和工具帮助我们获取元数据, 但是暂不对它们进行介绍。对于本书的读者来说, 如果对前面的内容有了大致的了解, 相信根本不需要这些系统提供的组件和工具, 就可以通过自己的方式实现对元数据的获取。现在自己动手实现元数据的获取。

2.4.1 自己动手实现元数据的获取

元数据的发布方式决定了元数据的获取行为。WCF 服务元数据架构体系通过 ServiceMetadataBehavior 服务行为实现了基于 WS-MEX 和 HTTP-GET 的元数据发布, 针对这两种不同的协议, 元数据获取的实现方式也是不同的。下面首先来实现基于 WS-MEX 的元数据获取方式。

1. 基于 WS-MEX 的元数据获取

ServiceMetadataBehavior 服务行为通过创建 MEX 终结点实现了基于 WS-MEX 的元数据的发布, 这个过程相当于是在服务端寄宿一个元数据服务。我们通过服务调用的形式获取元数据。

由于 MEX 终结点与一般意义上的终结点并没有本质的不同, 因此只需要创建与服务元数据发布方相匹配的终结点, 向目标地址发送期望的请求消息, 即可通过回复消息的形式获取元数据信息。现在以我们熟悉的计算服务为例, 在服务寄宿的时候通过以下的配置为该服务添加一个 MEX 终结点, 采用的 MEX 绑定和地址分别为: mexHttpBinding 和 http://127.0.0.1:9999/calculatorservice/mex。

```
<configuration>
  <system.serviceModel>
```

```

<services>
  <service name="Artech.WcfServices.Service.CalculatorService"
    behaviorConfiguration="mexBehavior">
    <endpoint address=" http://127.0.0.1:3721/calculatorservice"
      binding="ws2007HttpBinding"
      contract="Artech.WcfServices.Service.Interface.ICalculator"/>
    <endpoint address="http://127.0.0.1:9999/calculatorservice/mex"
      binding="mexHttpBinding"
      contract="IMetadataExchange"/>
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name="mexBehavior">
      <serviceMetadata/>
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

下面的代码展现了客户端获取元数据的程序, 这和一般的服务调用并无二致。首先通过 `MetadataExchangeBindings` 的 `CreateMexHttpBinding` 方法创建 MEX 绑定, 然后使用该绑定和地址 (元数据的目标地址: `http://127.0.0.1:9999/calculatorservice/mex`) 创建 `ChannelFactory<TChannel>` 对象 (由于 MEX 终结点契约类型为 `IMetadataExchange`, 这里的 `TChannel` 类型为 `IMetadataExchange`)。由于 MEX 终结点契约 `IMetadataExchange` 的 `Get` 方法的输入参数和输出参数均为 `Message` 对象, 所以我们创建 `Message` 对象, 并指定与 WS-MEX 匹配的 `Action` 后传入通过 `ChannelFactory<TChannel>` 创建的服务代理中进行服务调用。最后从回复消息中提取出包含元数据的 `MetadataSet` 对象, 并将其写入一个 XML 文件中。

```

MetadataSet metadata = null;
using (ChannelFactory<IMetadataExchange> channelFactory = new
ChannelFactory<IMetadataExchange>(MetadataExchangeBindings.CreateMexHttpBinding(),
new EndpointAddress("http://127.0.0.1:9999/calculatorservice/mex")))
{
    IMetadataExchange proxy = channelFactory.CreateChannel();
    using (proxy as IDisposable)
    {
        string action = "http://schemas.xmlsoap.org/ws/2004/09/transfer/Get";
        Message request = Message.CreateMessage(MessageVersion.Default, action);
        metadata = proxy.Get(request).GetBody<MetadataSet>();
    }
}
using (XmlWriter writer = new XmlTextWriter("metadata.xml", Encoding.UTF8))
{
    metadata.WriteTo(writer);
}
Process.Start("metadata.xml");

```

当程序成功执行后, 包含元数据的 XML 文件将会通过 IE 输出 (假设将 IE 作为默认的 XML 启动程序), 图 2-9 为运行后的截图。(S204)



图 2-9 通过 IE 显示获取的元数据（以 WS-MEX 方式发布）

2. 基于 HTTP-GET 的元数据获取

上面我们通过自定义的方式成功获取了服务端以 WS-MEX 方式发布的元数据，现在我们来实现基于 HTTP-GET 的元数据获取方式。既然服务端采用了基于 HTTP-GET 的元数据发布方式，那么就意味着可以通过简单的 HTTP 请求的方式获取相应的元数据资源。为此需要修改服务端配置以 HTTP-GET 的方式发布服务的元数据，元数据发布地址为 `http://127.0.0.1:3721/calculator/service/metadata`。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService"
        behaviorConfiguration="mexBehavior">
        <endpoint address="http://127.0.0.1:3721/calculator/service"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.ICalculator"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="mexBehavior">
          <serviceMetadata
            httpGetEnabled="true"
            httpGetUrl="http://127.0.0.1:9999/calculator/service/metadata"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

下面的代码以 HTTP-GET 的形式实现了相应的元数据获取。在这段代码中，通过指定目标地址创建了一个 `HttpWebRequest` 对象向元数据的发布地址发送请求。获取的元数据将以 `HttpWebResponse` 的形式返回。由于获取的元数据实际上是一个 WSDL 文档，所以我们可以通过 `ServiceDescription` 的 `Read` 方法直接读取生成一个 `ServiceDescription` 对象，并最终通过 `MetadataSection` 的静态方法 `CreateFromServiceDescription` 将其转换成一个 `MetadataSection` 对象。该 `MetadataSection` 对象被最终添加到创建的 `MetadataSet` 中，并被写入一个 XML 文件。

```

MetadataSet metadata = new MetadataSet();
string address = "http://127.0.0.1:9999/calculator/service/metadata";
HttpWebRequest request = (HttpWebRequest)WebRequest.Create(address);
request.Method = "Get";
HttpWebResponse response = (HttpWebResponse)request.GetResponse();
using (XmlReader reader =
    XmlDictionaryReader.CreateTextReader(response.GetResponseStream(), new
    XmlDictionaryReaderQuotas()))
{
    System.Web.Services.Description.ServiceDescription serviceDesc =
        System.Web.Services.Description.ServiceDescription.Read(reader);
    metadata.MetadataSections.Add(MetadataSection.CreateFromService
        Description(serviceDesc));
}
using (XmlWriter writer = new XmlTextWriter("metadata.xml", Encoding.UTF8))
{
    metadata.WriteTo(writer);
}
Process.Start("metadata.xml");

```

当上面的应用程序成功执行后, 包含获取的元数据的 XML 将会通过 IE 打开。如图 2-10 所示, 通过两种方式获取的元数据本质上是相同的。不过可能细心的读者会发现与上面的例子 (WS-MEX) 获取的 MetadataSet 不同, 通过 HTTP-GET 获取的 MetadataSet 仅仅包含一个元数据方言 (Dialect) 为 WSDL 的 MetadataSection。这是因为前面的例子实际上将 WSDL 中引用 (通过终结点地址或者资源地址) 的内容都生成了相应的 MetadataSection, 在这里并没有做这些工作。(S205)

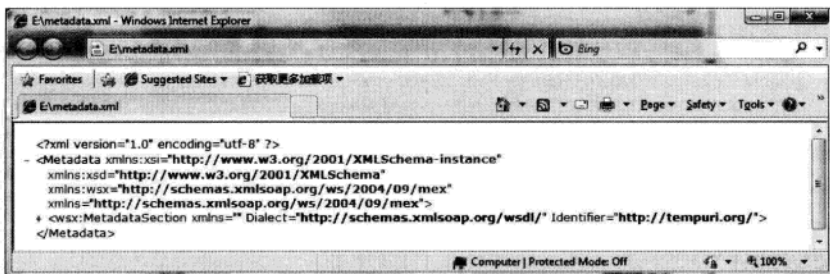


图 2-10 通过 IE 显示获取的元数据 (以 HTTP-GET 方式发布)

3. 通过 MetadataExchangeClient 获取元数据

上面我们通过自己的方式实现了对基于两种不同协议发布的元数据的获取, 可以发现具体实现的逻辑并不复杂。WCF 服务端元数据框架体系为我们创建了相应组件, 使我们很容易地获取目标服务的元数据, 这就是接下来要着重介绍的 System.ServiceModel.Description.MetadataExchangeClient。

在 WCF 元数据框架体系中, MetadataExchangeClient 是专门用于获取发布的元数据的客户端组件。无论是采用 WS-MEX, 还是 HTTP-GET 协议发布出来的元数据, 都可以通过 MetadataExchangeClient 获取。下面给出了 MetadataExchangeClient 的基本定义。


```

public class MetadataExchangeClient
{
    //构造函数
    public MetadataExchangeClient();
    public MetadataExchangeClient(Binding mexBinding);
    public MetadataExchangeClient(EndpointAddress address);
    public MetadataExchangeClient(string endpointConfigurationName);
    public MetadataExchangeClient(Uri address, MetadataExchangeClientMode mode);

    //BeginGetMetadata|EndGetMetadata
    public IAsyncResult BeginGetMetadata(AsyncCallback callback, object
        asyncState);
    public IAsyncResult BeginGetMetadata(EndpointAddress address,
        AsyncCallback callback, object asyncState);
    public IAsyncResult BeginGetMetadata(Uri address, MetadataExchangeC
        lientMode mode, AsyncCallback callback, object asyncState);
    public MetadataSet EndGetMetadata(IAsyncResult result);

    //GetMetadata
    public MetadataSet GetMetadata();
    public MetadataSet GetMetadata(EndpointAddress address);
    public MetadataSet GetMetadata(Uri address, MetadataExchangeClientMode mode);

    //GetChannelFactory & GetWebRequest
    protected internal virtual ChannelFactory<IMetadataExchange>
        GetChannelFactory(EndpointAddress metadataAddress, string dialect, string
        identifier);
    protected internal virtual HttpWebRequest GetWebRequest(Uri location,
        string dialect, string identifier);

    public int MaximumResolvedReferences { get; set; }
    public bool ResolveMetadataReferences { get; set; }
    public TimeSpan OperationTimeout { get; set; }
    public ICredentials HttpCredentials { get; set; }
    public ClientCredentials SoapCredentials { get; set; }
}

```

WS-MEX 是 `MetadataExchangeClient` 默认的元数据获取方式，可以传入 MEX 终结点相应的元素（元数据发布地址、MEX 绑定）创建该对象，也可以采用配置的方式提供 MEX 终结点（通过终结点配置名称构建 `MetadataExchangeClient` 对象）。如果需要创建基于 HTTP-GET 模式获取元数据的 `MetadataExchangeClient` 对象，可以通过在 `GetMetadata` 方法中通过 `mode` 参数来控制。该参数类型为具有如下定义的 `System.ServiceModel.Description.MetadataExchangeClientMode` 枚举，两个成员 `MetadataExchange` 和 `HttpGet` 分别代表 WS-MEX 和 HTTP-GET 两种元数据发布（获取）模式。

```

public enum MetadataExchangeClientMode
{
    MetadataExchange,
    HttpGet
}

```

`MetadataExchangeClient` 定义了一系列的 `GetMetadata` 和 `BeginGetMetadata|EndGetMetadata` 方法以实现同步和异步的方式获取元数据。获取出来的元数据以 `MetadataSet` 的形式返回。在上面的例子中，我们通过 `ChannelFactory<IMetadataExchange>` 创建的服务代理

和 `HttpWebRequest` 对象实现了对元数据的获取, 实际上 `MetadataExchangeClient` 内部采用的机制和我提供的实例完全是一样的。

`MetadataExchangeClient` 中提供了两个虚方法 `GetChannelFactory` 和 `GetWebRequest` 获取相应的 `ChannelFactory<IMetadataExchange>` 和 `HttpWebRequest` 对象。如果你需要对 `MetadataExchangeClient` 进行相应的扩展, 采用一些自定义的获取行为, 可以继承 `MetadataExchangeClient` 类并重写这两个方法。

元数据数据可以以内联的方式直接置于 WSDL 相应的元素中, 也可以通过引用的方式导入 WSDL 中。常见的引用方式包括 WS-Addressing 终结点引用 (Endpoint Reference) 和资源地址引用 (Location Reference)。在默认的情况下, 被引用的元数据将被解析和获取。可以通过 `ResolveMetadataReferences` 属性控制是否需要对元数据引用进行解析。属性 `MaximumResolvedReferences` 则用于控制最大允许解析的元数据引用的数量, 默认是 10。`OperationTimeout` 属性表示进行元数据获取操作的超时时限。`HttpCredentials` 和 `SoapCredentials` 则用于指定获取元数据请求的客户端凭证。

同样针对上面的例子, 如果采用 `MetadataExchangeClient` 作为元数据获取的手段, 我们的程序将会变得更加简洁。下面的代码模拟了通过 `MetadataExchangeClient` 对象以 HTTP-GET 的方式进行元数据获取, 由于将 `ResolveMetadataReferences` 的属性设成 `false`, 意味着被 WSDL 引用的元数据将不会被解析, 所以程序运行后将会得到如图 2-10 所示的结果, 否则输出的结果如图 2-9 所示。(S206)

```
Uri address = new Uri("http://127.0.0.1:9999/calculator/service/metadata");
MetadataExchangeClient metadataExchangeClient = new
MetadataExchangeClient(address, MetadataExchangeClientMode.HttpGet);
metadataExchangeClient.ResolveMetadataReferences = false;
MetadataSet metadata = metadataExchangeClient.GetMetadata();
using (XmlWriter writer = new XmlTextWriter("metadata.xml", Encoding.UTF8))
{
    metadata.WriteTo(writer);
}
Process.Start("metadata.xml");
```

2.4.2 MetadataImporter 与元数据导入

借助于 `MetadataExchangeClient`, 可以很容易地获取通过 WS-MEX 或者 HTTP-GET 模式发布出来的元数据, 获取的元数据以 `MetadataSet` 的形式体现。要利用获取的元数据进行服务调用, 还需要最后一个步骤, 那就是将以 `MetadataSet` 形式表示的元数据转换成相应的 `ServiceEndpoint` 对象。这个步骤实际上和我们上面介绍的元数据导出是相反的过程, 所以将此过程称为元数据导入 (Importing Metadata)。

服务端通过 `MetadataExporter` 实现元数据的导出, 客户端利用与之匹配的 `MetadataImporter` 对象实现元数据的导入。服务端定义了一个 `Wsdlexporter` 将 `ServiceEndpoint`

对象导出成基于 WSDL 的 MetadataSet，客户端通过相应的 WsdlImporter 对象将基于 WSDL 的 MetadataSet 转换成 ServiceEndpoint。下面首先对 MetadataImporter 和 WsdlImporter 进行简单的介绍。

1. MetadataImporter 与 WsdlImporter

WCF 的客户端元数据架构体系和服务端是相对的，服务端将 ServiceEndpoint 对象导出成元数据并将其发布，客户端则获取发布的元数据并借此导入成相应的 ServiceEndpoint 对象。由于导出和导入是一对逆操作，所以 MetadataImporter 定义了如下一套与 MetadataExpoprtter 相反的操作。

```
public abstract class MetadataImporter
{
    public abstract Collection<ContractDescription> ImportAllContracts();
    public abstract ServiceEndpointCollection ImportAllEndpoints();

    public Collection<MetadataConversionError> Errors { get; }
    public Dictionary<XmlQualifiedName, ContractDescription> KnownContracts
    { get; }
    public KeyedByTypeCollection<IPolicyImportExtension> PolicyImportExtensions
    { get; }
    public Dictionary<object, object> State { get; }
}
```

MetadataImporter 通过 ImportAllContracts 和 ImportAllEndpoints 将元数据导入生成相应的 ContractDescription 集合和 ServiceEndpointCollection 对象。Errors 和 State 属性与 MetadataExpoprtter 的同名属性具有一样的作用，即用于保存元数据导入过程中的错误和状态信息。KnownContracts 属性是一个值类型为 ContractDescription 的字典，用于保存已知服务契约，该字典的键为服务契约的有效名称。PolicyImportExtensions 则是一组 IPolicyImportExtension 对象列表，用于动态导入一些自定义 WS-Policy 策略断言。

MetadataImporter 是一个抽象类型，定义了元数据导入的基本操作（ImportAllContracts 和 ImportAllEndpoints 方法）。WCF 定义了继承自 MetadataImporter 的 WsdlImporter 类实现基于 WSDL 元数据的导入。下面的代码片段给出了 WsdlImporter 的基本定义。

```
public class WsdlImporter : MetadataImporter
{
    //构造函数
    public WsdlImporter(MetadataSet metadata);
    public WsdlImporter(MetadataSet metadata, IEnumerable<IPolicyImport
    Extension>
        policyImportExtensions, IEnumerable<IWsdlImportExtension>
        wsdlImportExtensions);
    public WsdlImporter(MetadataSet metadata, IEnumerable<IPolicyImport
    Extension>
        policyImportExtensions, IEnumerable<IWsdlImportExtension>
        wsdlImportExtensions, MetadataImporterQuotas quotas);

    //ImportContract& ImportAllContracts
```

```

public ContractDescription ImportContract(PortType wsdlPortType);
public override Collection<ContractDescription> ImportAllContracts();

//ImportBinding(s)
public Binding ImportBinding(Binding wsdlBinding);
public Collection<Binding> ImportAllBindings();

//ImportEndpoint(s)& ImportAllEndpoints
public ServiceEndpoint ImportEndpoint(Port wsdlPort);
public ServiceEndpointCollection ImportEndpoints(Binding wsdlBinding);
public ServiceEndpointCollection ImportEndpoints(PortType wsdlPortType);
public ServiceEndpointCollection ImportEndpoints(Service wsdlService);
public override ServiceEndpointCollection ImportAllEndpoints();

public ServiceDescriptionCollection WsdlDocuments { get; }
public KeyedByTypeCollection<IWsdImportExtension> WsdImportExtensions
    { get; }
public XmlSchemaSet XmlSchemas { get; }
}

```

WsdImporter 定义了三组导入 (Import) 方法。ImportContract/ImportAllContracts 将元数据导入生成 ContractDescription 对象和 ContractDescription 集合。ImportBinding(s)导入元数据生成 Binding 对象和 Binding 集合。而 ImportEndpoint(s)则利用导入的元数据生成 ServiceEndpoint 对象和 ServiceEndpoint 集合。需要注意的是, ImportBinding 和 ImportEndpoint(s) 方法中的 Binding、PortType 和 Service 定义在 System.Web.Services.Description.Binding 命名空间下, 它们实际上分别是对 WSDL 中同名节点的描述。

2. 实例演示: 通过 WsdImporter 创建终结点进行服务调用 (S207)

只要能够创建与服务端相匹配的终结点, 就能够实现正常的服务调用。在接下来的实例演示中, 我们将利用 MetadataExchangeClient 获取服务的元数据, 并使用 WsdImporter 将获取的元数据导入成 ServiceEndpoint 对象, 并最终实现正常的服务调用。

依然采用我们熟悉的计算服务的例子, 下面是该服务相应的服务契约、服务类型的定义和寄宿该服务采用的配置。服务的元数据通过 WS-MEX 模式发布出来, 发布的地址为 <http://127.0.0.1:3721/calculator/mex>。

```

<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService"
        behaviorConfiguration="mexBehavior">
        <endpoint address="http://127.0.0.1:3721/calculator/mex"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.ICalculator"/>
        <endpoint address="http://127.0.0.1:9999/calculator/mex"
          binding="mexHttpBinding"
          contract="IMetadataExchange" />
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>

```

```

        <behavior name="mexBehavior">
            <serviceMetadata/>
        </behavior>
    </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

接下来就可以通过下面的方式对该服务进行调用了。我们先创建 `MetadataExchangeClient` 对象并利用它获取包含元数据的 `MetadataSet` 对象，并利用该对象创建 `WsdImporter` 对象。接下来将服务契约 `ICalculator` 添加到该 `WsdImporter` 的已知契约列表中，并调用 `ImportAllEndpoints` 方法得到导入的 `ServiceEndpoint` 列表。最后根据导出的 `ServiceEndpoint` 对象创建 `ChannelFactory<ICalculator>` 对象，并创建服务代理进行服务调用。

```

Binding binding = MetadataExchangeBindings.CreateMexHttpBinding();
EndpointAddress address = new
EndpointAddress("http://127.0.0.1:9999/calculator/service/mex");
MetadataExchangeClient metadataExchangeClient = new
MetadataExchangeClient(binding);
MetadataSet metadata = metadataExchangeClient.GetMetadata(address);
WsdImporter wsdlImporter = new WsdImporter(metadata);
//添加已知契约类型
ContractDescription contract =
ContractDescription.GetContract(typeof(ICalculator));
wsdlImporter.KnownContracts.Add(new XmlQualifiedName(contract.Name,
contract.Namespace), contract);
ServiceEndpointCollection endpoints = wsdlImporter.ImportAllEndpoints();
using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>(endpoints[0]))
{
    ICalculator calculator = channelFactory.CreateChannel();
    Console.WriteLine("x + y = {2} when x = {0} and y = {1}", 1, 2,
        calculator.Add(1, 2));
}

```

3. WSDL 导入扩展与策略导入扩展

通过前面的介绍，我们知道了 `WsdExporter` 通过 WSDL 导出扩展和策略导出扩展实现了将一些自定义的 WSDL 元素和 WS-Policy 策略写入到最终导出的 WSDL。对于这些导出自定义的 WSDL 元素和 WS-Policy 策略，在进行元数据导入的时候，如何将其应用到导入生成的 `ServiceEndpoint` 上面呢？这就需要两个相对应的扩展，即 WSDL 导入扩展和策略导入扩展。

(1) WSDL 导入扩展 (WSDL Import Extension)

实现 WSDL 导入扩展的类型均实现了具有如下定义的 `IWsdImportExtension` 接口。`IWsdImportExtension` 和 `IWsdExportExtension` 的定义相对应，`ImportContract` 和 `ImportEndpoint` 方法用于将相应的 WSDL 元素导入到 `WsdContractConversionContext` 对象中，并最终应用到 `ServiceEndpoint` 的契约或者绑定上。额外的 `BeforeImport` 方法用户执行一些预导入的操作。

```
public interface IWsdlImportExtension
{
    void BeforeImport(ServiceDescriptionCollection wsdlDocuments,
        XmlSchemaSet xmlSchemas, ICollection<XmlElement> policy);
    void ImportContract(WsdlImporter importer, WsdlContractConversionContext
        context);
    void ImportEndpoint(WsdlImporter importer, WsdlEndpointConversionContext
        context);
}
```

WSDL 导出扩展主要是以行为（契约行为、操作行为和终结点行为）和绑定元素（BindingElement）的形式实现的，而 WSDL 导入扩展则需要进行单独定义。我们可以实现 IWsdlImportExtension 接口创建自定义的扩展，并通过构造函数的形式传递给 WsdlImporter 对象，在 WsdlImporter 的构造函数重载中具有一个 wsdlImportExtensions 参数，用于指定 WSDL 导入扩展列表。

```
public class WsdlImporter : MetadataImporter
{
    //其他成员
    public WsdlImporter(MetadataSet metadata, IEnumerable<IpolicyImport
        Extension>
        policyImportExtensions, IEnumerable<IWsdlImportExtension>
        wsdlImportExtensions);
    public WsdlImporter(MetadataSet metadata, IEnumerable<IpolicyImport
        Extension>
        policyImportExtensions, IEnumerable<IWsdlImportExtension>
        wsdlImportExtensions, MetadataImporterQuotas quotas);
}
```

自定义的 WSDL 导入扩展还可以直接定义通过配置的方式提供。只需要将相应的扩展类型定义在 ServiceMode 的 <client>/<metadata>/<wsdlImporters> 配置项中就可以了。下面的配置来源于 machine.config，其中定义了一个基于 WF 的 WSDL 导入扩展：ContextBindingElementImporter。

```
<system.serviceModel>
  <client>
    <metadata>
      <wsdlImporters>
        <extension
          type="System.ServiceModel.Channels.ContextBindingElementImporter,
            system.workflowservices,
            Version=4.0.0.0, Culture=neutral,
            PublicKeyToken=31bf3856ad364e35,
            processorArchitecture=MSIL"/>
      </wsdlImporters>
    </metadata>
  </client>
</system.serviceModel>
```

WCF 定义了一系列的 WSDL 扩展，比如为消息编码绑定元素和传输绑定元素定义了 MessageEncodingBindingElementImporter 和 DataContractSerializerMessageContractImporter。在初始化 WsdlImporter 的时候，这些系统定义的扩展和配置的扩展将会被自动添加到 WSDL

导入扩展列表中。当 `WsdImporter` 执行导入操作的时候, 这些扩展会被逐个调用, 从而实现将相应的 WSDL 元素应用到导入生成的 `ServiceEndpoint` 对象上。

(2) 策略导入扩展 (Policy Import Extension)

WSDL 导入扩展并不负责对 WS-Policy 策略的导入, WCF 通过另外一个扩展实现对 WS-Policy 策略的导入, 即策略导入扩展。策略导入扩展实现了如下定义的 `System.ServiceModel.Description.IPolicyImportExtension` 接口, 该接口唯一的 `ImportPolicy` 用于实现对 WS-Policy 策略的导入。两个参数分别表示实施策略导入的 `MetadataImporter` 对象和策略导出过程的上下文 `PolicyConversionContext`。

```
public interface IPolicyImportExtension
{
    void ImportPolicy(MetadataImporter importer, PolicyConversionContext context);
}
```



第3章 事务

(Transaction)

事务提供一种机制将一个活动涉及的所有操作纳入到一个不可分割的执行单元。组成事务的所有操作只有在所有操作均能正常执行的情况下方能提交，如果其中任一操作执行失败，都将导致整个事务的回滚。简单地说，事务提供一种“要么什么都不做，要么做全套（All or Nothing）”机制。



在一个基于 SOA 架构的分布式系统体系中，服务成为基本的功能提供单元。无论是与业务流程无关的基础功能，还是具体的业务逻辑，均实现在相应的服务之中。服务对外提供统一的接口，服务之间采用标准的通信方式进行交互，各个单一的服务经过有效的组合、编排成为一个有机的整体对外提供更完整的功能。

在这样一个分布式系统中某个活动（Activity）的实现往往需要跨越单个服务的边界。如何协调多个服务之间的关系涉及服务协作（Service Coordination）问题。一个分布式的活动可能会执行几秒钟（比如银行转账），也可能执行几分钟、几个小时、几天甚至更长时间（比如移民局处理移民的申请）。

事务无疑是属于短暂运行服务协作（Short-Running Service Coordination）的范畴。WCF 建立在 Windows 成熟的事务基础架构（MS DTC）和 .NET 事务组件（比如 System.Transactions 事务）之上，提供了一个声明式编程模型。事务编程结合相应的配置，使我们可以很容易地将事务应用到 WCF 服务之上。

3.1 WCF 需要怎样的事务控制

要让读者熟练掌握 WCF 的事务编程，深刻理解 WCF 内部的事务实现原理，首先要知道 WCF 服务到底需要怎样的服务控制方式。本节将会为你展现几种典型的分布式事务应用场景。在这之前先来了解一下事务的概念和经典的 ACID 属性。

3.1.1 什么是事务

事务提供一种机制将一个活动涉及的所有操作纳入一个不可分割的执行单元。组成事务的所有操作只有在均能正常执行的情况下方能提交，如果其中任一操作执行失败，都将导致整个事务的回滚。简单地说，事务提供一种“要么什么都不做，要么做全套（All or Nothing）”机制。事务具有如下 4 个属性，根据其首字母，我们一般将其称为事务的 ACID 属性。

- 原子性（Atomicity）：“原子”这个词的本义就是不可分割的意思。事务的原子性意味着一个事务的所有操作被捆绑成一个整体，所有的操作要么全部执行，要么都不执行。
- 一致性（Consistence）：事务的原子性确保一个事务不会破坏数据的一致性。如果事务成功提交，数据的状态是组成事务的所有操作按照事先编排的方式执行的结果。如果事务的任何一个中间步骤出错，整个事务回滚并将数据恢复到原来的状态。事务只会将数据状态从一个一致性状态转换到另一个一致性状态。
- 隔离性（Isolation）：从事务的外部来看，事务的一致性实现了数据在两个一致性状态之间的转换。但是从事务内部来看，组成事务的各个操作是按照一定的逻辑顺序执行的，所以数据具有位于两个一致性状态的“中间状态”。但是这种中间状态被隔离于事务内部，对于事务外部是不可见的。

- 持久性 (Durability): 持久性的意思是一旦成功提交, 基于持久化资源 (比如数据库) 的数据将会被持久化, 对数据的改变是永久性的。

事务最初来源于数据库管理系统 (DBMS), 反映的是对存储于数据库中的数据的操作。除了主流的关系型数据库管理系统, 比如 SQL Server, Oracle 和 DB2 等提供对事务的支持, 基于事务的数据操作方式也可以应用到其他一些数据存储资源 (比如 MSMQ)。自 Windows Vista 开始, 文件系统 (NTFS) 和注册表也成为事务型资源 (Transactional Resource)。

3.1.2 事务的显式控制

虽然事务型资源家族成员越来越多, 但是不可否认的是, 数据库还是我们使用频率最高的事务型资源。稍微有一些经验的开发人员, 应该都在存储过程 (Stored Procedure) 中编写过基于事务的 SQL, 或者编写过基于 ADO.NET 事务的代码。我们对事务的进一步介绍就从这里说起。

1. SQL 中的事务处理

无论是基于 SQL Server 的 T-SQL, 还是基于 Oracle 的 PL/SQL, 都对事务提供了原生的支持。有意思的是, T-SQL 中的 T 本身指的就是事务 (Transaction)。以 T-SQL 为例, 我们可以通过如下三个 SQL 语句实现事务的启动、提交与回滚。

- BEGIN TRANSACTION: 开始一个事务。
- COMMIT TRANSACTION: 提交所有位于 BEGIN TRANSACTION 和 COMMIT TRANSACTION 之间的操作。
- ROLLBACK TRANSACTION: 回滚所有位于 BEGIN TRANSACTION 和 COMMIT TRANSACTION 之间的操作。

下面举一个很典型的银行转账操作的例子, 这个例子将会贯穿于本章的始终。为此我们先创建一个最为简单的用于存储账户的 T_ACCOUNT 表。整个表包括三个字段 (ID、NAME 和 BALANCE), 它们分别代表银行账号的 ID、名称和余额。可以执行如下一段 T-SQL 来创建这个表。

```
CREATE TABLE [dbo].[T_ACCOUNT] (
    [ID]          VARCHAR(50)      PRIMARY KEY,
    [NAME]        NVARCHAR(50)     NOT NULL,
    [BALANCE]     FLOAT            NOT NULL)
GO
```

银行转账是一个简单的复合型操作, 由存储和提取这两个基本的操作共同完成, 即从一个账户中提取相应金额转入另一个账户。对数据完整性的要求是我们必须将这两个单一的操作纳入同一个事务。如果我们通过一个存储过程来完成整个转账的流程, 具体的 SQL 应该采用下面的写法。

```

CREATE Procedure P_TRANSFER
(
    @fromAccount    VARCHAR(50),
    @toAccount      VARCHAR(50),
    @amount         FLOAT
)
AS

--确保账户存在性
IF NOT EXISTS(SELECT * FROM [dbo].[T_ACCOUNT] WHERE ID = @fromAccount)
BEGIN
    RAISERROR ('AccountNotExists',16,1)
    RETURN
END
IF NOT EXISTS(SELECT * FROM [dbo].[T_ACCOUNT] WHERE ID = @toAccount)
BEGIN
    RAISERROR ('AccountNotExists',16,1)
    RETURN
END
--确保余额充足性
IF NOT EXISTS(SELECT * FROM [dbo].[T_ACCOUNT] WHERE ID = @fromAccount AND
BALANCE >= @amount)
BEGIN
    RAISERROR ('LackofBalance',16,1)
    RETURN
END
--转账
BEGIN TRANSACTION
    UPDATE [dbo].[T_ACCOUNT] SET BALANCE = BALANCE - @amount WHERE ID =
    @fromAccount
    IF @@ERROR <> 0
        BEGIN
            ROLLBACK TRANSACTION
        END
    UPDATE [dbo].[T_ACCOUNT] SET BALANCE = BALANCE + @amount WHERE ID = @toAccount
    IF @@ERROR <> 0
        BEGIN
            ROLLBACK TRANSACTION
        END
    COMMIT TRANSACTION
GO

```

2. ADO.NET 事务控制

对于.NET 开发人员,我们可以直接利用 ADO.NET 将基于单个数据库连接的多个操作纳入同一个事务之中。同样上面的银行转账事务为例,这次我们将整个转账作为一个服务 (BankingService) 的一个操作 (Transfer)。下面的代码通过一种与具体数据库类型无关的 ADO.NET 编程模式实现了整个银行转账操作,最终的转账通过调用一个存储过程实现。

```

public class BankingService : IBankingService
{
    //其他操作
    public void Transfer(string fromAccountId, string toAccountId, double
    amount)
    {

```

```

string connectionStringName = "BankingDb";
string connectionString = ConfigurationManager.ConnectionStrings
[connectionStringName].ConnectionString;
string providerName = ConfigurationManager.ConnectionStrings[connection-
StringName].ProviderName;
DbProviderFactory dbProviderFactory =
    DbProviderFactories.GetFactory(providerName);
using (DbConnection connection = dbProviderFactory.CreateConnection())
{
    connection.ConnectionString = connectionString;
    DbCommand command = connection.CreateCommand();
    command.CommandText = "P_TRANS";
    command.CommandType = CommandType.StoredProcedure;

    DbParameter parameter = dbProviderFactory.CreateParameter();
    parameter.ParameterName = BuildParameterName("fromAccount");
    parameter.Value = fromAccountId;
    command.Parameters.Add(parameter);

    parameter = dbProviderFactory.CreateParameter();
    parameter.ParameterName = BuildParameterName("toAccount");
    parameter.Value = toAccountId;
    command.Parameters.Add(parameter);

    parameter = dbProviderFactory.CreateParameter();
    parameter.ParameterName = BuildParameterName("amount");
    parameter.Value = amount;
    command.Parameters.Add(parameter);

    connection.Open();
    using (DbTransaction transaction = connection.BeginTransaction())
    {
        command.Transaction = transaction;
        try
        {
            command.ExecuteNonQuery();
            transaction.Commit();
        }
        catch
        {
            transaction.Rollback();
            throw;
        }
    }
}
}
}

```

3. 事务的显式控制限定于对单一资源的访问

通过在 SQL 中进行事务的控制, 只能将基于某一段 SQL 语句的操作纳入一个单一的事务中。如果采用基于 ADO.NET 的数据控制, 被纳入同一个事务的操作仅仅限于某个数据库连接。换句话说, 上面介绍的这两种对事务的显式控制仅限于对单一的本地资源的控制。

将事务的概念引入服务。倘若我们将一个单一的服务操作作为一个事务, 如果采用上述的显式事务控制的方式, 那么整个服务操作只能涉及一个单一的事务资源。服务与存取的资

源关系如图 3-1 所示。

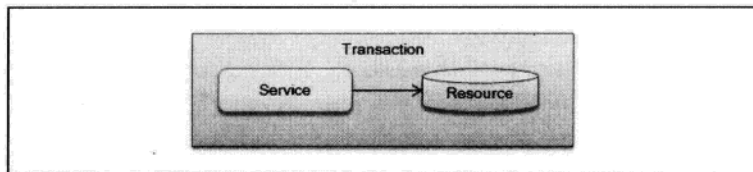


图 3-1 本地事务对单一资源的控制

上述的这种基于单一本地资源访问的事务，称为本地事务（Local Transaction）。在一个基于 SOA 分布式应用环境下，我们需要能同时将多个资源、多个服务进行统一协作的分布式事务（Distributed Transaction）。接下来介绍几种典型的分布式事务应用的场景。

3.1.3 分布式事务应用场景

对于一个分布式事务来讲，事务的参与者分布于不同的网络节点。也就是说，可以将多个事务资源纳入到一个单一的事务之中，并且这些事务资源可以分布到不同的主机上。这些承载分布式资源的主机可能处于同一个网络中，也可能处于不同的网络中。甚至说某个事务资源本质上就是一个通过 HTTP 访问的单纯的 Internet 资源。

站在 SOA 的角度来看分布式事务，意味着若将服务的某个操作视为一个单一的事务，该操作可能会访问不止一个事务资源（比如访问两个不同的数据库服务器），也可能调用另一个服务。下面介绍了三个典型的分布式事务应用场景，先从最简单的说起。

1. 将对多个资源的访问纳入同一事务

第一个分布式事务应用场景的服务操作涉及对多个事务资源的访问，如图 3-2 所示。下面列出了几种典型的场景。

- 一个服务操作访问两种类型的数据库（比如 SQL Server + Oracle）；
- 一个服务操作访问相同数据库，但是相应的数据库访问是基于不同的数据连接；
- 一个服务操作处理访问数据库资源，还需要访问其他非数据库的事务资源。

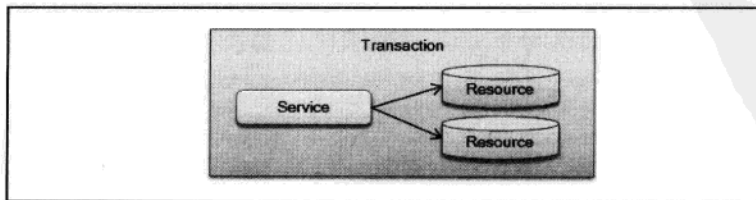


图 3-2 单一服务对多个事务资源的访问

2. 将对多个服务的调用纳入同一事务

如果一个服务操作需要调用另外一个服务，这时的事务就需要跨越多个服务了。在这种情况下，起始于某个服务的事务在调用另外一个服务的时候，需要以某种机制流转到另外一个服务，以使被调用的服务访问的资源自动注册到流入的事务中。WCF 事务体系的一项重要的目标就是实现事务的流转 (Transaction Flow)。图 3-3 反映了这样一个跨越多个服务的分布式事务。

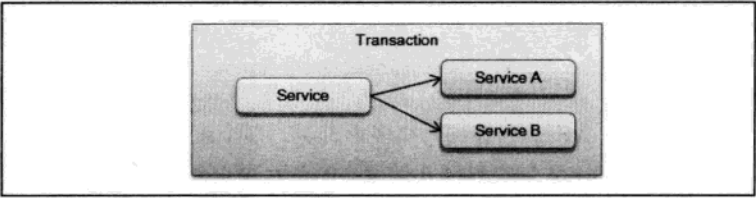


图 3-3 跨越多个服务的事务

3. 将对多个资源和服务的访问纳入同一个事务

如果将上面这两种场景（一个服务可以调用多个事务资源，也可以调用其他服务）结合在一起，整个事务的参与者将会组成如图 3-4 所示的树形拓扑结构。在一个基于分布式事务的服务调用中，事务的发起者和提交者均是同一个，它可以是整个调用的客户端，也可以是客户端最先调用的那个服务。

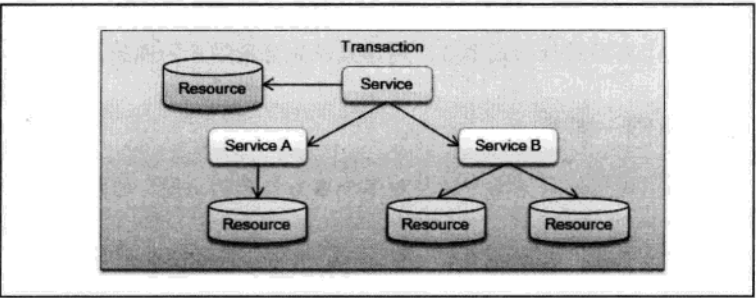


图 3-4 基于 SOA 分布式事务拓扑结构

较之基于单一资源访问的本地事务，分布式事务的实现机制要复杂得多。WCF 的事务系统充分利用了 Windows 平台现有的分布式事务基础架构。下面就来简单讨论一下 Windows 下的事务处理模型。

3.2 Windows 下的事务处理模型

通过上面的介绍，我们知道了 WCF 真正需要的是一个能够协调服务操作直接（通过服

务自身访问的资源)或者间接(通过被调用服务访问的资源)访问的所有资源的分布式事务管理系统,这是一个复杂的架构体系。好在 WCF 并不是另起炉灶,而是充分地利用了 Windows 现有的事务控制基础架构。本节着重讨论 Windows 事务处理模型,先来看看在这个模型中各个事务参与者各自扮演怎样的角色。

3.2.1 事务模型中的三种角色

对于所有的事务参与者,按照各自在事务生命周期各个阶段中的职责,大致扮演着如下三种角色。

- 应用(Application)、服务(Service)或者组件(Component):代表用户程序,或者是承载着某功能的服务或者组件;
- 资源管理器(RM, Resource Manager):代表用于管理具体事务型资源的软件程序,比如数据库管理系统(DBMS)或者消息队列(MSMQ)等;
- 事务管理器(TM, Transaction Manager):代表管理整个事务的中间件程序,为应用和资源管理器提供基本的事务控制服务。

1. 应用、服务和组件

事务最终是为用户程序、服务和组件服务的,它们利用了事务这种特殊的机制将一组相关的操作作为一个不可分割的整体来执行,从而确保了数据的一致性。在整个模型中,应用(服务或者应用,为了叙述简练,后续部分关于应用、服务和组件都简称为应用)主要负责如下一些事务相关的任务。

- 开始事务:事务开始的驱动者总是应用,但是并不是所有的应用都会开始一个新的事务,只有最初的应用才是事务的开启者。
- 事务的封送(Marshaling)和传播(Propagation):将应用的本地事务封送、传播给另一个应用或者资源管理器。
- 提交事务:事务的开启者同时也是事务最终的提交者,当事务相关的操作顺利完成后,最初开启事务的应用会提交该事务。

2. 资源管理器

在事务控制模型中,不论是应用还是事务管理器都不是直接地访问具体的事务型资源,而是通过资源管理器对目标资源进行存取。按照目标资源是否可被持久化,可以将相应的资源管理器分为如下两类。

- 持久化资源管理器(Durable Resource Manager):用于管理持久化资源,比如数据库管理系统和消息队列。当事务回滚的时候,具有可恢复性(Recovery)。

- 易失资源管理器 (Volatile Resource Manager): 用于管理像内存数据这样的不会被持久化的易失资源, 易失资源不具有可恢复性。

在后面介绍的实现分布式事务的两阶段提交 (2PC, Two-Phase Commit) 协议中, 对于这两种不同的资源管理器, 采用不同的登记 (Enlist) 方式。总的来说, 资源管理器在整个事务模型中主要承担如下几种职责。

- 帮助应用实现对目标资源的操作。
- 注册到相应的事务管理器, 以便事务回滚时可以从事务管理器中接收到恢复请求, 实现对数据的恢复。
- 向相应的事务管理器报告本地事务的结果。

3. 事务管理器

事务管理器是整个事务控制模型的核心和枢纽, 是它控制着事务的所有参与者, 协调整个事务从开始到完成的所有相关处理流程。事务管理器为应用和资源管理器提供一系列核心的事务性的服务, 实现事务的开始、提交和回滚。Windows 提供了三种不同的事务管理器。

(1) 轻量级事务管理器 (LTM, Lightweight Transaction Manager)

正如其名称隐含的意思一样, 轻量级事务管理器 (以下简称 LTM) 具有最小的负载, 是性能最高的事务管理器。LTM 的作用范围仅限于开启事务的应用程序域 (AppDomain) 中, 并且登记到事务中的持久化资源 (Durable Resource) 数量不能超过一个。

一般情况下, 被开启的事务就由 LTM 管理。如果事务涉及跨应用程序域的操作, 当前的事务会被封送传播到另一个执行上下文中, 此时事务将脱离 LTM 的管辖。基于 LTM 的事务中虽然仅仅允许登记唯一一个持久化资源, 但是可以同时登记 (Enlist) 多个易失型资源 (Volatile)。当第二个持久化资源被登记到当前事务中时, 该事务也将脱离 LTM 的管辖。

并不是所有的持久化资源都可以登记到 LTM, 实际上到目前为止, 能够登记到 LTM 的事务型资源仅限于 SQL Server 2005 和 SQL Server 2008, 即使是同属于 Windows 平台下的 SQL Server 2000 和 MSMQ 均不支持 LTM, 更不用说 Oracle 和 DB2。我们希望微软能够和其他的厂商进行合作, 让第三方开发的事务型资源也能利用 LTM 性能的优势。

(2) 内核事务管理器 (KTM, Kernel transaction Manager)

内核事务管理器 (以下简称 KTM) 从 Windows Vista 开始被引入, 并被用于后续的 Windows Server 2008 和 Windows 7 中。引入 KTM 的主要目的在于将文件管理和注册表管理纳入事务的范畴。借助于 KTM, 我们可以以事务的方式操作 NTFS 文件系统下的文件资源和注册表资源。我们将支持事务的文件系统和注册表称为事务型的文件系统 (TxF) 和事务型注册表 (TxR)。

之所以称为内核事务管理器,是因为基于 KTM 的事务控制引擎运行在内核模式(Kernel Mode),而不是用户模式(User Mode)下。和 LTM 一样, KTM 对易失型事务资源没有限制,却只能允许一个持久化事务资源被涉及。

从上面的介绍不难看出,无论是 LTM 还是 KTM,其管辖范围仅限于本地事务,对于分布式事务却无能为力。分布式事务依赖于一个更为强大的事务管理器,就是我们接下来着重介绍的分布式事务协调器。

(3) 分布式事务协调器(DTC, Distributed Transaction Coordinator)

分布式事务协调器(简称为 DTC,或者 MS DTC,以下直接简称为 DTC)用于管理跨边界(跨应用程序域、进程、机器以至跨网络)执行的分布式事务。它采用相应的事务管理协议,比如 Ole-Tx 和 WS-Atomic Transaction (WS-AT),协调一个分布式事务中的所有参与者。

每一台机器上具有一个唯一的 DTC,它管理着本地的所有资源管理器。当事务跨越多台机器时,它们各自的 DTC 需要按照相应的协议相互协作,实现对整个事务的一致性管理。

4. 事务提升(Transaction Promotion)

以上介绍了三种不同的事务管理器类型,从功能上讲, DTC 能够协调、管理一个分布式事务涉及的所有事务型资源,而不管具体的资源分布于何处。但由于其事务控制的复杂性(一般采用两阶段提交协议)并需要进行跨进程、跨机器甚至跨网络通信,在性能上无疑是最差的。所以我们不可能在任何事务场景中都采用 DTC。所谓“牛刀虽好,不便杀鸡”,我们应该根据事务控制的需要选择性能最高的事务管理器。

但是事务是一个动态执行的操作序列,系统不可能预知完整执行整个事务所有操作后的资源登记情况,所以不可能预先为其指定一个为相应事务度身定制的事务管理器,而只能在事务具体的执行过程中,动态地选择最适合当前事务执行情况的事务管理器。Windows 采用事务提升的机制进行事务管理器的选择。

一般情况下,事务开始的时候, LTM 默认作为当前的事务管理器。随着事务操作的逐步执行,如果该事务涉及对某个内核事务资源的访问,那么自动提升到基于 KTM 的事务。无论是基于 LTM 还是 KTM 的事务,当出现如下两种情况的时候,会向基于 DTC 的事务提升。

- 事务操作涉及对多个 LTM 资源的访问或访问的资源不被 LTM 支持:比如说当事务应用开启两个基于 SQL Server 2008 (LTM 事务型资源)的连接进行数据存取,或者访问开启一个基于 Oracle (非 LTM 事务型资源)的连接进行数据存取。
- 当前事务被跨应用程序域封送(Marshaling):比如,当一个服务调用另一个服务的时候,将当前事务进行序列化以实现向被调用服务方传播。

WCF 的事务体系解决的是事务在服务之间的流转,以及对服务操作直接或者间接访问的所有事务型资源的协作,这样的事务是通过基于 DTC 的分布式事务实现的。接下来就来简单讨论一下基于 DTC 的分布式事务是如何实现的。

3.2.2 分布式事务是如何实现的

当基于 LTM 或者 KTM 的事务提升到基于 DTC 的分布式事务后, DTC 成为本机所有事务型资源管理器的管理者。当一个事务型操作超出了本机的范围, 出现了跨机器的调用后, 本机的 DTC 需要与被调用者所在机器的 DTC 进行协助。上级对下级 (包括本机 DTC 对本机所有资源管理器, 以及上下级 DTC) 的管理的前提是下级在上级那里登记, 即事务登记 (Transaction Enlist)。所有事务参与者, 包括所有资源管理器和事务管理器 (即 DTC) 在事务登记完成之后形成了一个树形的层级结构, 该结构的形成使后续的事务提交成为可能, 因此我们将其称为事务提交树 (Transaction Commit Tree)。

1. 事务登记和事务提交树

事务登记的目的在于建立起事务参与者 (主要指资源管理器和事务管理器 DTC) 之间的关系, 促进相互之间的协作。整个事务登记流程大致如图 3-5 所示。

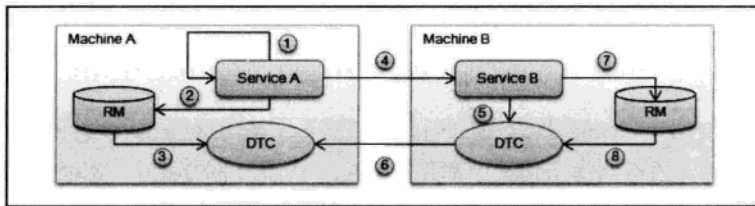


图 3-5 事务登记的流程

图 3-5 所示的事务涉及部署于两台机器 (Machine A 和 Machine B) 上的两个服务之间的交互。事务由 Service A 开启, 在调用 Service B 的时候被传播到 Machine B, 从而将分布于两台机器的资源管理器纳入到同一个事务之中。接下来对整个流程进行详细介绍。

首先, Service A 开始一个新的事务, 并将其作为当前执行上下文的环境事务 (Ambient Transaction)。当 Service A 调用本机的资源管理器的时候, 会将该资源管理器纳入到本事务之中。资源管理器 (RM) 向本机的 DTC 进行事务登记, 从此 DTC 和资源管理器之间建立起了上下级关系。

当 Service A 在调用 Service B 的时候, 会将当前事务的一些信息, 比如分布式事务的 ID 及关于本机 DTC 相关信息, 封装在消息中向对方传递。当 Service B 接收到服务调用请求消息时, 会将事务相关的信息提取出来在本地重建事务, 并将其作为当前的环境事务, 该事务和原事务具有相同的 ID。

与此同时, Service B 根据得到的关于 Machine A 的 DTC 相关信息, 让本机的 DTC 对 Machine A 的 DTC 进行事务登记, 进而使两台机器的 DTC 确立了上下级关系。和 Service A 访问本机的资源管理器一样, Machine B 的资源管理器被 Service B 调用并被纳入当前事务的

时候, 会向本机 DTC 进行事务登记。

当上面所述的事务登记流程结束后, 参与整个分布式事务的 DTC 和资源管理器形成了类似于图 3-6 所示的树形层次结构。由于该结构的构建主要是为了后面对整个事务的提交服务, 所以将其称为事务提交树 (Transaction Commit Tree)。事务提交树的根为事务初始化服务所在机器的 DTC, 在整个事务提交过程中, 它是总的协调者, 又称为全局提交协调器 (GCC, Global Commit Coordinator)。资源管理器充当事务提交树的叶子节点, 它们的父节点为本机的 DTC。分布于不同机器的 DTC 按照事务传播的路径形成上下级关系。

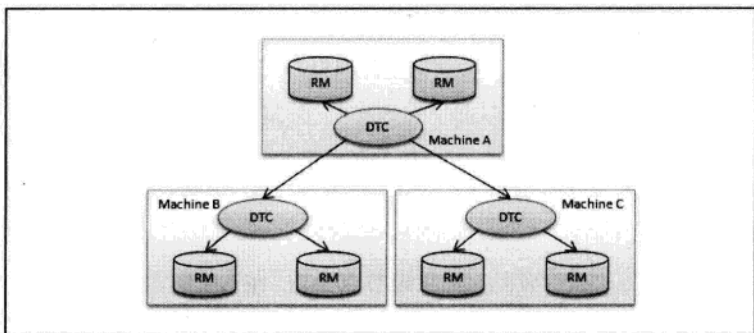


图 3-6 事务提交树

事务提交树的构建使得对分布式事务的提交成为可能。分布式事务的提交采用两阶段协议实现, 接下来详细介绍基于两阶段提交协议的事务提交机制。

2. 两阶段提交协议

不同于基于单一资源管理器的本地事务, 在一个分布式环境中实现一个涉及多个资源管理器的分布式事务要复杂得多。当事务初始化服务完成所有相关的操作后, 决定提交该事务。对于分布式事务的提交, 最终的结果有如下两个。

- 如果所有的操作能够顺利完成, 需要持久化的数据被相应的资源管理器写入目标资源;
- 如果任何一个环节失败, 所有持久化资源管理器将数据恢复到原来的状态。

分布式事务的整个提交过程, 采用两阶段提交协议完成。顾名思义, “两阶段提交”意味着整个事务提交阶段分为两个阶段, 下面就来详细介绍在这两个阶段中, 分别都在做些什么。

(1) 第一阶段 (Phase I): 准备 (Prepare) 阶段

在第一个阶段, 作为根节点的 DTC 沿着事务提交树的路径, 向所有事务的参与者发起请求, 要求它们对本地事务的结果进行投票。被请求的参与者将本地事务结果返回给自己的上级。对于资源管理器, 自己的上级就是本机的 DTC。如果自己本身就是 DTC, 那么自己的上级就是位于事务提交树父节点的 DTC。根据具体事务操作执行的情况, 参与者投票的类型包括如下三种。

- 就绪 (Prepared): 参与者同意对事务的提交, 并承诺在接收到真正的提交请求后完成本地的提交任务。
- 只读 (Read-only): 参与者同意对事务的提交, 但是不希望接收真正的事务提交请求。
- 中止 (Aborted): 参与者要求中止事务。

对于一个非根节点 DTC 来说, 当它从父节点接收到标准的“准备”请求后, 会立即将该请求沿着事务提交树发送给自己的下级 (本机的资源管理器和下级 DTC), 然后根据从下级接收的所有投票结果, 计算出自己投票的结果。具体的算法是: 如果所有的投票结果都是“就绪”和“只读”, 对应的结果是“提交”, 反之如果任何一个投票结果是“中止”, 则最终的结果就是“中止”。换句话说, 任何一个事务参与者具有一票否决权。最后 DTC 将计算出来的投票结果反馈给自己的上级。

当根节点 DTC 接收到隶属于自己的所有资源管理器和下级 DTC 的投票结果后, 采用与上面一样的算法决定整个分布式事务的最终结果。当根节点 DTC 决定了事务最终的结果后, 整个提交过程进入第二阶段。

此外, 我们可以设置事务的超时时限, 如果根节点 DTC 在该时限内没有接收到所有参与者的投票请求, 会对整个分布式事务做出回滚的决定。

(2) 第二阶段 (Phase II): 提交 (Commit) 或回滚 (Rollback)

作为事务提交树根节点的 DTC 根据最终的投票结果, 对整个事务进行最终的提交或者中止操作。同样是沿着事务提交树的路径, 提交或者中止请求被广播出去。相应的资源管理器根据从本机 DTC 获得的请求, 实施最终的提交或者恢复操作。当事务参与者完成了各自的任务后, 类似于第一个阶段的投票, 会将执行的结果沿着事务提交树逆向回馈给作为根节点的 DTC。

根节点 DTC 只有接收到所有事务参与者基于各自事务处理的回复, 才能确保整个事务被成功提交或者回滚。那么如果事务的参与者完成了第一阶段的投票后网络断开, 将会如何呢? 这就涉及对未决 (In-Doubt) 事务的处理。

(3) 未决 (In-Doubt) 事务的处理

对于某个分布式事务的参与者 (DTC 或者资源管理器) 来说, 在第一阶段向上级 (事务提交树的父节点) 投票表明提交就绪 (Prepared) 之后, 直到它接收到根节点 DTC 最终的提交或者回滚的请求, 它并不知道本地事务的结果。在这期间, 如果出现死机并重启, 本地的事务处于一种“未决 (In-Doubt)”状态。未决事务仅仅出现在非根节点 DTC 所在的机器。

分布式事务采用如下的机制处理未决事务: 当重启后, 对于本机的所有未决事务, DTC 会向上级 DTC 发送查询请求, 获取每一个事务最终的结果 (提交还是中止)。如果上级也不能决定事务的结果, 那么请求会沿着事务提交树不断向上 (沿着根的方向) 发送, 直到得到一个明确的答复 (不管怎样, 位于根节点的全局提交协调器总是清楚事务的结果的)。

而当下级 DTC 向自己发送相同的查询请求的时候, 该 DTC 会将获取到的结果回复给它

们。如果未决事务存在的时间太长，系统管理者可以强制提交或者中止该事务。

(4) 单阶段提交 (SPC, Single-Phase Commit) 优化

对于事务最终结果（提交或者中止）的决策者来说，如果它具有了不止一个下级，两阶段提交协议是唯一选择。但是如果仅具有一个唯一的下级，这种投票机制就没有必要了。在这种情况下，根节点 DTC 采用一种优化的协议来完成整个事务的提交，我们称之为单阶段提交 (SPC, Single-Phase Commit)。

顾名思义，SPC 表示将事务提交缩短为一个阶段。整个流程很简单，如果根节点 DTC 仅有一个登记的事务参与者（本机资源管理器或者下级 DTC），而不管这个下级自身具有几个下级，它不会像 2PC 一样先向下级发送“准备”请求，而是直接发送提交请求。我们将这个请求称为单阶段提交 (SPC) 请求。接收到 SPC 请求的参与者，如果是资源管理器，则直接提交本地事务，并将最终结果（成功提交或者失败中止）反馈给这个根节点 DTC。

如果 SPC 请求的接收者是 DTC，那么会根据隶属于自己的下级的数量选择相应的提交策略。具体做法和根节点 DTC 提交策略的选择方式一样，即如果自己具有唯一一个下级，则采用 SPC；反之采用 2PC。

也就是说，不仅仅是根节点 DTC 可以选择 SPC 提交事务，任何具有单一下级的 DTC 均可以采用 SPC。但是非根节点 DTC 只有在接收到 SPC 请求的情况下，才能选择通过 SPC 提交事务。如图 3-7 所示，给出了两棵事务提交树（图中忽略掉资源管理器，每个节点代表 DTC）。对于左边的树，因为根节点 A 和下级 B 均只有一个唯一的下级，所以 A 和 B 均采用 SPC，C 具有两个下级，则采用 2PC。而对于右边的树，因为根节点本身具有两个下级，决定了所有的节点均采用 2PC，即使对于只有一个下级的 B 和 C 也是如此。

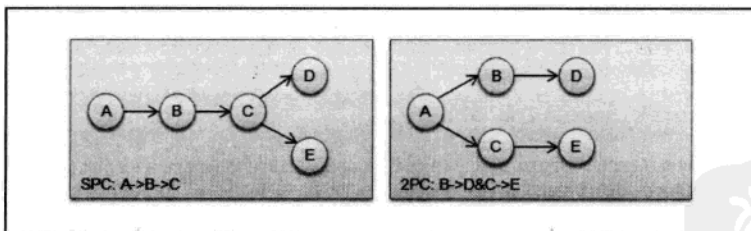


图 3-7 DTC 对 SPC 和 2PC 的选择

3.2.3 System.Transactions 事务

WCF 以基于 DTC 的 Windows 事务基础架构作为依托，提供了一个完善的分布式事务解决方案。WCF 本身是一个纯粹的基于 .NET 的分布式通信平台，其事务体系完全是基于 .NET 的事务，即本章我们要着重介绍的 System.Transactions 事务。

在 .NET 1.x 中，我们基本上是通过 ADO.NET 实现对不同数据库访问的事务。.NET 2.0

带来了全新的事务编程模型。由于所有事务组件或者类型均定义在 System.Transactions 程序集中的 System.Transactions 命名空间下, 我们直接称基于此的事务为 System.Transactions 事务。System.Transactions 事务编程模型使我们可以显式(通过 System.Transactions.Transaction)或者隐式(基于 System.Transactions.TransactionScope)地进行事务编程。下面先来看看这种全新的事务如何表示。

1. System.Transactions.Transaction

在 System.Transactions 事务体系下, 事务本身是通过具有如下定义的 Transaction 类型表示的。Transaction 定义了众多的属性和方法, 将在后续部分进行系统的介绍。

```
[Serializable]
public class Transaction : IDisposable, ISerializable
{
    public event TransactionCompletedEventHandler TransactionCompleted;

    public Transaction Clone();
    public DependentTransaction DependentClone(DependentCloneOption cloneOption);

    public Enlistment EnlistDurable(Guid resourceManagerIdentifier,
        IEnlistmentNotification enlistmentNotification, EnlistmentOptions enlistmentOptions);
    public Enlistment EnlistDurable(Guid resourceManagerIdentifier,
        ISinglePhaseNotification singlePhaseNotification, EnlistmentOptions enlistmentOptions);
    public bool EnlistPromotableSinglePhase(IPromotableSinglePhaseNotification promotableSinglePhaseNotification);
    public Enlistment EnlistVolatile(IEnlistmentNotification enlistmentNotification, EnlistmentOptions enlistmentOptions);
    public Enlistment EnlistVolatile(ISinglePhaseNotification singlePhaseNotification, EnlistmentOptions enlistmentOptions);

    public void Rollback();
    public void Rollback(Exception e);

    void ISerializable.GetObjectData(SerializationInfo serializationInfo,
        StreamingContext context);

    public static Transaction Current { get; set; }
    public IsolationLevel IsolationLevel { get; }
    public TransactionInformation TransactionInformation { get; }
}
```

(1) Transaction 是可序列化的

从上面的定义可以看到, Transaction 在类型上应用了 SerializableAttribute 特性, 并且实现了 ISerializable 接口, 意味着一个 Transaction 对象是可以被序列化的。Transaction 的这一特性在 WCF 整个分布式事务中意义重大。因为要让事务能够控制整个服务操作, 必须实现事务的传播, 而传播的前提就是事务可被序列化。

(2) 如何登记事务参与者

在 Transaction 中, 定义了 5 个 EnlistXxx 方法用于将资源管理器登记到当前事务中。其中 EnlistDurable 和 EnlistVolatile 分别实现了对持久化资源管理器和易失资源管理器的事务登记, 而 EnlistPromotableSinglePhase 则针对的是可被提升的资源管理器 (比如基于 SQL Server 2005 和 SQL Server 2008 的)。

事务登记的目的是建立事务提交树, 使得处于根节点的事务管理器在事务提交的时候能够沿着这棵树将相应的通知发送给所有的事务参与者。这种自上而下的通知机制依赖于具体采用事务提交协议, 或者说某个资源要求参与到当前事务之中, 必须满足基于协议需要的接收和处理相应通知的能力。System.Transactions 将不同事务提交协议对参与者的要求定义在相应的接口中。其中 System.Transactions.IEnlistmentNotification 和 System.Transactions.ISinglePhaseNotification 分别基于 2PC 和 SPC。

如果要为相应的资源开发能够参与到 System.Transactions 事务的资源管理器, 需要实现 IEnlistmentNotification 接口, 对基本的 2PC 协议提供支持。当满足 SPC 要求的时候, 如果希望采用 SPC 优化协议, 则需要实现 ISinglePhaseNotification 接口。如果希望像 SQL Server 2005 或者 SQL Server 2008 那样支持事务提升机制, 则需要实现 IPromotableSinglePhaseNotification 接口。

(3) 环境事务 (Ambient Transaction)

Transaction 定义了一个类型为 Transaction 的可读写 Current 静态属性, 表示当前的事务。作为当前事务的 Transaction 存储于当前线程的 TLS (Thread Local Storage) 中 (实际上是定义在一个应用了 System.ThreadStaticAttribute 特性的静态字段上), 所以仅对当前线程有效。如果进行异步调用, 当前事务并不能自动实现跨线程传播。将异步操作纳入到当前事务, 需要使用到依赖事务。

这种基于当前线程的事务又称环境事务, 很多资源管理器都具有对环境事务的感知能力。也就是说, 如果我们通过 Current 属性设置了环境事务, 当对某个具有环境事务感知能力的资源管理器进行访问的时候, 相应的资源管理器会自动登记到当前事务中来。我们将具有这种感知能力的资源管理器称为 System.Transactions 资源管理器。

(4) 事务标识

Transaction 具有一个只读的类型为 System.Transactions.TransactionInformation 的 TransactionInformation 属性, 表示事务的一些基本的信息。如下面的代码所示, TransactionInformation 类定义了 4 个只读的属性。

```
public class TransactionInformation
{
    public DateTime CreationTime { get; }
    public TransactionStatus Status { get; }

    public string LocalIdentifier { get; }
    public Guid DistributedIdentifier { get; }
}
```

TransactionInformation 的 CreationTime 和 Status 表示创建事务的时间和事务的当前状态。事务具有活动 (Active)、提交 (Committed)、中止 (Aborted) 和未决 (In-Doubt) 4 种状态, 通过具有如下定义的 System.Transactions.TransactionStatus 枚举表示。

```
public enum TransactionStatus
{
    Active,
    Committed,
    Aborted,
    InDoubt
}
```

事务具有两个标识符, 一个是本地标识, 另一个是分布式标识, 分别通过 TransactionInformation 的只读属性 LocalIdentifier 和 DistributedIdentifier 表示。本地标识由标识为本地应用程序域分配的轻量级事务管理器 (LTM) 的 GUID 和一个递增的整数 (表示当前 LMT 管理的事务序号) 组成。在下面的代码中, 分别打印出三个新创建的可提交事务 (CommitableTransaction, 为 Transaction 的子类, 后面会详细介绍) 的本地标识。

```
Console.WriteLine(new
CommittableTransaction().TransactionInformation.LocalIdentifier);
Console.WriteLine(new
CommittableTransaction().TransactionInformation.LocalIdentifier);
Console.WriteLine(new
CommittableTransaction().TransactionInformation.LocalIdentifier);
```

输出结果:

```
AC48F192-4410-45fe-AFDC-8A890A3F5634: 1
AC48F192-4410-45fe-AFDC-8A890A3F5634: 2
AC48F192-4410-45fe-AFDC-8A890A3F5634: 3
```

一旦本地事务提升到基于 DTC 的分布式事务, 系统会为之生成一个 GUID 作为其唯一标识。当事务跨边界执行的时候, 分布式事务标识会随着事务一并被传播。所以在不同的执行上下文中, 你会得到相同的 GUID。分布式事务标识通过 TransactionInformation 的只读属性 DistributedIdentifier 表示, 该标识经常在审核 (Audit) 中被使用。

除了事务的标识, Transaction 通过只读属性 IsolationLevel 表示隔离级别。通过 Rollback 和 Clone 方法对事务进行回滚和克隆一个新的 Transaction 对象。通过上面对 Transaction 的介绍, 细心的读者可能会发现如下两个问题。

- Transaction 并没有提供公有的构造函数, 意味着我们不能直接通过 new 操作符创建 Transaction 对象。
- Transaction 只有两个重载的 Rollback 方法, 并没有 Commit 方法, 意味着我们直接通过它进行事务提交。

在一个分布式事务中, 事务初始化和提交只能由相同的参与者担当。也就是说只有最初

开始的事务的应用才能最终提交该事务。我们将这种能被初始化和提交的事务称为可提交事务（**Committable Transaction**）。随着分布式事务参与者逐个登记到事务之中，它们本地的事务实际上依赖于这个最初开始的事务，所以我们称这种事务为依赖事务（**Dependent Transaction**）。

2. 可提交事务

只有可提交事务才能被直接初始化，对可提交事务的提交驱动着对整个分布式事务的提交。可提交事务通过具有如下定义的 **System.Transactions.CommittableTransaction** 类型表示。

```
[Serializable]
public sealed class CommittableTransaction : Transaction, IAsyncResult
{
    public CommittableTransaction();
    public CommittableTransaction(TimeSpan timeout);
    public CommittableTransaction(TransactionOptions options);

    public void Commit();
    public IAsyncResult BeginCommit(AsyncCallback asyncCallback, object
    asyncState);
    public void EndCommit(IAsyncResult asyncResult);

    object IAsyncResult.AsyncState { get; }
    WaitHandle IAsyncResult.AsyncWaitHandle { get; }
    bool IAsyncResult.CompletedSynchronously { get; }
    bool IAsyncResult.IsCompleted { get; }
}
```

（1）可提交事务的超时时限和隔离级别

CommittableTransaction 直接继承自 **Transaction**，提供了三个公有的构造函数。通过 **TimeSpan** 类型的 **timeout** 参数指定事务的超时时限。该超时时限从事务被初始化那一刻开始算起，一旦超过了该时限，事务会被中止。通过 **System.Transactions.TransactionOptions** 类型的 **options** 可以同时指定事务的超时时限和隔离级别。下面的代码片段给出了 **TransactionOptions** 的基本定义，从中可见它是一个结构体，两个属性 **Timeout** 和 **IsolationLevel** 分别代表事务的超时时限和隔离级别。

```
[StructLayout(LayoutKind.Sequential)]
public struct TransactionOptions
{
    //其他成员
    public TimeSpan Timeout { get; set; }
    public IsolationLevel IsolationLevel { get; set; }
}
```

如果调用默认无参的构造函数来创建 **CommittableTransaction** 对象，意味着采用一个默认的超时时限。这个默认的时间是 1 分钟，不过它可以通过配置的方式进行指定。事务超时时限相关的参数定义在 **<system.transactions>** 配置节中，下面的 XML 体现的是默认的配置。从该段配置可以看到，不但可以通过 **<defaultSettings>** 设置事务默认的超时时限，还可以通过

<machineSettings>设置最高可被允许的事务超时时限，默认为 10 分钟。在对这两项进行配置的时候，前者的时间必须小于后者，否则将用后者作为事务默认的超时时限。

```
<configuration>
  <system.transactions>
    <defaultSettings timeout="00:01:00"/>
    <machineSettings maxTimeout="00:10:00"/>
  </system.transactions>
</configuration>
```

作为事务 ACID 四大属性之一的隔离性 (Isolation)，确保事务操作的中间状态的可见性仅限于事务内部。隔离机制通过对访问的数据进行加锁，防止数据被事务的外部程序操作，从而确保了数据的一致性。但是隔离机制在另一方面又约束了对数据的并发操作，降低了数据操作的整体性能。为了权衡这两个互相矛盾的两个方面，我们可以根据具体的情况选择相应的隔离级别。

在 System.Transactions 事务体系中，为事务提供了 7 种不同的隔离级别。这 7 种隔离级别分别通过具有如下定义的 System.Transactions.IsolationLevel 枚举的 7 个枚举项表示。

```
public enum IsolationLevel
{
    Serializable,
    RepeatableRead,
    ReadCommitted,
    ReadUncommitted,
    Snapshot,
    Chaos,
    Unspecified
}
```

7 个隔离级别之中，Serializable 具有最高隔离级别，代表的是一种完全基于序列化（同步）的数据存取方式，这也是 System.Transactions 事务默认采用的隔离级别。按照隔离级别自高向低，7 个不同的隔离级别具有如下含义。

- **Serializable**: 可以在事务期间读取可变数据，但是不可以修改，也不可以添加任何新数据。
- **RepeatableRead**: 可以在事务期间读取可变数据，但是不可以修改。可以在事务期间添加新数据。
- **ReadCommitted**: 不可以在事务期间读取可变数据，但是可以修改。
- **ReadUncommitted**: 可以在事务期间读取和修改可变数据。
- **Snapshot**: 可以读取可变数据。在事务修改数据之前，它验证在它最初读取数据之后另一个事务是否更改过这些数据。如果数据已被更新，则会引发错误。这样使事务可获取先前提交的数据值。
- **Chaos**: 无法覆盖隔离级别更高的事务中挂起的更改。
- **Unspecified**: 正在使用与指定隔离级别不同的隔离级别，但是无法确定该级别。如果设置了此值，则会引发异常。

(2) 事务的提交

CommittableTransaction 提供了同步(通过 Commit 方法)和异步(通过 BeginCommitToEnd Commit 方法组合)的事务提交方式。CommittableTransaction 还是实现了 System.IAsyncResult 这样一个接口,如果采用异步的方式调用 BeginCommit 方法提交事务,方法返回的 IAsyncResult 对象的各属性值会反映在 CommittableTransaction 的同名属性上面。

前面我们提到了环境事务及 System.Transactions 资源管理器对环境事务的自动感知能力。当创建了 CommittableTransaction 对象的时候,被创建的事务并不会自动作为环境事务,需要手工将其指定到 Transaction 的静态 Current 属性中。接下来将通过一个简单的例子演示如何通过 CommittableTransaction 实现一个分布式事务。

(3) 实例演示:通过 CommittableTransaction 实现分布式事务

在这个实例演示中,我们沿用介绍事务显式控制时使用的银行转账的场景,下面两个方法分别实现提取、存储。

```
static void Withdraw(string accountId, double amount)
{
    //省略实现
}
static void Deposit(string accountId, double amount)
{
    //省略实现
}
```

现在要借助于 CommittableTransaction 将整个转账操作纳入同一个事务中,可以采用如下的方式编写程序。

```
private static void Transfer(string accountFrom, string accountTo, double
amount)
{
    Transaction originalTransaction = Transaction.Current;
    CommittableTransaction transaction = new CommittableTransaction();
    try
    {
        Transaction.Current = transaction;
        Withdraw(accountFrom, amount);
        Deposit(accountTo, amount);
        transaction.Commit();
    }
    catch (Exception ex)
    {
        transaction.Rollback(ex);
        throw;
    }
    finally
    {
        Transaction.Current = originalTransaction;
        transaction.Dispose();
    }
}
```

3. 依赖事务

通过前面给出的 Transaction 的定义,细心的读者应该看到了一个叫做 DependentClone 的方法。该方法用于创建基于现有 Transaction 对象的“依赖事务 (DependentTransaction)”。与可提交事务是一个独立的事务对象不同,依赖事务依附于现有的某个事务 (可能是可提交事务,也可能是依赖事务)。

依赖事务通过具有如下定义的 System.Transactions.DependentTransaction 类型表示。和 CommittableTransaction 一样, DependentTransaction 也是 Transaction 的子类。因为 DependentTransaction 依赖于现有的 Transaction 对象而存在,相当于被依赖事务的子事务,所以无法执行对事务的提交,也自然不会定义 Commit 方法。但是 DependentTransaction 具有一个唯一的 Complete 方法。调用这个方法意味着向被依赖事务发送通知,表明所有与依赖事务相关的操作已经完成。

```
[Serializable]
public sealed class DependentTransaction : Transaction
{
    public void Complete();
}
```

(1) 通过 DependentTransaction 将异步操作纳入现有事务

通过 Transaction 的静态属性 Current 表示的环境事务是基于当前线程的。这就意味着即使环境事务存在,通过异步调用的操作也不可能自动加入到当前事务之中。在这种情况下,我们需要做的就是手工将当前事务传递到另一个线程中,作为它的环境事务。通过依赖事务很容易实现这一点。

DependentTransaction 通过 Transaction 的 DependentClone 方法创建。该方法具有一个 System.Transactions.DependentCloneOption 枚举类型的参数,表示被依赖的事务在尚未接收到依赖事务通知 (调用 Complete 或者 Rollback 方法) 的情况下,提交 (可提交事务) 或者完成 (依赖事务) 所采取的行为。

DependentCloneOption 提供了两个选项,其中 BlockCommitUntilComplete 表示被依赖事务会一直等待接收到依赖事务的通知或者超过事务设定的超时时限,而 RollbackIfNotComplete 则会直接将依赖的事务回滚,并抛出 System.Transactions.TransactionAbortedException 异常。

```
[Serializable]
public class Transaction : IDisposable, ISerializable
{
    //其他成员
    public DependentTransaction DependentClone(DependentCloneOption cloneOption);
}
public enum DependentCloneOption
{
    BlockCommitUntilComplete,
    RollbackIfNotComplete
}
```

下面的代码演示了如何通过依赖事务采用异步的方式进行银行转账操作。借助于 `ThreadPool` 将主线程环境事务的依赖事务传递给异步操作代理。开始异步操作的时候将此依赖事务作为当前的环境事务，那么之后的操作将自动在当前事务下进行。

```
private static void Transfer(string accountFrom, string accountTo, double
amount)
{
    Transaction originalTransaction = Transaction.Current;
    CommittableTransaction transaction = new CommittableTransaction();
    try
    {
        Transaction.Current = transaction;
        ThreadPool.QueueUserWorkItem(state =>
        {
            Transaction.Current = state as DependentTransaction;
            try
            {
                Withdraw(accountFrom, amount);
                Deposit(accountTo, amount);
                (state as DependentTransaction).Complete();
            }
            catch (Exception ex)
            {
                Transaction.Current.Rollback(ex);
            }
            finally
            {
                (state as IDisposable).Dispose();
                Transaction.Current = null;
            }
        }, Transaction.Current.DependentClone(DependentCloneOption.BlockCommit
UntilComplete));
        //其他操作
        transaction.Commit();
    }
    catch (TransactionAbortedException ex)
    {
        transaction.Rollback(ex);
        Console.WriteLine("转账失败, 错误信息: {0}", ex.InnerException.Message);
    }
    catch (Exception ex)
    {
        transaction.Rollback(ex);
        throw;
    }
    finally
    {
        Transaction.Current = originalTransaction;
        transaction.Dispose();
    }
}
```

因为在调用 `DependentClone` 方法创建依赖事务时指定的参数为 `DependentCloneOption.BlockCommitUntilComplete`，所以主线程在调用 `Commit` 方法提交事务的时候，如果依赖事务尚未结束（调用 `Complete` 或者 `Rollback` 方法），会一直等待超出事务设置的超时时限。

(2) 通过 `DependentTransaction` 实现事务型方法

这里所说的事务型方法是指方法的执行总是在事务中执行。具体来讲,有两种不同的事务应用场景:

- 如果当前不存在环境事务,那么方法的执行将在一个独立的事务中执行;
- 如果存在环境事务,则方法执行会自动加入到环境事务之中。

比如,存储(`Deposit`)和提取(`Withdraw`)就是典型的事务型操作。对于单纯的存取款,应该创建一个新的事务来控制存储和提取操作的执行,以确保单一账户款项的数据一致性。在转账的场景中,应在转账开始之前就创建一个新的事务,让提取和存储的操作自动加入到这个事务之中。

下面就结合可提交事务和依赖事务将 `Deposit` 和 `Withdraw` 两个方法定义成事务型方法。在这里把事务控制部分定义在如下一个 `InvokeInTransaction` 静态方法中。

```
static void InvokeInTransaction(Action action)
{
    Transaction originalTransaction = Transaction.Current;
    CommittableTransaction committableTransaction = null;
    DependentTransaction dependentTransaction = null;
    if (null == Transaction.Current)
    {
        committableTransaction = new CommittableTransaction();
        Transaction.Current = committableTransaction;
    }
    else
    {
        dependentTransaction = Transaction.Current.DependentClone(
            DependentCloneOption.RollbackIfNotComplete);
        Transaction.Current = dependentTransaction;
    }

    try
    {
        {
            action();
            if (null != committableTransaction)
            {
                committableTransaction.Commit();
            }

            if (null != dependentTransaction)
            {
                dependentTransaction.Complete();
            }
        }
    }
    catch (Exception ex)
    {
        Transaction.Current.Rollback(ex);
        throw;
    }
    finally
    {

```

```

        Transaction transaction = Transaction.Current;
        Transaction.Current = originalTransaction;
        transaction.Dispose();
    }
}

```

InvokeInTransaction 方法的参数是一个 **Action** 类型的委托 (**Delegate**)，表示具体的业务操作。在开始的时候记录下当前的环境事务，当整个操作结束之后应该将环境事务恢复成该值。如果存在环境事务，则创建环境事务的依赖事务，反之直接创建可提交事务。最后将新创建的依赖事务或者可提交事务作为当前的环境事务。

目标操作的执行放在 **try/catch** 中。在目标操作顺利执行后，调用依赖事务的 **Complete** 方法或者可提交事务的 **Commit** 方法。如果抛出异常，则调用环境事务的 **Rollback** 进行回滚。在 **finally** 块中将环境事务恢复到之前的状态，并调用 **Dispose** 方法对创建的事务进行回收。

借助于 **InvokeInTransaction** 这个辅助方法，我们以事务型方法的形式定义了如下的 **Withdraw** 和 **Deposit** 两个方法分别实现提取和存储的操作。

```

static void Withdraw(string accountId, double amount)
{
    Dictionary<string, object> parameters = new Dictionary<string, object>();
    parameters.Add("id", accountId);
    parameters.Add("amount", amount);
    InvokeInTransaction(() => DbAccessUtil.ExecuteNonQuery("P_WITHDRAW",
        parameters));
}
static void Deposit(string accountId, double amount)
{
    Dictionary<string, object> parameters = new Dictionary<string, object>();
    parameters.Add("id", accountId);
    parameters.Add("amount", amount);
    InvokeInTransaction(() => DbAccessUtil.ExecuteNonQuery("P_DEPOSIT",
        parameters));
}

```

如果单独调用 **Withdraw** 或者 **Deposit** 方法，可以确保所有的操作在一个事务中执行。如果当前具有环境事务，方法的执行会自动纳入环境事务中来。在下面的代码中，我们以不一样的方式提供了转账的实现。

```

private static void Transfer(string accountFrom, string accountTo, double amount)
{
    Transaction originalTransaction = Transaction.Current;
    CommittableTransaction transaction = new CommittableTransaction();
    try
    {
        Transaction.Current = transaction;
        Withdraw(accountFrom, amount);
        Deposit(accountTo, amount);
    }
    catch (Exception ex)
    {

```

```

        transaction.Rollback();
        Console.WriteLine("转账失败, 错误信息为: {0}", ex.Message);
    }
    finally
    {
        Transaction.Current = originalTransaction;
        transaction.Dispose();
    }
}

```

4. TransactionScope

在上一节中, 结合可提交事务和依赖事务, 以及环境事务的机制提供了对事务型操作的实现。实际上如果借助于 `TransactionScope` 相应的代码将会变得非常简单。下面的代码中, 通过 `TransactionScope` 对 `InvokeInTransaction` 进行了改写, 从执行效果来看这和原来的代码完全一致。

```

static void InvokeInTransaction(Action action)
{
    using (TransactionScope transactionScope = new TransactionScope())
    {
        action();
        transactionScope.Complete();
    }
}

```

通过 `InvokeInTransaction` 方法前后代码的对比, 可以明显看到 `TransactionScope` 确实能够使我们的事务控制变得非常简单。实际上在利用 `System.Transactions` 事务进行编程的时候, 一般不会使用到可提交事务, 对于依赖事务也只有在异步调用的时候会使用到。基于 `TransactionScope` 的事务编程方式才是我们推荐的。

正如其名称所表现的一样, `TransactionScope` 为一组事务型操作创建一个执行范围, 而这个范围始于 `TransactionScope` 创建之时, 结束于 `TransactionScope` 被回收 (调用 `Dispose` 方法)。在对 `TransactionScope` 进行深入介绍之前, 照例先来看看它的定义。

```

public sealed class TransactionScope : IDisposable
{
    public TransactionScope();
    public TransactionScope(Transaction transactionToUse);
    public TransactionScope(TransactionScopeOption scopeOption);
    public TransactionScope(Transaction transactionToUse, TimeSpan
        scopeTimeout);
    public TransactionScope(TransactionScopeOption scopeOption, TimeSpan
        scopeTimeout);
    public TransactionScope(TransactionScopeOption scopeOption, Transaction
        Options
        transactionOptions);
    public TransactionScope(Transaction transactionToUse, TimeSpan
        scopeTimeout,
        EnterpriseServicesInteropOption interopOption);
    public TransactionScope(TransactionScopeOption scopeOption,
        TransactionOptions

```

```

        transactionOptions, EnterpriseServicesInteropOption interopOption);

    public void Complete();
    public void Dispose();
}

```

TransactionScope 实现了 IDisposable 接口, 除了 Dispose 方法之外, 仅具有唯一的 Complete 方法。但是 TransactionScope 却有一组丰富的构造函数。下面看看这些构造函数相应的参数如何影响 TransactionScope 对事务控制的行为。

5. TransactionScopeOption

实际上前面编写的 InvokeInTransaction 方法基本上体现了 TransactionScope 的内部实现。也就是说, TransactionScope 也是通过创建可提交事务或者依赖事务, 并将其作为事务范围内的环境事务, 从而将范围的所有操作纳入一个事务之中。

通过在构造函数中指定 System.Transactions.TransactionScopeOption 类型的 scopeOption 参数, 控制 TransactionScope 当环境事务存在的时候应该采取怎样的方式执行事务范围内的操作。我们具有如下三种不同的选择。

- 如果已经存在环境事务, 则使用该环境事务。否则, 在进入范围之前创建新的事务。
- 总是为该范围创建新事务。
- 环境事务上下文在创建范围时被屏蔽。范围中的所有操作都在无环境事务上下文的情况下完成。

TransactionScopeOption 是一个枚举, 三个枚举值 Required、RequiresNew 和 Suppress 依次对应上面的三种行为。

```

public enum TransactionScopeOption
{
    Required,
    RequiresNew,
    Suppress
}

```

对于 Required 选项, 在当前存在环境事务存在的情况下 TransactionScope 会创建环境事务的依赖事务, 否则创建可提交事务。然后将创建的依赖事务或者可提交事务作为事务范围内的环境事务。对于 RequiresNew 选项, TransactionScope 总是会创建可提交事务并将其作为事务范围内的环境事务。如果选择 Suppress 选项, TransactionScope 会将事务范围内的环境事务设为空, 意味着事务范围内的操作并不受事务的控制。

Required 是默认选项, 意味着事务范围内的事务将会作为当前环境事务的一部分。如果不希望某个操作被纳入当前的环境事务, 相应的操作也需要事务的控制以确保所操作数据的一致性。比如, 当业务逻辑失败导致异常抛出, 需要对相应的错误信息进行日志记录。对于日记的操作就可以放入基于 RequiresNew 选项创建 TransactionScope 中。对于一些不重要的操作 (操作的错误可被忽略), 并且不需要通过事务来控制的操作, 比如发送一些不太重要

的通知, 就可以采用 Suppress 选项。

(1) TransactionOptions 和 EnterpriseServicesInteropOption

TransactionOptions 在前面已经提及, 用于控制事务的超时时限和隔离级别。对于超时时限, 也可以选择 TransactionScope 相应的构造函数以 TimeSpan 的形式指定。而对于事务的隔离级别, 有一点值得引起注意, 那就是当选择 TransactionScopeOption.Required 选项时, TransactionScope 指定的隔离级别必须与环境事务 (如果有) 相匹配。

```
using (TransactionScope outerScope = new TransactionScope())
{
    TransactionOptions transactionOptions = new TransactionOptions() {
        IsolationLevel = IsolationLevel.ReadCommitted };
    using (TransactionScope innerScope = new
        TransactionScope(TransactionScopeOption.Required, transactionOptions))
    {
        //事务型操作
        innerScope.Complete();
    }
    //事务型操作
    outerScope.Complete();
}
```

比如上面的例子中, 定义了两个嵌套的 TransactionScope。至于隔离级别, 外部的 TransactionScope 采用默认的隔离级别, 内部采用 ReadCommitted 隔离级别, 当执行这段代码的时候, 会抛出如图 3-8 所示的 ArgumentException 异常。

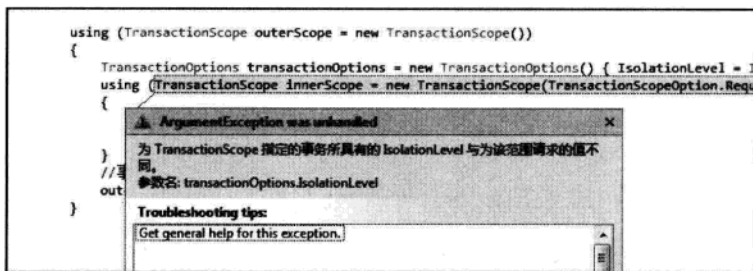


图 3-8 隔离级别不一致导致的异常

在 System.Transactions 事务机制被引入之前, Enterprise Service 主要依赖于基于 COM+ 的分布式事务。TransactionScope 通过 EnterpriseServicesInteropOption 控制 System.Transactions 事务如何与 COM+ 的分布式事务进行互操作。

(2) 事务提交和回滚

对于事务范围中的事务, 无论是事务的提交 (对于可提交事务)、完成 (依赖事务) 和回滚都是在 Dispose 方法中执行的。TransactionScope 中定义了一个私有的布尔类型字段 (complete) 表示事务是否正常结束。当调用 TransactionScope 的 Complete 方法的时候会将此字段设置成 True。当 Dispose 执行的时候, 如果该字段的值为 False, 会调用事务的 Rollback

方法对该事务实施回滚。否则会调用 Commit 方法（对于可提交事务）对事务进行提交或者调用 Complete 方法（依赖事务）通知被依赖的事务本地事务已经正常完成。除了执行事务的提交/完成或者回滚之外，TransactionScope 的 Dispose 方法还负责将环境事务恢复到事务范围开始之前的状态。

在调用 TransactionScope 的 Complete 和 Dispose 之间，环境事务处于不可用的状态，如果此时试图获取环境事务，会抛出异常。

```
using (TransactionScope transactionScope = new TransactionScope())
{
    //其他事务操作
    transactionScope.Complete();
    Transaction ambientTransaction = Transaction.Current;
}
```

比如在上面的代码中，在事务范围内部调用 Complete 方法后，通过 Transaction 的 Current 静态属性获取当前环境事务，会抛出如图 3-9 所示的 InvalidOperationException 异常，提示“当前 TransactionScope 已完成”。

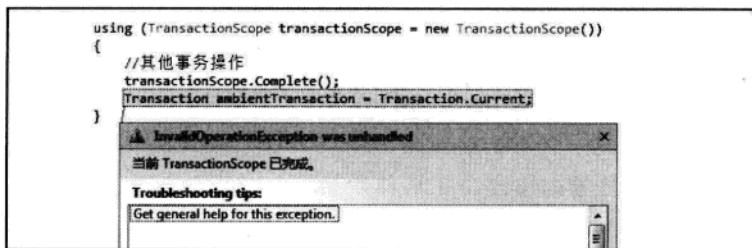


图 3-9 在 TransactionScope 完成之后获取环境事务导致的异常

3.3 事务处理协议: OleTx 和 WS-AT

总的来说，WCF 的分布式事务采用的是两阶段提交（2PC，Two Phase Commit）协议。就具体的协议选择来说，则具有如下两种选择。

- OleTx: OleTx 采用 RPC 作为通信的手段，并采用二进制消息编码，所以 OleTx 是最高效的分布式处理协议。但它是专用于 Windows 平台下的分布式处理协议。当一个分布式事务的所有参与者均基于 Windows 平台时，OleTx 是最好的选择。如果有任何一个非 Windows 的事务资源参与进来，OleTx 就无能为力了。说白了，OleTx 并不具有跨平台的特质。
- WS-AT: WS-AT (WS-Atomic Transaction) 是 WS-* 大家庭中的一员，是由结构化信息标准促进组织 (OASIS, Organization for the Advancement of Structured Information Standards) 制定的基于分布式事务的标准。对于 WCF 来说，WS-AT 弥补了 OleTx 不具备跨平台特质的不足。

限于篇幅, 本书不会对 OleTx 具体的实现进行介绍, 有兴趣的读者可以从 MSDN 网站下载 OleTx 事务管理的规范文档 (<http://msdn.microsoft.com/en-us/library/cc229116%28PROT.10%29.aspx>)。下面讨论 WS-AT 是以怎样的方式来对分布式事务的管理的, 而 WS-AT 建立在 WS-Coordination 规范之上。

3.3.1 WS-Coordination

以 SOAP 和 WSDL 为核心的 Web 服务规范定义了一套完善的协议以实现 Web 服务的互操作, 使我们可以将若干参与者组合起来构成一个分布式的计算单元 (Distributed Computational Unit), 我们将这些计算单元称为分布式活动 (Distributed Activity)。这些活动可能在结构上非常复杂, 各个参与者之间也可能具有复杂的关系。它们也可能由于业务流程的复杂性或者需要与用户交互, 需要很长的执行时间。分布式事务和工作流就是两个典型的分布式活动。

由于组成这个分布式计算单元或者活动的参与者可能隶属于不同的厂商, 因此有效地协调这些参与者必须依赖于一个开放的标准或者规范, WS-Coordination 就是这样一个规范。WS-Coordination 由 OASIS 制定, 到目前为止先后推出了 1.1 和 1.2 两个版本。本节讨论的是 WS-Coordination 1.1 版本。

WS-Coordination 通过一个协调器 (Coordinator) 和若干协调协议 (Coordination Protocol) 定义了一个可扩展的框架去协调一个分布式活动的所有参与者。WS-Coordination 为分布式活动协调定义了一个统一、抽象的协议, 其本身并不用于解决具体的协调问题。基于 WS-Coordination 定义系列的具体标准会被陆续制定出来以解决具体的分布式活动的协调问题。WS-AT 和 WS-BA (WS-BusinessActivity) 就是两个基于 WS-Coordination 的协调协议。接下来就按照逐层深入的方式介绍基于 WS-Coordination 的分布式活动协调机制。

1. 基于协调器 (Coordinator) 的协调模型

WS-Coordination 的协调模型以协调器为中心, 而协调器提供一系列的协调服务。分布式活动的参与者调用本地协调器相应的服务, 各个分布的协调器相互通信从而构建了一个统一的协调上下文环境, 并将所有的参与者纳入其中 (如图 3-10 所示)。

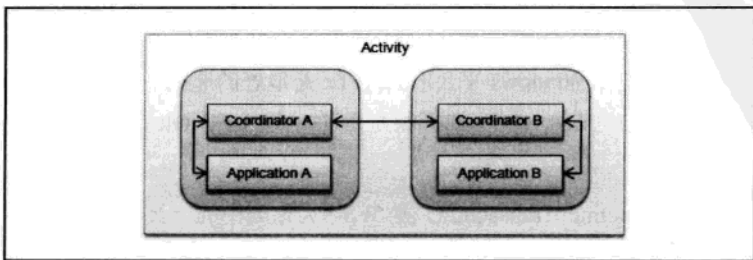


图 3-10 基于协调器的协调模型

一个协调器可以看成是一个协调服务的容器，包含在该容器中的所有服务按照其作用的不同分成如下三种类型，它们在协调器中的构成如图 3-11 所示。

- 激活服务（Activation Service）：当一个分布式活动开始的时候，初始化该活动的参与者调用激活服务的 `CreateCoordinationContext` 操作创建协调上下文（`Coordination Context`）。
- 注册服务（Registration Service）：参与者通过调用注册服务的 `Register` 操作参与到相应的协调协议之中。
- 协议服务（Protocol Service）：一个协调器具有一系列基于具体协调协议（比如 WS-AT 和 WS-BA 等）的服务。

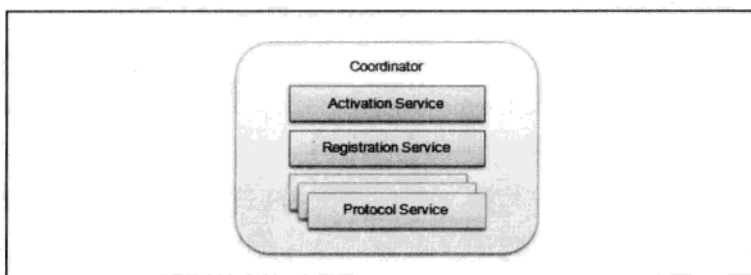


图 3-11 协调器的服务构成

接下来详细介绍基于 WS-Coordination 协议如何实现将分布式活动在不同的参与者之间进行传播，整个过程如图 3-12 所示。分布式活动具有两个参与者：Application 1 和 Application 2，它们具有各自的协调器 Coordinator 1 和 Coordinator 2，AS 和 RS 分别表示协调器的激活服务和注册服务。

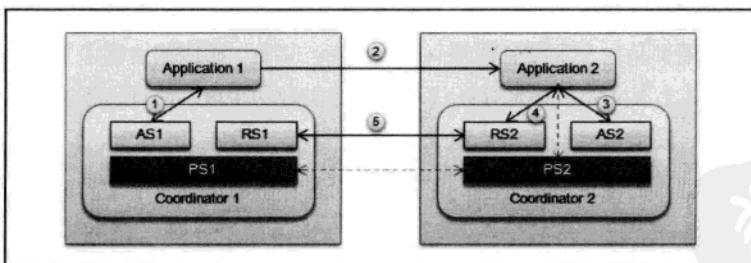


图 3-12 基于 WS-Coordination 协议对一个分布式活动进行协调的全过程

假设协调整个活动的具体的协调服务类型为 Q （你可以假设这就是我们接下来要介绍的 WS-AT），而 PS1 和 PS2 分别表示基于该协调类型的协议服务（假设就是 WS-AT 协议服务）。由于 PS1 和 PS2 最终实现了对整个活动的协调管理，所以整个流程的目的在于如何让 PS1 和 PS2 能够互相通信。说得更加具体一点，按照 WS-Addressing 规范，服务通过终结点引用

(Endpoint Reference) 来定位, 整个流程的目的在于交换 PS1 和 PS2 的终结点引用, 使之能够彼此调用。

步骤 1: 为了实现对基于协调类型 Q 的服务 PS1 的注册, 初始化活动的 Application 1 调用本地的协调器 Coordinator 1 的激活服务 AS1 创建协调上下文。我们称该上下文为 Context 1, 它包含了活动标识、协调器 Coordinator 1 的注册服务 RS1 的终结点引用和协调类型 (Q)。

步骤 2: Application 1 将创建的协调上下文 Context 1 发送到另一个活动参与者 Application 2。

步骤 3: Application 2 以此协调上下文 Context 1 作为输入, 利用本地协调器 Coordinator 2 创建新的协调上下文 Context 2。Context 1 和 Context 2 具有相同的活动标识和协调类型, 但是注册服务终结点指向本地协调器的注册服务 RS2。

步骤 4: Application 2 根据接收到的协调上下文中的协调类型信息 Q 确定具体的协调协议, 并调用注册服务 RS2 将 PS2 注册到 Coordinator 2 上。注册的结果是 Application 2 和 PS2 进行了终结点引用交换, 从而建立起两者之间的逻辑关系。

步骤 5: RS2 根据接收到的协调上下文 Context 1 得到 RS1 的终结点引用, 将 PS2 的终结点引用作为输入调用 RS1 注册到 Coordinator 1 上, RS1 将 PS1 的终结点引用返回。到此为止, PS1 和 PS2 实现了相互之间的终结点引用的交换。

2. 从消息交换看激活服务和注册服务

从上面的步骤不难看出, 整个处理流程主要涉及对协调器两个服务(激活服务和注册服务)的调用。接下来从消息交换的角度对这两个服务的调用进行进一步的介绍。在这之前先来介绍一下上面频繁提到的协调上下文。

(1) 协调上下文

当一个分布式活动需要从一个参与者传播到另一个参与者时, 相应的协调上下文需要进行传播。换句话说, 协调上下文的传播是分布式活动传播实现的手段。协调上下文包括活动标识、协调的具体类型、注册服务的终结点地址及其他一些扩展信息。

协调上下文通过协调器的激活服务创建, 接下来就来看看创建协调上下文的激活服务操作具有怎样的消息结构。

(2) 激活服务

激活服务的目的就是创建协调上下文, 所以它只有一个唯一的 CreateCoordinationContext 操作。活动初始化参与者向激活服务发送 CreateCoordinationContext 请求, 并指定具体的协调类型及其他相关信息。下面的 XML 片段表示了一个 CreateCoordinationContext 请求消息的结构。

```
<CreateCoordinationContext ...>
  <Expires> ... </Expires>?
```

```

<CurrentContext> ... </CurrentContext>?
<CoordinationType> ... </CoordinationType>
...
</CreateCoordinationContext>

```

其中只有<CoordinationType/>节点是必需的, 表示代表具体协调类型的 URI (比如 WS-AT 通过 <http://docs.oasis-open.org/ws-tx/ws-sat/2006/06> 表示)。<Expires/>结点的值是一个以正数表示的过期时限, 单位是毫秒, 自该上下文被接收的那一刻算起。<CurrentContext>结点表示的是当前的协调上下文, 如果为空, 则激活服务会创建一个全新的上下文, 否则创建一个与之关联的上下文 (具有相同的活动标识)。通过指定当前上下文, 让创建的上下文和当前上下文来建立起一种上下级的依赖关系。

当激活服务接收到 CreateCoordinationContext 请求后, 根据请求的内容创建相应的协调上下文, 并将其置于 CreateCoordinationContextResponse 消息中返回。CreateCoordinationContextResponse 消息的主体结构如下面的 XML 片段所示。其中 CoordinationContext 就是创建出来的协调上下文。

```

<CreateCoordinationContextResponse>
<CoordinationContext> ... </CoordinationContext>
...
</CreateCoordinationContextResponse>

```

(3) 注册服务

当分布式活动的参与者从本地协调器获取了协调上下文后, 就可以调用注册服务将具体的协调服务注册到该上下文对应的活动中。下级协调器 (比如图 3-12 中的 Coordinator 2) 通过调用上级协调器的注册服务进行相类似的注册。

注册服务具有一个唯一的 Register 操作。活动参与者 (或者下级协调器) 向本地协调器 (或者上级协调器) 发送相应的 Register 请求, 并指定具体协调协议的标识和参与方协调服务的终结点引用。Register 请求消息主体部分结构如下。

```

<Register ...>
  <ProtocolIdentifier> ... </ProtocolIdentifier>
  <ParticipantProtocolService> ... </ParticipantProtocolService>
  ...
</Register>

```

下面是 WS-AT 基于 Volatile2PC (WS-AT 定义了两个 2PC 协议: Volatile2PC 和 Durable 2PC) 协议的 Register 请求消息, 其中参与方 Volatile2PC 服务终结点引用的地址为 <http://Adventure456.com/participant2PCservice>。

```

<Register>
  <ProtocolIdentifier>
    http://docs.oasis-open.org/ws-tx/ws-sat/2006/06/Volatile2PC
  </ProtocolIdentifier>
  <ParticipantProtocolService>
    <wsa:Address>
      http://Adventure456.com/participant2PCservice
    </wsa:Address>

```

```

<wsa:ReferenceParameters>
  <BetaMark>AlphaBetaGamma</BetaMark>
</wsa:ReferenceParameters>
</ParticipantProtocolService>
</Register>

```

当注册服务接收到 **Register** 请求后, 会返回具有如下结构的 **RegisterResponse** 消息。该消息中包含己方相应的协调服务的终结点引用。

```

<RegisterResponse ...>
  <CoordinatorProtocolService> ... </CoordinatorProtocolService>
  ...
</RegisterResponse>

```

3.3.2 WS-AT

WS-Coordination 为处理不同类型的协调定义了一个统一、可扩展的框架, 而原子事务 (**AT**, **Atomic Transaction**) 是众多协调类型中最为典型的一种。原子事务协调的对象是那些运行生命周期相对短暂的活动 (**Short Lived Activity**), 使之保持“要么全做, 要么都不做 (**All or Nothing**)”的属性。**WS-AT** 在 **WS-Coordination** 的基础上, 为原子事务这样一种协调类型定义了具体的协议, 使隶属于不同平台或厂商的事务处理系统可以进行互操作。

WS-AT 规范的制定者也是结构化信息标准促进组织 (**OASIS**)。目前具有 **WS-AT 1.0** 和 **WS-AT1.1** 两个版本, 以下的内容基于 **WS-AT 1.1**。由于 **WS-AT** 建立在 **WS-Coordination** 之上, 完全按照 **WS-Coordination** 规定的方式进行协调上下文的创建和服务注册, 因此下面先来简单介绍一下基于 **WS-AT** 的协调上下文。

1. WS-AT 协调上下文

当一个分布式事务的参与者需要将当前事务传播给另外一个事务参与者的时候, 需要调用本地协调器的激活服务创建基于原子事务协调类型的协调上下文, 并将上下文通过消息交换的方式进行传播。下面的 **XML** 展现了一个简单的原子事务协调上下文的结构。

```

<CoordinationContext>
  <Expires>100000</Expires>
  <Identifier>urn:uuid:f0660731-4eb8-4ed2-a7bd-1b3a68d2443a</Identifier>
  <CoordinationType>
    http://docs.oasis-open.org/ws-tx/wsac/2006/06
  </CoordinationType>
  <RegistrationService>
    <wsa:Address>
      http://Business456.com/tm/registration
    </wsa:Address>
    <wsa:ReferenceParameters>
      <myapp:PrivateInstance>1234</myapp:PrivateInstance>
    </wsa:ReferenceParameters>
  </RegistrationService>
</CoordinationContext>

```


其中表示协调类型的<CoordinationType>的值为 <http://docs.oasis-open.org/ws-tx/wsac/2006/06>, 代表基于 WS-AT 1.1 的原子事务协调类型 (这个 URI 也这正是 WS-AT 1.1 的命名空间)。<Expires>结点代表上下文过期的时限, 这实际上也就是分布式事务超时时限。<RegistrationService>是创建该上下文的协调器注册服务的终结点引用。

协调上下文通过调用协调器激活服务的 CreateCoordinationContext 操作获得。在 CreateCoordinationContext 请求中, 是可以指定一个现有的协调上下文的。在这种情况下, 注册服务会创建与该上下文相关联的上下文, 这种关联使它们具有相同的上下文标识。反映在具体事务场景中, 就意味着事务初始化应用调用原子事务协调器的激活服务创建一个全新的上下文, 并将其传播给后续的参与者。而事务后续参与者在创建上下文时, 会将接收到的上下文作为输入, 调用本地的协调器创建与之关联的上下文。两个原子事务协调器建立起一个上下级的关系, 两个上下文具有相同的标识即事务的分布式 ID。

2. 原子事务协议

WS-AT 为原子事务处理定义了两种基本的协议, 即 Completion 和两阶段提交 (2PC) 协议。

(1) Completion

分布式事务的初始化应用使用 Completion 协议通知协调器提交或者中止开启的原子事务。当事务结束后, 最终的状态 (被中止或者被提交) 返回到该应用。Completion 协议的协调器必须是原子事务的根协调器。对于非根协调器的注册请求, 注册服务将会返回 “Cannot Register Participant” WS-Coordination 错误 (Fault)。注册 Completion 协议采用的协议标识为 <http://docs.oasis-open.org/ws-tx/wsac/2006/06/Completion>。

(2) 两阶段提交 (2PC)

该协议通过协调原子事务的所有参与者, 对整个事务做出最终的决定 (提交或者中止), 并确保最终的结果通知到所有的参与者。根据事务参与者的不同, 两阶段提交协议具有两个变体, 即 Volatile 2PC 和 Durable 2PC。

- Volatile 2PC: 事务的参与者管理易失型事务资源 (Volatile Transactional Resource), 注册 Volatile 2PC 协议采用的协议标识为 <http://docs.oasis-open.org/ws-tx/wsac/2006/06/Volatile2PC>。
- Durable 2PC: 事务的参与者管理持久化型事务资源 (Durable Transactional Resource), 注册 Volatile 2PC 协议采用的协议标识为 <http://docs.oasis-open.org/ws-tx/wsac/2006/06/Durable2PC>。

限于篇幅, 我们不能够对 WS-AT 的三种协议进行深入的讨论, 有兴趣的读者可以从 OASIS 的网站上直接下载 WS-AT 的官方文档。接下来介绍 WCF 基于事务的编程问题。

3.4 WCF 事务编程

到此为止，本章的内容已经完成了一大半，但是我们似乎现在似乎才开始进入本章的主题。在我看来，对于 WCF 的很多知识点，如果单从编程的角度来讲都很简单，实在没有太多可书的地方。本书的目的旨在剖析 WCF 底层的运行机制，并在此基础上为采用 WCF 的用户提供最佳实践，所以有必要了解其依赖的平台（基于 DTC 的分布式事务框架）和支持的规范（OleTx 和 WS-AT）。之所以说 WCF 事务编程很简单，是因为我们可以用三句话进行概括。

- 通过服务契约决定事务流转（Transaction Flow）的策略。
- 通过绑定实施事务的流转。
- 通过服务/操作行为控制事务的相关行为。

3.4.1 通过服务契约决定事务流转的策略

契约是一种双边协定，是双方就某个关注点达成的一种共识。分布式事务的实现需要解决的是事务流转的问题，即将客户端的事务流向服务端。要解决事务的流转，需要在事务的发送方和接收方达成共识，即双方采用相匹配的事务发送和接收策略。毫无疑问，这样的开关需要定义在服务契约之上。同时事务是基于操作层面的，所以事务流转策略最终应用到服务契约的操作上面。

WCF 通过具有如下定义的 `System.ServiceModel.TransactionFlowAttribute` 特性将相应的事务流转策略关联到某个操作之上。我们在定义服务契约的时候，直接将 `TransactionFlowAttribute` 特性应用到相应的操作方法上。从下面的代码可以看到 `TransactionFlowAttribute` 并不仅仅是一个简单的自定义特性，它更是一个操作行为。至于该操作行为对事务流转行为做了怎样的控制，在下一节中会具体介绍。

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class TransactionFlowAttribute : Attribute, IOperationBehavior
{
    //其他成员
    public TransactionFlowAttribute(TransactionFlowOption transactions);
    public TransactionFlowOption Transactions { get; }
}
```

当我们将 `TransactionFlowAttribute` 特性应用到操作契约的时候，具体事务流转策略通过参数 `transactions` 指定。该参数是一个具有如下定义的 `System.ServiceModel.TransactionFlowOption` 枚举。

```
public enum TransactionFlowOption
{
```

```

    NotAllowed,
    Allowed,
    Mandatory
}

```

TransactionFlowOption 一共定义了三个选项 (NotAllowed、Allowed 和 Mandatory)，它们分别代表如下几个不同的事务流转策略。

- **NotAllowed:** 客户端的事务不允许被流转到服务端，服务端也不会试图去接收流入的事务，这是默认选项。
- **Allowed:** 如果客户端事务存在，则被流转到服务端，服务端会试图去接收流入的事务。
- **Mandatory:** 客户端必须在一个事务中进行服务调用，相应的事务会被流转到服务端。服务端接收到的消息中必须包含被序列化的事务。

在下面定义的 IBankingService 服务契约中，我们将 TransactionFlowAttribute 特性应用到了用于进行转账操作的 Transfer 方法之上，并指定事务流转选项为 Mandatory。

```

[ServiceContract(Namespace = "http://www.artech.com/")]
public interface IBankingService
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Mandatory)]
    void Transfer(string accountFrom, string accountTo, double amount);
}

```

对于 Mandatory 选项，如果客户端在进行服务调用的时候并不存在事务，或者说服务端并没有接收到任何流入的事务，都会抛出如图 3-13 所示的 ProtocolException 异常，提示“服务操作需要事务成为流”。

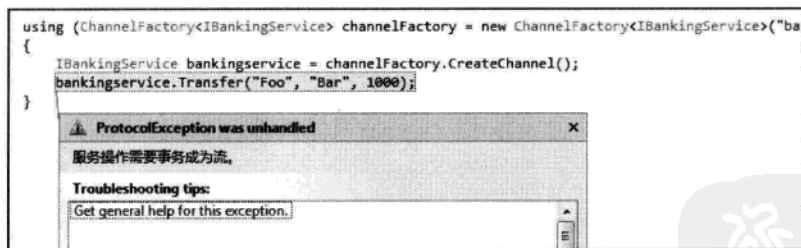


图 3-13 在选用 Mandatory 选项下客户端不存在事务导致的异常

由于分布式事务在客户端和服务端之间的协调过程依赖于它们之间进行的相互消息交换，这在单工的消息交换模式中是实现不了的，因此在一个单向 (One-Way) 操作契约上，不允许将应用在上面的 TransactionFlowAttribute 特性的参数指定为 Allowed 或者 Mandatory 选项。

```

[ServiceContract(Namespace = "http://www.artech.com/")]
public interface IBankingService
{
    [OperationContract(IsOneWay = true)]

```

```

[TransactionFlow(TransactionFlowOption.Mandatory)]
void Transfer(string accountFrom, string accountTo, double amount);
}

```

上面的服务契约定义是不合法的。如果某个服务实现了这样的服务契约, 在进行服务寄宿的时候, 会抛出如图 3-14 所示的 `InvalidOperationException` 异常, 提示“事务无法在单向操作上流动”。

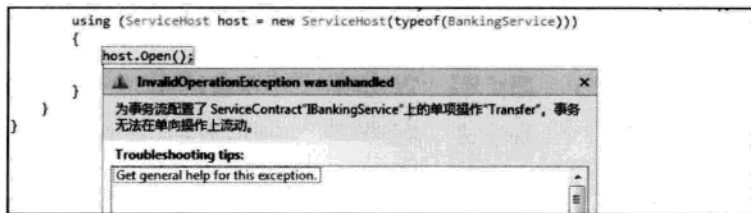


图 3-14 在单向操作上应用 `TransactionFlowAttribute` 特性导致的异常

3.4.2 通过绑定实施事务的流转

通过将 `TransactionFlowAttribute` 特性应用在服务契约的某个操作之上, 并指定相应的 `TransactionFlowOption` 枚举值, 仅仅定义了事务流转的策略而已。或者说通过这种方式确定对事务流转的一种意愿, 即客户端是否愿意将当前事务流出, 服务端是否愿意接收流入的事务。服务操作上定义的 `TransactionFlowAttribute` 特性是是否进行事务流转的总开关, 真正的事务传播是建立在 `TransactionFlowOption.Allowed` 或者 `TransactionFlowOption.Mandatory` 之上的。

至于 WCF 框架是否有能力对事务进行流转, 按照怎样的协议进行流转, 则是通过绑定实现的。下面我们首先看看怎样的绑定具有事务流转的能力。

1. 绑定对事务流转的支持

绑定是一系列绑定元素 (`BindingElement`) 的有序组合, 相应的绑定元素对消息进行相应的处理以实现特定的目标, 比如 `MessageEncodingBindingElement` 实现对消息的编码和解码, `TransportBindingElement` 实现对消息的传输。

消息交换是 WCF 进行通信的唯一手段, 任何需要传输的数据最终都需要作为消息的一部分。对于事务流转来说, 客户端需要将当前事务进行序列化并嵌入消息中。服务端则需要从接收到的消息中提取事务相关信息, 反序列化以重建事务。这样的操作同样实现在一个绑定元素中, 该绑定元素类型为 `System.ServiceModel.Channels.TransactionFlowBindingElement`。

既然 `TransactionFlowBindingElement` 实现了对事务的流转, 那么我们就可以根据某个绑定的绑定元素集合中是否包含该元素判断绑定是否支持事务流转。为此我写了如下一个

`PrintTransactionFlowSupport` 方法, 传入相应的 `Binding` 对象, 打印出相应的绑定类型是否支持事务流转。

```
static void PrintTransactionFlowSupport(Binding binding)
{
    TransactionFlowBindingElement transactionFlowElement =
        binding.CreateBindingElements().Find< TransactionFlowBindingElement>();
    Console.WriteLine("{0,-30} {1}",
        binding.GetType().Name, transactionFlowElement != null ? "Yes" : "No");
}
```

现在通过调用 `PrintTransactionFlowSupport` 方法, 判断常用的系统绑定是否为事务流转提供支持。从输出结果来看, 除了 `BasicHttpBinding`、`NetMsmqBinding` 和 `MsmqIntegrationBinding` 之外, 其余的系统绑定均包含 `TransactionFlowBindingElement` 绑定元素, 也就是说它们均具有对事务进行传播的能力。由于 `BasicHttpBinding` 基于 WS-I Basic Profile 标准的绑定, 而两个基于 MSMQ 的绑定 (`NetMsmqBinding` 和 `MsmqIntegrationBinding`) 只能采用单向 (One-Way) 的消息交换模式, 所以它们不具有事务流转的能力。

```
Console.WriteLine("{0,-30} {1}", "Binding", "Transaction Flow");
Console.WriteLine("-----");
//BasicHttpBinding
PrintTransactionFlowSupport(new BasicHttpBinding());

//WS Binding
PrintTransactionFlowSupport(new WSHttpBinding());
PrintTransactionFlowSupport(new WS2007HttpBinding());
PrintTransactionFlowSupport(new WSDualHttpBinding());
PrintTransactionFlowSupport(new WSFederationHttpBinding());
PrintTransactionFlowSupport(new WS2007FederationHttpBinding());

//TCP and IPC Binding
PrintTransactionFlowSupport(new NetTcpBinding());
PrintTransactionFlowSupport(new NetNamedPipeBinding());
//MSMQ Binding
PrintTransactionFlowSupport(new NetMsmqBinding());
PrintTransactionFlowSupport(new MsmqIntegrationBinding());
```

输出结果:

Binding	Transaction Flow
-----	-----
BasicHttpBinding	No
WSHttpBinding	Yes
WS2007HttpBinding	Yes
WSDualHttpBinding	Yes
WSFederationHttpBinding	Yes
WS2007FederationHttpBinding	Yes
NetTcpBinding	Yes
NetNamedPipeBinding	Yes
NetMsmqBinding	No
MsmqIntegrationBinding	No

即使对于支持事务的绑定, 事务流转选项默认也是被关闭的。在真正需要事务流转的场景中, 需要通过配置或者编程的方式开启该选项。事务流转还涉及事务在消息中的格式化问

题，而事务的格式化取决于采用的协议。通过上一节我们知道，WCF 支持三种不同的事务处理协议：OleTx，WS-AT 1.0 和 WS-AT 1.1。事务处理协议通过具有如下定义的 System.ServiceModel.TransactionProtocol 类型表示。

```
public abstract class TransactionProtocol
{
    public static TransactionProtocol Default { get; }

    public static TransactionProtocol OleTransactions { get; }
    public static TransactionProtocol WSAAtomicTransactionOctober2004 { get; }
    public static TransactionProtocol WSAAtomicTransaction11 { get; }
}
```

TransactionProtocol 是一个抽象类，定义了三种静态只读属性 OleTransactions、WSAtomicTransactionOctober2004 和 WSAAtomicTransaction11，用于获取分别代表 OleTx，WS-AT 1.0 和 WS-AT 1.1 三种协议的具体 TransactionProtocol 对象。Default 只读属性返回默认的事务处理协议，和 OleTransactions 属性值一致。

对于 NetTcpBinding 和 NetNamedPipeBinding 来说，我们可以通过属性 TransactionFlow 设置或者获取绑定是否支持事务流转的开关，并通过 TransactionProtocol 属性设置或者获取绑定支持的事务处理协议。

```
public class NetTcpBinding : Binding, IBindingRuntimePreferences
{
    //其他成员
    public bool TransactionFlow { get; set; }
    public TransactionProtocol TransactionProtocol { get; set; }
}

public class NetNamedPipeBinding : Binding, IBindingRuntimePreferences
{
    //其他成员
    public bool TransactionFlow { get; set; }
    public TransactionProtocol TransactionProtocol { get; set; }
}
```

对于基于 WS 的绑定来说，由于绑定本身就是为跨平台和互操作设计的，因此它们仅仅支持基于 WS-AT 的事务处理协议。其中 WSHttpBinding、WSDualHttpBinding 和 WSFederationHttpBinding 支持的协议是 WS-AT 1.0，而 WS2007HttpBinding 和 WS2007FederationHttpBinding 支持的是 WS-AT 1.1。所以它们仅仅具有 TransactionFlow 属性，并没有 TransactionProtocol 属性。TransactionFlow 属性定义在它们的基类 WSHttpBindingBase 上。

```
public abstract class WSHttpBindingBase : Binding, IBindingRuntimePreferences
{
    //其他成员
    public bool TransactionFlow { get; set; }
}
```

系统绑定的 TransactionFlow 和 TransactionProtocol 属性（仅限于 NetTcpBinding 和 NetNamedPipeBinding）可以通过配置的方式指定。下面的配置中定义了开启了 transactionFlow

开关的两个绑定 (NetTcpBinding 和 WS2007HttpBinding), 并将其中的 NetTcpBinding 的 TransactionProtocol 设置成基于 WS-AT 1.0 的协议(transactionProtocol="WSAtomicTransactionOctober2004")。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="transactionalTcpBinding"
          transactionFlow="true"
          transactionProtocol="WSAtomicTransactionOctober2004" />
      </netTcpBinding>
      <ws2007HttpBinding>
        <binding name="transactionalHttpBinding"
          transactionFlow="true" />
      </ws2007HttpBinding>
    </bindings>
    ...
  </system.serviceModel>
</configuration>
```

如果现有的系统绑定不能满足需要(比如需要同时采用 HTTP 传输协议和 OleTx 事务处理协议), 可以通过编程或者配置的方式创建自定义绑定。创建支持事务流转的自定义绑定的时候, 需要做的仅仅是将 TransactionFlowBindingElement 添加到绑定元素集合中, 并设置 TransactionFlow 和 TransactionProtocol 属性。下面的配置就定义了这样一个基于 OleTx 的 HTTP 绑定。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="transactionalBinding">
          <textMessageEncoding />
          <transactionFlow transactionProtocol="OleTransactions"/>
          <httpTransport />
        </binding>
      </customBinding>
    </bindings>
    ...
  </system.serviceModel>
</configuration>
```

2. 绑定与 TransactionFlow 设置

通过应用 TransactionFlowAttribute 特性为某个操作设置相应的事务流转策略, 而绑定决定了实现事务流转的能力和方式, 两者的不同组合表现出不同的事务流转行为。在这里事务的流转包含两个层面的意思, 即事务的流出或者发送, 以及事务的流入或者接收。

对于 WCF 客户端来说, 操作行为的 NotAllowed 和 Allowed 对绑定的事务流转能力没有任何要求, 而 Madantory 选项则强制要求终结点的绑定能够实现事务的流转(绑定本身能够

支持事务流转并且 TransactionFlow 开关必须开启)。结合上面所介绍的, 事务流转选项和绑定类型两两组合所表现出的行为如表 3-1 所示(这里的事务绑定表示 TransactionFlow 开关开启的支持事务流转的绑定)。

表 3-1 事务流转选项和绑定类型两两组合表现出的行为

	事务绑定	非事务绑定
NotAllowed	当前事务不需要存在, 存在的当前事务不会被流出	当前事务不需要存在, 存在的当前事务不会被流出
Allowed	当前事务不需要存在, 存在的当前事务会被流出	当前事务不需要存在, 存在的当前事务不会被流出
Mandatory	当前事务必须存在, 存在的当前事务会被流出	不合法的组合

对于一个服务契约来说, 如果任何一个操作的 TransactionFlow 选项被定义成 Mandatory, 相应终结点所采用的绑定必须是事务绑定。下面的代码和配置中, 通过 TransactionFlow Attribute 特性将唯一的 Transfer 操作的事务流转选项设置为 Mandatory, 并选用不支持事务流转的 BasicHttpBinding。

服务契约:

```
[ServiceContract (Namespace="http://www.artech.com/")]
public interface IBankingService
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Mandatory)]
    void Transfer(string accountFrom, string accountTo, double amount);
}
```

配置:

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address="http://127.0.0.1:3721/bankingservice"
        binding="basicHttpBinding"
        contract="Artech.Transactionalservices.IBankingService"
        name="bankingservice" />
    </client>
  </system.serviceModel>
</configuration>
```

当使用创建的 ChannelFactory<TChannel>创建服务代理的时候, 抛出如图 3-15 所示的 InvalidOperationException 异常, 提示“IBankingService 协定上至少有一个操作配置为将 TransactionFlowAttribute 属性设置为‘强制’, 但是通道的绑定‘BasicHttpBinding’未使用 TransactionFlowBindingElement 进行配置。没有 TransactionFlowBindingElement, 无法使用设置为‘强制’的 TransactionFlowAttribute 属性”。

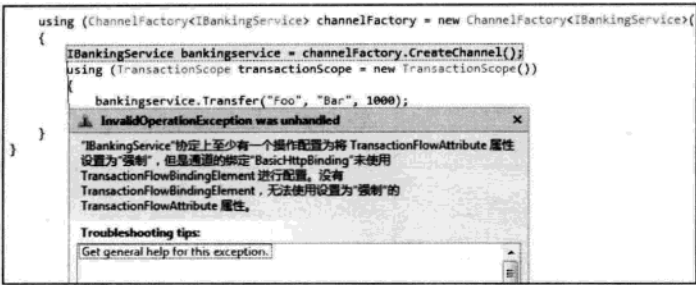


图 3-15 客户端在 Mandatory 事务流转选项情况下采用非事务绑定抛出的异常

上面所说的是不同的事务流转选项和绑定类型在客户端的表现行为, 现在我们将目光转移到服务端。较之客户端, 服务端的情况要稍微复杂一些, 除要考虑事务流转选项和绑定对事务流转的支持之外, 还需要考虑以下三个因素。

- 接收的消息 (SOAP 报头) 中是否具有包含流入事务。
- 表示流入事务的 XML 是否与绑定采用的事务处理协议一致。
- 如果不一致, 需要考虑事务 (SOAP) 报头的 MustUnderstand 属性是 True (或 1) 还是 False (或 0)。

WCF 服务端服务流转表现出来的最终行为决定于上述的 5 个要素, 表 3-2 列出了它们之间不同的组合最终表现出来的事务处理行为。

表 3-2 不同的事务流转选项和绑定类型组合表现出的事务处理行为

事务流转选项	是否事务绑定	消息是否包含事务	是否匹配事务协议	Must Understand	最终事务流转行为
NotAllowed	Yes	Yes	Yes	True	抛出异常, 因为事务报头不能理解
				False	忽略接收到的事务
		No	No	True	抛出异常, 因为事务报头不能理解
				False	忽略接收到的事务
	No	No	N/A	N/A	NIL
Allowed	Yes	Yes	Yes	True/False	正常处理事务
				True	抛出异常, 因为事务报头不能理解
			No	False	忽略接收到的事务
				True	抛出异常, 因为事务报头不能理解
				False	忽略接收到的事务
				True	抛出异常, 因为事务报头不能理解

续表

事务流转选项	是否事务绑定	消息是否包含事务	是否匹配事务协议	Must Understand	最终事务流转行为
Allowed	No	No	N/A	N/A	N/A
		Yes	N/A	True	抛出异常, 因为事务报头不能理解
			N/A	False	忽略接收到的事务
		No	N/A	N/A	N/A
Mandatory	Yes	Yes	Yes	True/False	正常处理事务
			No	True/False	抛出异常, 因为服务强制需要一个流入的事务
		No	N/A	N/A	N/A
	No	Yes/No	N/A	True/False	抛出异常, 因为 Mandatory 选项要求一个事务绑定

如果一个服务契约的任何一个操作的 TransactionFlow 选项定义成 Mandatory, 那么强制要求相应的终结点采用事务绑定。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.TransactionalServices.BankingService">
        <endpoint address="http://127.0.0.1:3721/bankingservice"
          binding="ws2007HttpBinding"
          contract="Artech.TransactionalServices.
            IBankingService"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

同样对于上面定义的 IBankingService 服务契约 (TransactionFlowOption.Mandatory), 但是采用上面的配置, 由于默认的 WS2007HttpBinding 的 TransactionFlow 是关闭的, 在进行服务寄宿的时候, 会抛出如图 3-16 所示的 InvalidOperationException 异常。

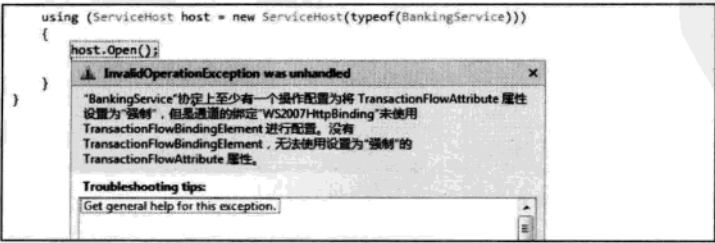


图 3-16 客户端在 Mandatory 事务流转选项情况下采用非事务绑定抛出的异常

其次,同样对于 TransactionFlow 选项为 Mandatory 的操作,如果接收的消息并不包含流入事务的 SOAP 报头,或者说流入的事务在 SOAP 报头中的表示并不符合绑定采用的事务处理协议,服务端会返回一个错误消息,并导致客户端抛出异常。

客户端配置:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="nonTransactionalBinding" transactionFlow="true"/>
      </netTcpBinding>
    </bindings>
    <client>
      <endpoint address="net.tcp://127.0.0.1:3721/bankingservice"
        binding="netTcpBinding"
        bindingConfiguration="nonTransactionalBinding"
        contract="Artech.TransactionalServices.IBankingService"
        name="bankingservice" />
    </client>
  </system.serviceModel>
</configuration>
```

服务端配置:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="transactionalBinding"
          transactionFlow="true"
          transactionProtocol="WSAtomicTransaction11" />
      </netTcpBinding>
    </bindings>
    <services>
      <service name="Artech.TransactionalServices.BankingService">
        <endpoint address="net.tcp://127.0.0.1:3721/bankingservice"
          binding="netTcpBinding"
          bindingConfiguration="transactionalBinding"
          contract="Artech.TransactionalServices.
            IBankingService"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

同样对于前面给定的 IBankingService 服务契约,如果我们采用上面的配置将客户端和服务端终节点的绑定配置成不同的事务处理协议(比如客户端采用默认的 OleTx,服务端则采用 WS-AT 1.1),客户端在进行服务调用的时候,会抛出如图 3-17 所示的 ProtocolException 异常。

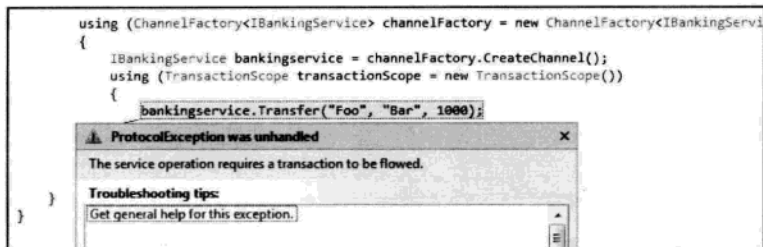


图 3-17 客户端和服务端采用不同的事务处理协议导致的异常 (Mandatory)

倘若接收到的消息中存在事务报头，并且报头的 MustUnderstand 属性为 True 或者 1，对于 Allowed 选项来说，如果采用非事务绑定，或者说虽然采用事务绑定，但是事务报头与绑定采用的事务处理协议不符，在这种情况下，服务端不能有效地理解事务报头，也会向客户端返回一个错误消息，并导致客户端抛出异常。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface IBankingService
{
    [OperationContract]
    [TransactionFlow(TransactionFlowOption.Allowed)]
    void Transfer(string accountFrom, string accountTo, double amount);
}
```

比如我们采用上面提供的配置（客户端和服务端绑定采用不同的事务处理协议），如果将服务契约 IBankingService 的 Transfer 操作的 TransactionFlow 选项设置为 Allowed，客户端在进行服务调用的时候会抛出如图 3-18 所示的 ProtocolException 异常。但是如果 MustUnderstand 属性为 False 或者 0，事务报头会被忽略。

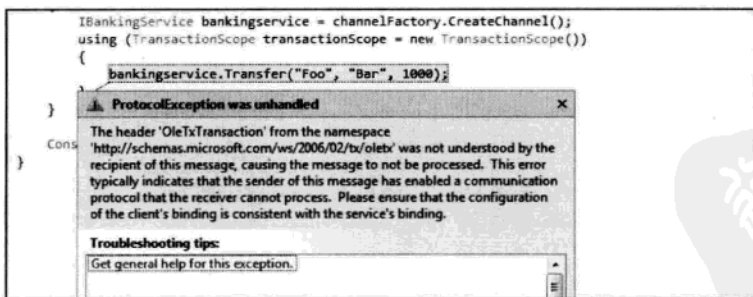


图 3-18 客户端和服务端采用不同的事务处理协议导致的异常 (Allowed)

相似情况同样发生在 TransactionFlow 选项为 NotAllowed 的时候。而对于后者，即使接收到的事务报头与绑定采用的事务处理协议相匹配，仍然会导致事务报头不能理解的异常。

3.4.3 通过服务（操作）行为控制事务

在 WCF 事务编程模型下，通过服务契约确定事务流转的策略，通过事务绑定实施事务的流转。对于事务绑定接收到并成功创建的事务来说，服务操作的执行是否需要自动登记到该事务之中，以及服务操作采用怎样的提交方式，就是服务端自己说了算。正因如此，WCF 通过服务（操作）行为的形式定义事务的登记和提交方式。

1. 事务的自动登记（Enlistment）与提交

在 `OperationBehaviorAttribute` 特性中定义了如下两个与事务管理相关的属性，即 `TransactionAutoComplete` 和 `TransactionScopeRequired`。

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute, IOperation
Behavior
{
    //其他成员
    public bool TransactionScopeRequired { get; set; }
    public bool TransactionAutoComplete { get; set; }
}
```

这两个属性均为布尔类型，它们代表的含义如下。

- **TransactionScopeRequired**: 表示相应操作的执行是否自动纳入一个事务中。具体来讲，如果客户端事务成功地流入服务端，将该属性设为 `True` 意味着整个操作的执行将自动被纳入流入的事务之中，服务操作将会成为客户端事务的一部分。如果服务端没有接收到任何流入的事务，将该属性设为 `True` 意味着操作的执行将会被纳入一个新创建的事务中。`TransactionScopeRequired` 的默认值为 `False`。
- **TransactionAutoComplete**: 表示如果操作执行过程中没有抛出异常，完成后将自动提交事务。`TransactionAutoComplete` 的默认值为 `True`。

如果需要将整个操作（而不是操作的一部分）纳入事务中执行，只需要按照如下的方式将 `OperationBehaviorAttribute` 特性应用到服务类型中的相应的方法之上，并将 `TransactionScopeRequired` 属性设为 `True` 即可。

```
public class BankingService : IBankingService
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void Transfer(string accountFrom, string accountTo, double amount)
    {
        //省略实现
    }
}
```

将 `TransactionAutoComplete` 属性设为 `True`（默认就是 `True`）可以使我们不考虑对本地事务的提交问题。只要执行完最后一句代码无异常抛出，就会提交事务。但是有时候需

要采用手工的方式提交事务。在这种情况下, 可以将该属性设为 `False`, 通过调用当前 `OperationContext` 的 `SetTransactionComplete` 方法实现对本地事务的提交。

```
public sealed class OperationContext : IExtensibleObject<OperationContext>
{
    //其他成员
    public void SetTransactionComplete();
}
```

除了定义在 `OperationBehaviorAttribute` 特性中的基于操作的行为, 还有一些与事务相关的服务行为, 它们定义在我们熟悉的 `ServiceBehaviorAttribute` 特性中。

2. 事务相关的服务行为

如下面的代码所示, `ServiceBehaviorAttribute` 特性定义了如下 4 个与事务相关的属性。

- `TransactionIsolationLevel`: 事务的隔离级别, 默认值为 `IsolationLevel.Serializable`。
- `TransactionTimeout`: 以字符串形式定义事务的超时时限。
- `TransactionAutoCompleteOnSessionClose`: 在会话正常结束 (没有出现异常) 之后是否自动提交或完成开启的事务, 默认值为 `False`。
- `ReleaseServiceInstanceOnTransactionComplete`: 当事务完成之后是否需要将服务实例释放掉, 默认为 `False`。

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public IsolationLevel TransactionIsolationLevel { get; set; }
    public string TransactionTimeout { get; set; }
    public bool TransactionAutoCompleteOnSessionClose { get; set; }
    public bool ReleaseServiceInstanceOnTransactionComplete { get; set; }
}
```

在下面的代码中, 通过在 `BankingService` 上应用 `ServiceBehaviorAttribute` 特性将隔离级别设置成 `ReadCommitted`, 将超时时限设置成 5 分钟, 并将 `TransactionAutoCompleteOnSessionClose` 设置成 `True`。

```
[ServiceBehavior(TransactionIsolationLevel = IsolationLevel.ReadCommitted,
    TransactionTimeout = "00:05:00",
    TransactionAutoCompleteOnSessionClose = true)]
public class BankingService : IBankingService
{
    //省略成员
}
```

当我们通过 `ServiceBehaviorAttribute` 特性对上述 4 个属性中的任何一个进行设置 (即使是设置成默认值) 时, 如果服务中并不存在一个 `TransactionScopeRequired` 属性为 `True` 的操作, 在进行服务寄宿的时候将会抛出异常。

```

[ServiceBehavior(TransactionIsolationLevel = IsolationLevel.ReadCommitted,
    TransactionTimeout = "00:05:00",
    TransactionAutoCompleteOnSessionClose = true)]
public class BankingService : IBankingService
{
    public void Transfer(string accountFrom, string accountTo, double amount)
    {
        //省略实现
    }
}

```

以上面的设置为例，在 `BankingService` 中唯一的 `Transfer` 方法上，并没有通过 `OperationBehaviorAttribute` 将 `TransactionScopeRequired` 属性设置为 `True`，在对服务进行寄宿的时候，就会抛出如图 3-19 所示的 `InvalidOperationException` 异常。

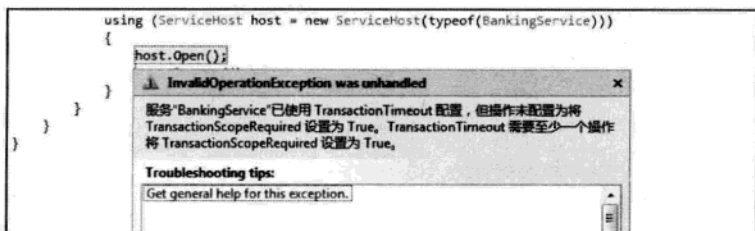


图 3-19 为不存在 `TransactionScopeRequired` 操作的服务设置相关服务行为导致的异常

通过 `TransactionTimeout` 设置的事务超时时限最终会被赋予 `ChannelDispatcher` 的同名属性。该属性的默认值为 `TimeSpan.Zero`。在这种情况下，运行时将会采用 `System.Transactions` 的默认超时设置。如果设置的 `TransactionTimeout` 的值超过了 `System.Transactions` 设置的最大超时时限，后者将会自动作为运行时的事务超时时限。

```

public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public TimeSpan TransactionTimeout { get; set; }
}

```

基于事务超时时限的服务行为也可以通过配置的方式指定，对应的配置节为 `<serviceTimeouts>`。只需按照所需的格式设置 `transactionTimeout` 属性值即可。在下面的配置中，将服务的事务超时时限设置成 30 分钟。

```

<configuration>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration="transactionBehavior" ...>
        ...
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="transactionBehavior">
          <serviceTimeouts transactionTimeout="00:30:00" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

```

        </behavior>
    </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

在事务流转的场景中,流入的事务和目标服务的事务隔离级别必须一致。也就是说,如果服务调用存在于客户端现有的事务之中,当前客户端事务必须和目标服务具有相同的隔离级别。否则,服务端将会返回相应的错误消息并导致客户端抛出异常。

```

using (ChannelFactory<IBankingService> channelFactory = new
ChannelFactory<IBankingService>("bankingservice"))
{
    IBankingService bankingservice = channelFactory.CreateChannel();
    using (TransactionScope transactionScope = new TransactionScope())
    {
        bankingservice.Transfer("Foo", "Bar", 1000);
        transactionScope.Complete();
    }
}

```

同样对于上面我们定义的 `BankingService(TransactionIsolationLevel = IsolationLevel.Read Committed)`,如果客户端按照下面的方式进行调用,由于客户端事务采用默认的隔离级别 `Serializable`,会抛出如图 3-20 所示的 `ProtocolException` 异常。

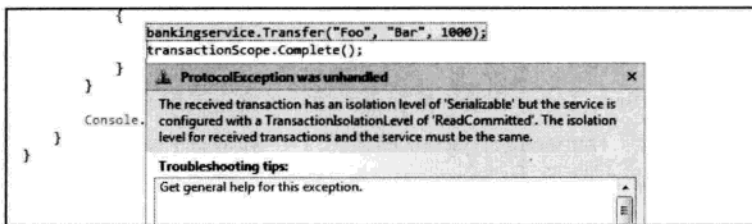


图 3-20 事务隔离级别不一致导致的异常

3.4.4 实例演示: 创建事务型服务 (S301)

本实例将提供一个完整的分布式事务的 WCF 服务应用。通过本例,读者不仅会了解到如何编程实现事务型服务,还会获得其他相关的知识(比如 DTC 和 WS-AT 的配置等)。本例还是沿用银行转账应用场景。我们将会创建一个 `BankingService` 服务,并将其中的转账操作定义成事务型操作。下面先从物理部署的角度来了解一下 `BankingService` 服务,以及需要实现怎样的分布式事务。

1. 从部署的角度看分布式事务

既然是实现分布式事务,那么事务会跨越多台机器,所以我使用两个机器来模拟。有条件的读者可以在自己的局域网中进行练习。假设两台机器名分别是 `Foo` 和 `Bar`,整个应用的

物理拓扑结构如图 3-21 所示。

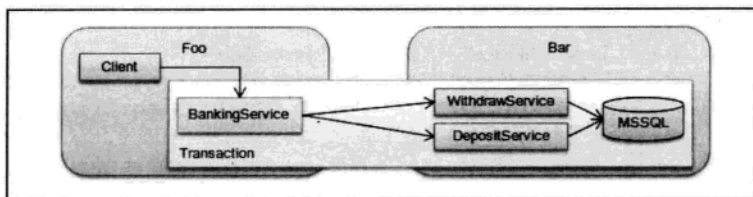


图 3-21 BankingService 物理部署拓扑

BankingService 和客户端部署于主机 Foo，定义在 BankingService 的转账的两个子操作“提取（Withdraw）”和“存储（Deposit）”通过调用部署于主机 Bar 的同名服务（WithdrawService 和 DepositService）实现。WithdrawService 和 DepositService 访问本地数据库。整个应用主要涉及三个服务（BankingService、WithdrawService 和 DepositService），下面先来看看服务契约和服务的实现。

2. 步骤 1：服务契约和服务的实现

我们仍然采用契约共享的方式将服务契约定义在单独的项目之中，供服务端和客户端共享。涉及的三个服务对应的服务契约定义如下，事务型操作的 TransactionFlow 选项被设置为 Allowed（默认值）。

IBankingService:

```
using System.ServiceModel;
namespace Artech.TransactionService.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/banking/")]
    public interface IBankingService
    {
        [OperationContract]
        [TransactionFlow(TransactionFlowOption.Allowed)]
        void Transfer(string fromAccountId, string toAccountId, double amount);
    }
}
```

IWithdrawService:

```
using System.ServiceModel;
namespace Artech.TransactionService.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/banking/")]
    public interface IWithdrawService
    {
        [OperationContract]
        [TransactionFlow(TransactionFlowOption.Allowed)]
        void Withdraw(string accountId, double amount);
    }
}
```

IDepositService:

```
using System.ServiceModel;
namespace Artech.Transactionalservice.Service.Interface
{
    [ServiceContract(Namespace="http://www.artech.com/banking/")]
    public interface IDepositService
    {
        [OperationContract]
        [TransactionFlow(TransactionFlowOption.Allowed)]
        void Deposit(string accountId, double amount);
    }
}
```

实现了契约接口 `IWithdrawService` 和 `IDepositService` 的 `WithdrawService` 和 `DepositService` 分别实现基于给定银行账户的提取和存储操作。限于篇幅,具体对数据库相应数据的更新操作就不在这里一一介绍了。下面是 `WithdrawService` 和 `DepositService` 的定义,由于不管是单独被调用,还是作为转账的一个子操作, `Withdraw` 和 `Deposit` 操作均需要在一个事务中执行,因此需要通过应用 `OperationBehaviorAttribute` 将 `TransactionScopeRequired` 属性设为 `True`。

WithdrawService:

```
using System.ServiceModel;
using Artech.Transactionalservice.Service.Interface;
namespace Artech.Transactionalservice.Service
{
    public class WithdrawService : IWithdrawService
    {
        [OperationBehavior(TransactionScopeRequired = true)]
        public void Withdraw(string accountId, double amount)
        {
            //省略实现
        }
    }
}
```

DepositService:

```
using System.ServiceModel;
using Artech.Transactionalservice.Service.Interface;
namespace Artech.Transactionalservice.Service
{
    public class DepositService : IDepositService
    {
        [OperationBehavior(TransactionScopeRequired = true)]
        public void Deposit(string accountId, double amount)
        {
            //省略实现
        }
    }
}
```

定义在 `BankingService` 的 `Transfer` 操作就是调用上述的两个服务,由于服务调用涉及对服务代理的关闭及异常的处理(相关的内容在上册的第8章“客户端(Client)”中有详细的

介绍), 为了实现代码的复用, 我们定义了一个静态的 `ServiceInvoker` 类。`ServiceInvoker` 定义如下, 泛型方法 `Invoke<TChannel>` 用于进行服务的调用, 并实现了服务代理的关闭 (`Close`), 异常抛出是对服务代理的中止 (`Abort`)。 `Invoke<TChannel>` 的泛型参数类型为服务契约类型, 方法接受两个操作, 委托 `action` 代表服务调用操作, `endpointConfigurationName` 表示配置的终结点名称。

```
using System;
using System.ServiceModel;
namespace Artech.Transactionalservice.Service.Interface
{
    public static class ServiceInvoker
    {
        public static void Invoke<TChannel>(Action<TChannel> action, string
            endpointConfigurationName)
        {
            Guard.ArgumentNotNull(action, "action");
            Guard.ArgumentNotNullOrEmpty(endpointConfigurationName,
                "endpointConfigurationName");

            using (ChannelFactory<TChannel> channelFactory = new
                ChannelFactory<TChannel>(endpointConfigurationName))
            {
                TChannel channel = channelFactory.CreateChannel();
                using (channel as IDisposable)
                {
                    try
                    {
                        action(channel);
                    }
                    catch (TimeoutException)
                    {
                        (channel as ICommunicationObject).Abort();
                        throw;
                    }
                    catch (CommunicationException)
                    {
                        (channel as ICommunicationObject).Abort();
                        throw;
                    }
                }
            }
        }
    }
}
```

借助于 `ServiceInvoker`, `BankingService` 的定义就很简单了。对于 `Transfer` 操作, 我们依然通过 `OperationBehaviorAttribute` 特性将 `TransactionScopeRequired` 设置成 `True`。

```
using System;
using System.Collections.Generic;
using System.ServiceModel;
using Artech.Transactionalservice.Service.Interface;
```

```

namespace Artech.TransactionService.Service
{
    public class BankingService : IBankingService
    {
        [OperationBehavior(TransactionScopeRequired = true)]
        public void Transfer(string fromAccountId, string toAccountId, double amount)
        {
            ServiceInvoker.Invoke<IWithdrawService>(proxy =>
                proxy.Withdraw(fromAccountId, amount),
                "withdrawservice");
            ServiceInvoker.Invoke<IDepositService>(proxy =>
                proxy.Deposit(toAccountId, amount),
                "depositservice");
        }
    }
}

```

3. 步骤 2: 部署服务

BankingService 和它所依赖的 WithdrawService 与 DepositService 已经定义好了, 现在需要对它们进行部署。本实例采用基于 IIS 的服务寄宿方式, 在进行部署之前需要为三个服务创建 .svc 文件。在这里, 将 .svc 文件命名为与服务类型相同的名称 (BankingService.svc、WithdrawService.svc 和 DepositService.svc)。

我们需要分别在主机 Foo 和 Bar 上创建两个 IIS 虚拟目录 (假设名称为 Banking), 并将定义服务契约和服务类型的两个程序集复制到 Foo\Banking\Bin 和 Bar\Banking\Bin。然后再将 BankingService.svc 复制到 Foo\Banking 下, 将 WithdrawService.svc 和 DepositService.svc 复制到 Bar\Banking 下。最后, 需要创建两个 Web.config, 分别复制到 Foo\Banking\Bin 和 Bar\Banking 下面。下面两段 XML 代表两个 Web.config 的配置。

Bar\Banking\Web.config:

```

<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="transactionalBinding">
          <textMessageEncoding />
          <transactionFlow/>
          <httpTransport/>
        </binding>
      </customBinding>
    </bindings>
    <services>
      <service name="Artech.TransactionService.Service.WithdrawService">
        <endpoint binding="customBinding"
          bindingConfiguration="transactionalBinding"
          contract="Artech.TransactionService.Service.Interface.IWithdrawService" />
      </service>
      <service name="Artech.TransactionService.Service.DepositService">
        <endpoint binding="customBinding"
          bindingConfiguration="transactionalBinding"
          contract="Artech.TransactionService.Service.Interface.IDepositService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

```

    </service>
  </services>
</system.serviceModel>
</configuration>

```

Foo\Banking\Web.config:

```

<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="transactionalBinding">
          <textMessageEncoding />
          <transactionFlow/>
          <httpTransport/>
        </binding>
      </customBinding>
    </bindings>
    <services>
      <service name="Artech.Transactionalservice.Service.BankingService">
        <endpoint binding="customBinding"
bindingConfiguration="transactionalBinding"
contract="Artech.Transactionalservice.Service.Interface.IBankingService" />
      </service>
    </services>
    <client>
      <endpoint name="withdrawservice"
        address="http://Bar/banking/withdrawservice.svc"
        binding="customBinding"
        bindingConfiguration="transactionalBinding"
contract="Artech.Transactionalservice.Service.Interface.IWithdrawService" />
      <endpoint name="depositservice"
        address="http://Bar/banking/depositservice.svc"
        binding="customBinding"
        bindingConfiguration="transactionalBinding"
contract="Artech.Transactionalservice.Service.Interface.IDepositService"/>
    </client>
  </system.serviceModel>
</configuration>

```

4. 步骤 3: 调用 BankingService

已经部署好了定义三个服务，现在我们可以调用它们实施转账处理了。可以像调用普通服务一样调用 `BankingService`，无须考虑事务的问题。因为我们通过 `OperationBehaviorAttribute` 特性将 `BankingService` 的 `Transfer` 操作的 `TransactionScopeRequired` 设置成 `True`，这会确保整个操作的执行是在一个事务（可能是流入的事务，也可能是重新创建的事务）中进行的。下面是进行转账处理的客户端代码和配置。

服务调用程序：

```

string fromAccountId = "123456789";
string tooAccountId = "987654321";
double amount = 1000;
ServiceInvoker.Invoke<IBankingService>(proxy => proxy.Transfer(fromAccountId,
tooAccountId, amount), "bankingservice");

```

配置:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="transactionalBinding">
          <textMessageEncoding />
          <transactionFlow/>
          <httpTransport/>
        </binding>
      </customBinding>
    </bindings>
  </system.serviceModel>
  <client>
    <endpoint name="bankingservice"
      address="http://Foo/banking/bankingservice.svc"
      binding="customBinding"
      bindingConfiguration="transactionalBinding"
      contract="Artech.Transactionalservice.Service.Interface.IBankingService"/>
  </client>
</configuration>
```

定义在 **BankingService** 的 **Transfer** 操作完全是通过调用 **WithdrawService** 和 **DepositService** 实现的, 我们也可以绕过 **BankingService** 直接调用这两个服务实现转账的处理。为此需要在配置中添加调用 **WithdrawService** 和 **DepositService** 的终结点。(S302)

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="withdrawservice"
        address="http://Bar/banking/withdrawservice.svc"
        binding="customBinding"
        bindingConfiguration="transactionalBinding"
        contract="Artech.Transactionalservice.Service.Interface.IWithdrawService"/>
      <endpoint name="depositervice"
        address="http://Bar/banking/depositervice.svc"
        binding="customBinding"
        bindingConfiguration="transactionalBinding"
        contract="Artech.Transactionalservice.Service.Interface.IDepositService" />
    </client>
  </system.serviceModel>
</configuration>
```

由于整个转账的操作必须纳入一个事务中进行, 并且客户端主动发起对 **WithdrawService** 和 **DepositService** 两个服务的调用, 所以客户端是事务的初始化者。我们需要将对这两个服务的调用放到一个 **TransactionScope** 中进行, 相应的代码如下所示。

```
string fromAccountId = "123456789";
string toAccountId = "987654321";
double amount = 1000;
using (TransactionScope transactionScope = new TransactionScope())
{
```

```

ServiceInvoker.Invoke<IWithdrawService>(proxy => proxy.Withdraw
(fromAccountId, amount), "withdrawservice");
ServiceInvoker.Invoke<IDepositService>(proxy => proxy.Deposit
(toAccountId, amount), "depositservice");
transactionScope.Complete();
}

```

上面两种不同的实现方式,实际上都已经涉及分布式事务的应用,所以需要借助于 DTC。如果读者在运行该实例的时候,两个主机的 DTC 没有进行合理的设置,将不会成功运行。下面简单介绍一下如何进行 DTC 的设置。

5. 步骤 4: 设置 DTC

通过“控制面板”|“管理工具”|“组件服务”打开组件服务的对话框。然后右击“组件服务”|“计算机”|“我的电脑”节点,并在上下文菜单中选择“属性”,会弹出“我的电脑属性”对话框。在该对话框的“MSDTC”Tab 页选择默认的协调器,一般选择“使用本地协调器”选项。

选择使用本地协调器作为默认的 DTC 之后,在组件服务对话框的“组件服务”|“计算机”|“我的电脑”|“Distributed Transaction Coordinator”节点下面会出现“本地 DTC”节点。右击该节点选择“属性”选项,会弹出如图 3-22 所示的“本地 DTC 属性”。可以对 DTC 的跟踪(Trace)方式、日志记录、安全和 WS-AT 进行相应的设置。

在这里要使 DTC 在本实例中可用,重点是对“安全”进行正确的设置。图 3-22 是我机器上的设置,限于篇幅问题,不能每一个选项进行详细说明,有兴趣的读者可以很容易地从网上找到相关的参考资料。如果已经对 DTC 做了相应设置之后还出现 DTC 的问题,可以看看 DTC 通信是否被防火墙屏蔽了。

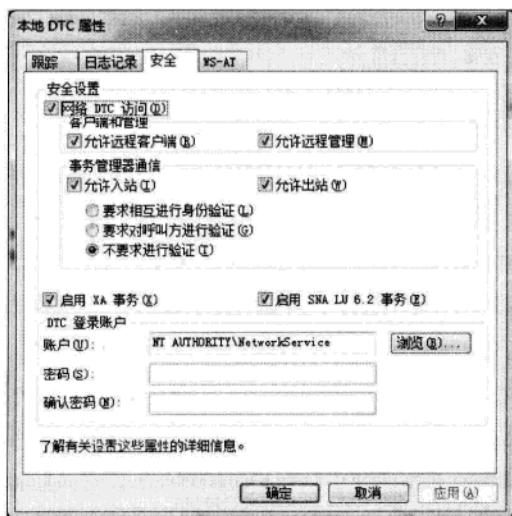


图 3-22 本地 DTC 设置对话框

6. 步骤5: 采用 WS-AT 协议

在本例中, 所有终结点采用的绑定类型均是包含有 `TransactionFlowBindingElement` 的自定义绑定。通过前面的介绍我们知道, 默认采用的事务处理协议是 `OleTx`。如果希望采用 `WS-AT` 协议, 需要通过配置将协议类型改成 `WSAtomicTransactionOctober2004` 或者 `WSAtomicTransaction11`。在下面的配置中, 我们将实例中使用的绑定支持的事务处理协议设置成 `WSAtomicTransaction11`, 使之采用 `WS-AT` 协议进行事务处理。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="TransactionalBinding">
          <textMessageEncoding />
          <transactionFlow transactionProtocol="WSAtomicTransaction11" />
          <httpTransport/>
        </binding>
      </customBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

DTC 通过基于 HTTPS 的通信方式为 `WS-AT` (1.0 和 1.1) 提供实现。我们知道 HTTPS 通过 SSL 实现传输安全, 而 SSL 需要将相应的证书 (Certificate) 绑定在 HTTPS 站点上面。为了让 DTC 支持 `WS-AT`, 需要对 DTC 进行相关的配置。

首先需要为参与到事务的两台主机创建相对应的证书, 在这里直接采用 `Makecert.exe` 这个 X.509 证书生成工具。可以通过下面两个命令行创建两张 X.509 证书, 分别代表 `Foo` 和 `Bar` 两台主机, 读者在练习这个例子时, 需要换成自己相应的机器名称。关于 `Makecert.exe` 的命令行选项的含义, 可以参考 MSDN (<http://msdn.microsoft.com/zh-cn/library/bfsktyk3%28VS.80%29.aspx>)。

```
MakeCert -n CN=Foo -pe -sky exchange -sr LocalMachine -ss MY
MakeCert -n CN=Bar -pe -sky exchange -sr LocalMachine -ss MY
```

在主机 `Foo` 中运行上面的命令行后, 会创建两张 X.509 证书并将其存入“本地计算机\个人 (LocalMachine\MY)”存储中。需要通过 MMC 将其导出成证书文件, 并将其导入主机 `Foo` 的“本地计算机\受信任的根证书颁发机构 (LocalMachine\Root)”存储中, 以及主机 `Bar` 的“本地计算机\个人 (LocalMachine\MY)”和“本地计算机\受信任的根证书颁发机构 (LocalMachine\Root)”存储中。需要注意的是, 在导出的时候务必选择“导出私钥”选项, 因为不包含私钥的证书是不能和 SSL 站点绑定的。

DTC 的 `WS-AT` 可以借助于 `WS-AT` 配置工具 `wsatConfig.exe` 以命令行的方式进行设置, 该工具位于“%WINDIR%\Microsoft.NET\Framework\v3.0\Windows Communication Foundation”目录下面。关于 `wsatConfig.exe` 配置工具的用法, 可以参考 MSDN 相关文档 (<http://msdn.microsoft.com/zh-cn/library/ms732007.aspx>)。在这里我需要介绍的是一种可视化的 `WS-AT` 配置方式。

如图 3-22 所示的 DTC 设置对话框中, 有一个 WS-AT Tab 页, 通过它可以很容易地进行 WS-AT 的相关配置。不过在默认的情况下, 这个 Tab 页是不存在的。需要借助于 Regasm.exe 这个程序集注册工具以命令行的形式对包含有 WS-AT 配置界面的程序集进行注册, 该程序集位于 Windows SDK 目录下: %PROGRAMFILES%\Microsoft SDKs\Windows\v6.0\Bin 或者 %PROGRAMFILES%\Microsoft SDKs\Windows\v6.0A\Bin。当运行下面的命令行后, DTC 设置对话框中就会出现 WS-AT Tab 页了。

```
regasm.exe /codebase WsatUI.dll
```

选择 WS-AT Tab, 将会看到如图 3-23 所示的界面。然后我们针对主机 Foo 和 Bar 分别进行如下的设置, 使之建立相互信任关系。

- Foo: 选择证书 Foo(CN=Foo)和 Bar(CN=Bar)分别作为终结点证书(Endpoint certificate)和授权证书(Authorized certificates)。
- Bar: 选择证书 Bar(CN=Bar)和 Foo(CN=Foo)分别作为终结点证书(Endpoint certificate)和授权证书(Authorized certificates)。

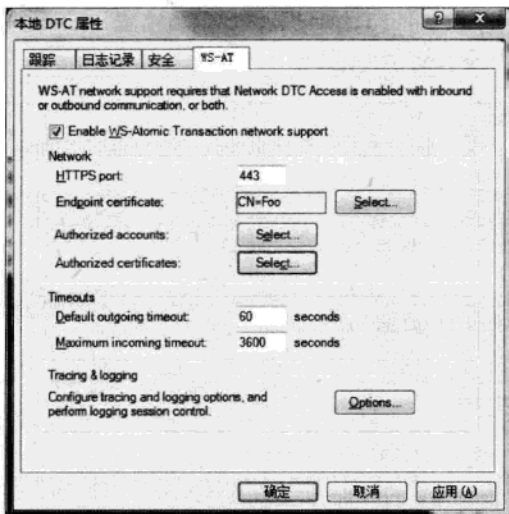


图 3-23 WS-AT 设置界面

对于本实例来说, 即使你将绑定的事务处理协议设置成 WSAAtomicTransactionOctober2004 或者 WSAAtomicTransaction11, 并对 WS-AT 进行了正确的设置, 但是依然采用的是 OleTx 协议。这是由于 DTC 的 OleTx 提升(OleTx Upgrade)机制导致的。关于 OleTx 提升机制, 会在本章后续部分介绍。如果希望本实例真正采用 WS-AT 进行事务控制, 你需要显式关闭 OleTx 的自动提升。我们只需要通过注册表编辑器在 HKLM\SOFTWARE\Microsoft\WSAT3.0 节点下添加一个名称为 OleTxUpgradeEnabled 的双字节(DWORD)注册表项, 并将值设为 0。

3.5 WCF 事务实现原理

WCF 事务编程主要涉及三个方面：通过服务（操作）契约确定 TransactionFlow 的策略，通过事务绑定实现事务流转，通过服务操作行为控制事务的自动登记（Enlistment）行为，以及对事务超时时限、隔离级别和实例行为的设定。那么在 WCF 内部这三者之间究竟是如何通过相互协作实现分布式事务的呢？

3.5.1 TransactionFlowAttribute 行为

通过将 TransactionFlowAttribute 特性应用于服务契约的某个操作，实际上是指定了事务流转的策略（NotAllowed、Allowed 和 Mandatory）。绑定最终需要根据设置的 TransactionFlow 选项，决定是否对事务实施流转。那么两者是如何联系在一起的呢？

TransactionFlowAttribute 并不是一个简单的特性，它是一个实现了 IOperationBehavior 接口的操作行为。如下面的伪代码所示，设置的 TransactionFlow 选项通过 AddBindingParameters 这个方法被传入了绑定上下文（BindingContext）。

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class TransactionFlowAttribute : Attribute, IOperationBehavior
{
    public TransactionFlowAttribute(TransactionFlowOption transactions);
    void IOperationBehavior.AddBindingParameters(OperationDescription
description, BindingParameterCollection parameters)
    {
        //将 TransactionFlow 选项 (NotAllowed、Allowed 和 Mandatory) 存入绑定上下文
        AddTransactionFlowOptionInBindingContext();
    }
    void IOperationBehavior.ApplyClientBehavior(OperationDescription
description, ClientOperation proxy){ }
    void IOperationBehavior.ApplyDispatchBehavior(OperationDescription
description, DispatchOperation dispatch) {}
    void IOperationBehavior.Validate(OperationDescription description) { }

    public TransactionFlowOption Transactions { get; }
}
```

传入绑定上下文的 TransactionFlow 选项可以被绑定创建的信道（Channel）获取并根据相应的值控制自身消息处理的行为。在这里真正使用到该 TransactionFlow 选项的信道，就是通过事务绑定的 TransactionFlowBindingElement 创建的事务信道。关于绑定、绑定元素和信道之间的关系，在上册的第3章“绑定（Binding）”中有详细的介绍。

3.5.2 事务绑定

由于消息交换是 WCF 进行通信的唯一手段，因此事务的流转最终需要将事务本身作为

消息的一部分进行传输，而这是通过事务绑定实现的。我们所说的事务绑定就是包含有 TransactionFlowBindingElement 绑定元素，并且 TransactionFlow 开关被开启的绑定。

对于客户端来说，TransactionFlowBindingElement 创建事务信道工厂 (TransactionChannelFactory)。而基于不同的信道形状 (Channel Shape) 和对会话的支持，事务信道工厂会创建相应的事务信道，比如事务输出信道 (TransactionOutputChannel)、事务请求信道 (TransactionRequestChannel) 和事务双工信道 (TransactionDuplexChannel) 等。

对于服务端来说，TransactionFlowBindingElement 会创建事务信道监听器 (TransactionChannelListener)，而事务信道监听器也会创建基于不同信道形状 (Channel Shape) 和对会话的支持的事务信道，比如事务输入信道 (TransactionInputChannel)、事务回复信道 (TransactionReplyChannel) 和事务双工信道 (TransactionDuplexChannel) 等。事务流转相关的绑定元素、绑定管理器 (信道工厂和信道监听器) 和信道之间的关系如图 3-24 所示。

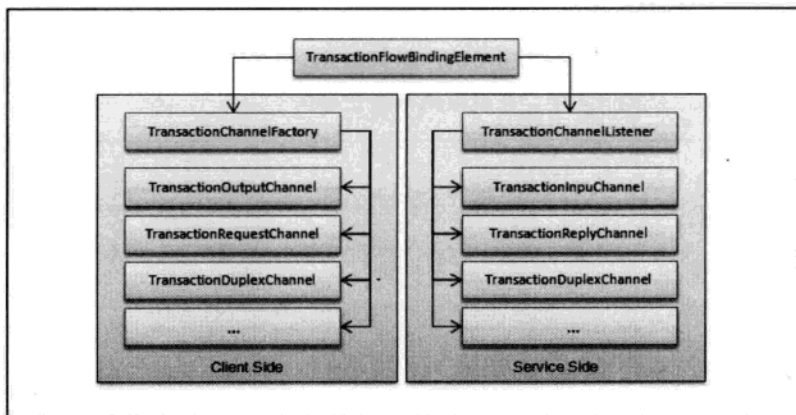


图 3-24 事务流转相关的绑定元素、信道管理器和信道结构之间的关系

客户端的事务信道需要将当前事务写入消息，而服务端的事务信道则需要将流入的事务从服务中读出来。WCF 将事务的读写操作定义在一个称为 TransactionFormatter 的类型中。这是一个内部类型，不能被直接使用。TransactionFormatter 的定义大体上如下面的代码所示，其中 ReadTransaction 和 WriteTransaction 分别实现对事务的读取和写入操作。事务通过 TransactionInfo 对象的形式被读取出来。TransactionInfo 也是一个内部对象，可以通过调用 UnmarshalTransaction 得到真正的 Transaction 对象。

```
internal abstract class TransactionFormatter
{
    //其他成员
    public abstract TransactionInfo ReadTransaction(Message message);
    public abstract void WriteTransaction(Transaction transaction, Message message);
}
```

```
internal abstract class TransactionInfo
{
    //其他成员
    public abstract Transaction UnmarshalTransaction();
}
```

如果采用不同事务处理协议，相同的事务需要按照不同的方式进行格式化，所以 WCF 事务体系内部创建了继承自 `TransactionFormatter` 的三个具体的 `TransactionFormatter` 类型 (`OleTxTransactionFormatter`、`WsatTransactionFormatter10` 和 `WsatTransactionFormatter11`)，它们分别对应于 OleTx、WS-AT 1.0 和 WS-AT 1.1 三种协议。我想很多人很想知道一个 `Transaction` 对象被不同的 `TransactionFormatter` 写入到 `Message` 对象后，`Message` 具有怎样的格式。接下来通过一个简单的实例来演示。

实例演示：通过 `TransactionFormatter` 进行事务的写入 (S303)

本实例是一个简单的控制台应用，我们将用它来演示事务绑定是如何将当前事务写入消息的。由于上面提到的 `TransactionFormatter` 和 `TransactionInfo` 都是内部类型，因此只能通过反射的方式使用它们。为此我写了一个简单的工具类型 `ReflectUtil`，用于通过反射的方式创建对象和调用某个方法，原理很简单，在这里就不多做介绍了。

```
internal static class ReflectUtil
{
    public static object CreateInstance(string typeAssemblyQName, params
    object[]
    parameters)
    {
        Type typeofInstance = Type.GetType(typeAssemblyQName);
        BindingFlags bindingFlags = BindingFlags.Instance | BindingFlags.
        Public | BindingFlags.NonPublic;
        return Activator.CreateInstance(typeofInstance, bindingFlags, null,
        parameters, null);
    }
    public static object Invoke(string methodName, object targetInstance, params
    object[] parameters)
    {
        BindingFlags bindingFlags = BindingFlags.Instance | BindingFlags.
        NonPublic | BindingFlags.Public;
        return targetInstance.GetType().GetMethod(methodName,
        bindingFlags).Invoke(targetInstance, parameters);
    }
}
```

然后我们创建自己的 `TransactionFormatter` 类型，它具有相同的方法 `ReadTransaction` 和 `WriteTransaction`，`ReadTransaction` 的返回类型直接是 `Transaction`。相应的事务协议通过构造函数指定，事务协议决定了最终创建的真正的 `TransactionFormatter` 的类型。真正的 `TransactionFormatter` 通过 `ReflectUtil` 创建，相应的方法也通过 `ReflectUtil` 调用。

```
public class TransactionFormatter
{
    const string OleTxFormatterType =
```

```

"System.ServiceModel.Transactions.OleTxTransactionFormatter, System.ServiceModel, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089";
    const string Wsat10FormatterType =
"System.ServiceModel.Transactions.WsatTransactionFormatter10, System.ServiceModel, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089";
    const string Wsat11FormatterType =
"System.ServiceModel.Transactions.WsatTransactionFormatter11, System.ServiceModel, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089";

    public object InternalFormatter{ get; private set; }

    public TransactionFormatter(TransactionProtocol transactionProtocol)
    {
        if (transactionProtocol == TransactionProtocol.OleTransactions)
        {
            this.InternalFormatter = ReflectUtil.CreateInstance(OleTxFormatterType);
        }
        else if (transactionProtocol ==
            TransactionProtocol.WSAtomicTransactionOctober2004)
        {
            this.InternalFormatter =
                ReflectUtil.CreateInstance(Wsat10FormatterType);
        }
        else
        {
            this.InternalFormatter =
                ReflectUtil.CreateInstance(Wsat11FormatterType);
        }
    }

    public Transaction ReadTransaction(Message message)
    {
        object transInfo = ReflectUtil.Invoke("ReadTransaction",
            this.InternalFormatter, message);
        return ReflectUtil.Invoke("UnmarshalTransaction", transInfo) as
            Transaction;
    }

    public void WriteTransaction(Transaction transaction, Message message)
    {
        ReflectUtil.Invoke("WriteTransaction", this.InternalFormatter,
            transaction, message);
    }
}

```

接下来我们基于三种不同的事务处理协议创建了相应的 `TransactionFormatter` 对象, 并将相同的 `Transaction` 对象写入一个 `Message` 对象中, 并且 `Message` 的主体部分为 `Transaction` 对象本身。最终的 `Message` 对象被写入三个 XML 文件中。

```

static void Main(string[] args)
{
    using (TransactionScope transactionScope = new TransactionScope())
    {
        WriteTransaction(TransactionProtocol.OleTransactions, Transaction.
            Current, "oletx.xml");
    }
}

```

```

        WriteTransaction(TransactionProtocol.WSAtomicTransactionOctober
            2004, Transaction.Current, "wsat10.xml");
        WriteTransaction(TransactionProtocol.WSAtomicTransaction11,
            Transaction.Current, "wsat11.xml");
    }
}

static void WriteTransaction(TransactionProtocol transactionProtocol,
    Transaction transaction, string fileName)
{
    string action = string.Format("http://www.artech.com/
        transactionformat/{0}",
            transactionProtocol.GetType().Name);
    Message message = Message.CreateMessage(MessageVersion.Default, action,
        Transaction.Current);
    TransactionFormatter formatter = new TransactionFormatter(transac
        tionProtocol);
    formatter.WriteTransaction(Transaction.Current, message);
    using (XmlWriter writer = new XmlTextWriter(fileName, Encoding.UTF8))
    {
        message.WriteMessage(writer);
    }
    Process.Start(fileName);
}

```

程序成功运行后, 你将会得到三个表示 Message 对象的 XML 文件, 它们的内容如下。有兴趣的读者可以结合相应事务处理协议规范, 认真分析一下对应消息的结构, 相信可以加深对事务处理协议的理解。

OleTx.xml (OleTx):

```

<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
    <a:Action
s:mustUnderstand="1">http://www.artech.com/transactionformat/OleTransactio
nsProtocol</a:Action>
    <OleTxTransaction s:mustUnderstand="1" d3p1:Expires="59392"
xmlns:d3p1="http://schemas.xmlsoap.org/ws/2004/10/wscoor"
xmlns="http://schemas.microsoft.com/ws/2006/02/tx/oletx">
      <PropagationToken>AQAAAAAAdknHb/v0zMQKhDZHfsF0Y1AAQAQAAAAACEAAAAAMW2bRzF
tm00W6xnBOMYAHfXuW00W6xnTOQYADi1JADAu1YAKOMYADg5YjE0NzZkLTE2ZmYtNGI4MS05Zj
EwLTE5MDE3ZTkwyJU1MgBiWwwMAAAAZM1kzSEAAABKSU50QU4tVkl1TVEEAAAAAHAAAAEoASQBO
AE4AQQBOAC0AVgBJAFMAVBABAAAAAABAAAAAABMAAAB0aXA6Ly9KaW5uYW4tVmlzZdGEvAA
==</PropagationToken>
    </OleTxTransaction>
  </s:Header>
  <s:Body>
    <Transaction xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns:x="http://www.w3.org/2001/XMLSchema"
xmlns:d3p3="http://schemas.datacontract.org/2004/07/System.Transactions.Ol
etx" z:FactoryType="d3p3:OletxTransaction"
xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
xmlns="http://schemas.datacontract.org/2004/07/System.Transactions">
      <OletxTransactionPropagationToken i:type="x:base64Binary"
xmlns="">AQAAAAAAdknHb/v0zMQKhDZHfsF0Y1AAQAQAAAAACEAAAAAMW2bRzFtm00W6xnB

```

```

OMYAHfXuW00W6xnTOQYADi1JADAu1YAKOMYADg5YjEONzZkLTE2ZmYtNGI4MS05ZjEwLTE5MDE
3ZTkwyYjU1MgDriGgMAAAAZM1kzSEAAABKSUS0QU4tVklTVEEAAAAHAHAEEoASQBOAE4AQQBOA
C0AVgBJAFMAVABBAABAAAAABMAAAB0aXA6Ly9Kaw5uYW4tVmlzdGEvAA==</Oletx
TransactionPropagationToken>
</Transaction>
</s:Body>
</s:Envelope>

```

Wsat10.xml (WS-AT 1.0):

```

<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
    <a:Action>
      s:mustUnderstand="1">http://www.artech.com/transactionformat/WSAtomicTrans
      actionOctober2004Protocol</a:Action>
    <CoordinationContext s:mustUnderstand="1"
      xmlns:mstx="http://schemas.microsoft.com/ws/2006/02/transactions"
      xmlns="http://schemas.xmlsoap.org/ws/2004/10/wscoor">
      <wscoor:Identifier
        xmlns:wscoor="http://schemas.xmlsoap.org/ws/2004/10/wscoor">urn:uuid:ff769
        ce4-4cbf-40cc-a843-6477ec174635</wscoor:Identifier>
      <Expires>59392</Expires><CoordinationType>http://schemas.xmlsoap.org/
      ws/2004/10/wsat</CoordinationType>
      <RegistrationService>
        <Address xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">
          https://jinnan-pc/WsatService/Registration/Coordinator//</Address>
        <ReferenceParameters xmlns="http://schemas.xmlsoap.org/ws/2004/08/
          addressing">
          <mstx:RegisterInfo>
            <mstx:LocalTransactionId>ff769ce4-4cbf-40cc-a843-6477ec174635</mstx:
              LocalTransactionId>
          </mstx:RegisterInfo>
        </ReferenceParameters>
      </RegistrationService>
      <mstx:IsolationLevel>0</mstx:IsolationLevel><mstx:LocalTransactionId>
        ff769ce4-4cbf-40cc-a843-6477ec174635</mstx:LocalTransactionId>
      <PropagationToken
        xmlns="http://schemas.microsoft.com/ws/2006/02/tx/oletx">AQAAAAAAMADknHb/v
        0zMqKhDZHfSF0Y1AAQAAAAACEAAAAAMW2bRzFtm00W6xnBOMYAHfXuW00W6xnTOQYADi1JAD
        Au1YAKOMYADg5YjEONzZkLTE2ZmYtNGI4MS05ZjEwLTE5MDE3ZTkwyYjU1MgDtiGgMAAAAZM1kz
        SEAAABKSUS0QU4tVklTVEEAAAAHAHAEEoASQBOAE4AQQBOAC0AVgBJAFMAVABBAABAAAAABAA
        AAAAAABMAAAB0aXA6Ly9Kaw5uYW4tVmlzdGEvAA==</PropagationToken>
      </CoordinationContext>
    </s:Header>
    <s:Body>
      <Transaction xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:x="http://www.w3.org/2001/XMLSchema"
        xmlns:d3p3="http://schemas.datacontract.org/2004/07/System.Transactions.Ol
        etx" z:FactoryType="d3p3:OletxTransaction"
        xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
        xmlns="http://schemas.datacontract.org/2004/07/System.Transactions">
        <OletxTransactionPropagationToken i:type="x:base64Binary"
          xmlns="x" >AQAAAAAAMADknHb/v0zMqKhDZHfSF0Y1AAQAAAAACEAAAAAMW2bRzFtm00W6xnB
          OMYAHfXuW00W6xnTOQYADi1JADAu1YAKOMYADg5YjEONzZkLTE2ZmYtNGI4MS05ZjEwLTE5MDE
          3ZTkwyYjU1MgAAAAAAMAAAZM1kzSEAAABKSUS0QU4tVklTVEEAJvWAAAAHAHAEEoASQBOAE4AQQBOA
          C0AVgBJAFMAVABBAABAA//8BAAAAAABMAAAB0aXA6Ly9Kaw5uYW4tVmlzdGEvAA==</Oletx
          TransactionPropagationToken>

```

```

    </Transaction>
  </s:Body>
</s:Envelope>

```

Wsat11.xml(WS-AT 1.1):

```

<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
  xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
    <a:Action
      s:mustUnderstand="1">http://www.artech.com/transactionformat/WSAtomicTrans
      action11Protocol</a:Action>
    <CoordinationContext s:mustUnderstand="1"
      xmlns:mstx="http://schemas.microsoft.com/ws/2006/02/transactions"
      xmlns="http://docs.oasis-open.org/ws-tx/wscoor/2006/06">
      <wscoor:Identifier
        xmlns:wscoor="http://docs.oasis-open.org/ws-tx/wscoor/2006/06">urn:uuid:ff
        769ce4-4cbf-40cc-a843-6477ec174635</wscoor:Identifier>
      <Expires>59392</Expires><CoordinationType>
        http://docs.oasis-open.org/ws-tx/wsac/2006/06</CoordinationType>
      <RegistrationService>
        <a:Address>https://jinnan-pc/WsatService/Registration/Coordinator11
        </a:Address>
        <a:ReferenceParameters>
          <mstx:RegisterInfo>
            <mstx:LocalTransactionId>ff769ce4-4cbf-40cc-a843-6477ec174635
            </mstx:LocalTransactionId>
          </mstx:RegisterInfo>
        </a:ReferenceParameters>
      </RegistrationService>
      <mstx:IsolationLevel>0</mstx:IsolationLevel>
      <mstx:LocalTransactionId>ff769ce4-4cbf-40cc-a843-6477ec174635
      </mstx:LocalTransactionId>
      <PropagationToken
        xmlns="http://schemas.microsoft.com/ws/2006/02/tx/oletx">AQAAAAAAADknHb/v
        0zMQKhDZHfsF0Y1AAQAQAAAAACEAAAAAMW2bRzFtm00W6xnBOMYAHfXuW00W6xnTOQYADi1JAD
        Au1YakOMYADg5YjE0NzZkLTE2ZmYtNGI4MS05ZjEwLTE5MDE3ZTkWYjU1MgAAMAAMAAAZM1kz
        SEAAABKSU5OQU4tVkl1TVEEAAGWAHAAAAEoASQBOAE4AQQBOAC0AVgBJAFMAVABBAABgABAAA
        AAAAAABMAAB0aXA6Ly9KaW5uYW4tVmlzdGEvAA==</PropagationToken>
      </CoordinationContext>
    </s:Header>
    <s:Body>
      <Transaction xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:x="http://www.w3.org/2001/XMLSchema"
        xmlns:d3p3="http://schemas.datacontract.org/2004/07/System.Transactions.Ol
        etx" z:FactoryType="d3p3:OletxTransaction"
        xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
        xmlns="http://schemas.datacontract.org/2004/07/System.Transactions">
        <OletxTransactionPropagationToken i:type="x:base64Binary"
          xmlns=" " >AQAAAAAAADknHb/v0zMQKhDZHfsF0Y1AAQAQAAAAACEAAAAAMW2bRzFtm00W6xnB
          OMYAHfXuW00W6xnTOQYADi1JADAu1YakOMYADg5YjE0NzZkLTE2ZmYtNGI4MS05ZjEwLTE5MDE
          3ZTkWYjU1MgDRIGMAAAAZM1kzSEAAABKSU5OQU4tVkl1TVEEAHAAAAHAAAAEoASQBOAE4AQQBOA
          C0AVgBJAFMAVABBAABAAAAABMAAB0aXA6Ly9KaW5uYW4tVmlzdGEvAA==</Oletx
          TransactionPropagationToken>
        </Transaction>
      </s:Body>
    </s:Envelope>

```


3.5.3 事务的自动登记 (Enlistment)

TransactionFlow 选项通过 TransactionFlowAttribute 这个操作契约写入绑定上下文, 由事务绑定创建的事务信道获取该选项并以此作为是否对事务实施传播(发送或者接收)的依据。客户端事务信道通过 TransactionFormatter 对当前事务按照指定的事务处理协议进行格式化, 并嵌入出栈消息。服务端则通过 TransactionFormatter 从入栈消息中提取相应的数据重建事务。这就是事务流转实现的本质。但是整个 WCF 事务还有一个重要的步骤需要实现, 那就是如何将通过 OperationBehaviorAttribute 特性标记为 TransactionRequired 的操作的执行自动纳入流入的事务之中?

被格式化的事务最终是作为一个消息报头 (MessageHeader) 的形式被传输的。服务端事务信道接收到包含有流入事务的消息后, 按照指定的协议从相应的报头中获取将被格式化的事务, 并通过 TransactionFormatter 对事务进行重新创建。被重新创建的事务对象最终以消息属性 (MessageProperty) 的形式重新放入入栈消息。

这样一个包含有事务对象的消息属性定义在一个类型为 System.ServiceModel.Channels.TransactionMessageProperty 的对象之中, 具有如下的定义。只读属性 Transaction 获取内嵌于消息属性对象的事务, 而静态方法 Set 则将事务作为消息属性植入指定的消息。该消息属性在消息中的 Key 为 “TransactionMessageProperty”, 即类型的名称。

```
public sealed class TransactionMessageProperty
{
    //其他成员
    public static void Set(Transaction transaction, Message message);
    public Transaction Transaction { get; }
}
```

WCF 运行时根据消息的 Action 报头定位到相应的操作, 如果该操作应用了 OperationBehaviorAttribute 特性并将 TransactionRequired 属性设为 True, 会进行如下的操作。

- 如果入栈消息中包含事务消息属性, 则提取事务并基于该事务创建 TransactionScope 对象。TransactionScope 对象的其他一些属性, 比如超时时限、隔离级别等, 采用通过服务行为指定的值。结合前面对 System.Transactions 事务的介绍, 该过程的本质就是创建流入事务的依赖事务, 并将创建的依赖事务作为当前的环境事务。
- 如果入栈消息不存在事务属性, 则创建一个新的 TransactionScope 对象, 相当于创建一个可提交事务, 并将其作为当前的环境事务。

上面的过程是在操作方法被调用之前完成的, 并且和操作方法处于相同的线程中。环境事务的存在确保操作方法的执行被纳入流入的事务或一个全新的事务之中。至于事务参与者之间的协调问题, 已经不属于 WCF 体系管辖的范围了, DTC 会接收余下的工作。如果将上面的实现通过代码的形式写出来, 相信读者会理解得更加深刻。

```
[OperationBehavior(TransactionScopeRequired = true)]
public void Transfer(string fromAccountId, string toAccountId, double amount)
{
    //转账操作
}
```

对于上面这段代码, 现在我们将 `OperationBehaviorAttribute` 特性从 `Transfer` 方法中拿掉, 通过自己的方式实现事务的自动登记。如果不考虑超时时限和隔离级别等问题, 整个实现会如下面的代码所示。

```
public void Transfer(string accountFrom, string accountTo, double amount)
{
    TransactionScope transactionScope = null;
    if (OperationContext.Current.IncomingMessageProperties.ContainsKey(
        "TransactionMessageProperty"))
    {
        TransactionMessageProperty transactionMessageProperty =
            (TransactionMessageProperty)OperationContext.Current.IncomingMessageProperties["TransactionMessageProperty"];
        transactionScope = new
            TransactionScope(transactionMessageProperty.Transaction);
    }
    else
    {
        transactionScope = new TransactionScope();
    }
    try
    {
        //转账操作
        transactionScope.Complete();
    }
    finally
    {
        transactionScope.Dispose();
    }
}
```

3.5.4 OleTx 提升 (OleTx Upgrade) 机制

在 3.4 节的实例演示中我们谈到, 即使将绑定采用的事务处理协议设置成 `WS-AT`, 并且在 `DTC` 中对 `WS-AT` 进行了正确的设置, `WCF` 运行时仍有可能采用 `OleTx` 协议进行事务处理, 这就涉及 `OleTx` 提升的话题。

`OleTx` 是 Windows 平台下默认的分布式事务协议, 它采用安全 `RPC` (`Secure RPC`, `SRPC`) 协议进行通信, 并采用二进制编码, 具有最好的性能优势。对于 `WCF` 事务来说, 即使我们显式地将 `WS-AT` 设置成绑定采用的事务协议, 如果 `DTC` 发现当前的事务应用场景仍然能够采用 `OleTx` 进行处理, 会自动将 `WS-AT` 协议提升到 `OleTx` 协议, 这就是 `OleTx` 提升机制。

将视线再次移向上面基于 `TransactionFormatter` 的例子, 通过分析包含有格式化事务

数据的三种基于不同事务协议的 SOAP 消息的结构，我们会发现基于 OleTx 的所有信息均包含在基于 WS-AT 的消息之中。实际上 OleTx 需要的仅仅是事务的传播令牌 (Propagation Token)。也就是说，对于 WS-AT 协调上下文接收方的 DTC 是可以通过 OleTx 进行事务处理的。

在默认的情况下，OleTx 提升机制自动生效。可以通过修改相应的注册表项对 OleTx 提升进行开启和关闭，该注册表项就是上面提到的 HKLM\SOFTWARE\Microsoft\WSAT\3.0\OleTxUpgradeEnabled。接下来将介绍如图 3-25 所示的 4 种较为典型的 OleTx 提升机制场景。

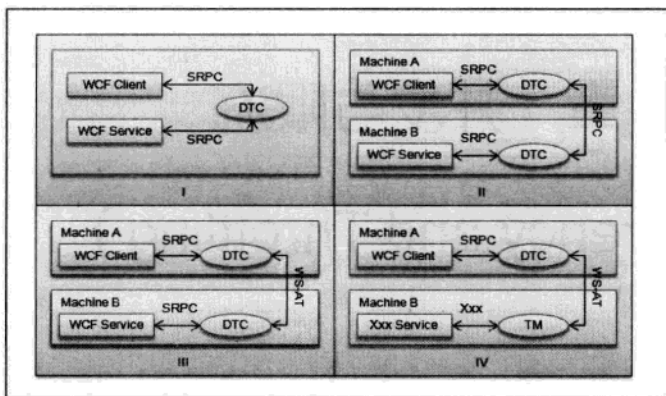


图 3-25 OleTx 提升的 4 个典型应用场景

在这 4 种 OleTx 提升场景中，WCF 客户端应用程序与本地 DTC 之间，服务与本地事务管理器之间，以及客户端本地的 DTC 和服务端本地事务管理器之间按照如下的方式进行通信。

- 场景一：WCF 客户端调用同一台机器上的 WCF 服务，客户端应用程序/服务和 DTC 采用 SRPC 通信。
- 场景二：WCF 客户端调用另一台机器上的 WCF 服务，客户端应用程序/服务与本地 DTC，以及两台机器的 DTC 均采用 SRPC 进行通信。
- 场景三：WCF 客户端调用另一台机器上的 WCF 服务，客户端应用程序/服务与本地 DTC 采用 SRPC 进行通信，而两台机器的 DTC 之间采用 WS-AT 进行通信。
- 场景四：WCF 客户端调用另一台机器上的非 WCF 服务（比如 Java 平台 Web 服务），客户端应用程序与本地 DTC 采用 SRPC 进行通信，客户端本地 DTC 和服务端事务管理器 (TM) 采用 WS-AT 进行通信。

第4章 并发与限流

(Concurrency and Throttling)

WCF 服务寄宿于资源有限的环境中，要实现服务性能的最大化，需要考虑如何利用现有的资源实现最大的吞吐量 (Throughput)。提高吞吐量就某个寄宿的服务实例 (Service Instance) 来说，一个重要的途径就是让它能够同时处理来自多个客户端 (服务代理) 的并发访问。但是资源的有限性决定了并发量有一个最大值，如果 WCF 不进行流量控制，那么一旦超过这个临界值，整个服务端将会由于资源耗尽而崩溃。

资源有限
并发访问
PDG

软件架构是一门权衡的艺术，需要综合考虑各种相互矛盾的因素，找到一种最优的组合方式。提高单个服务实例允许的并发访问量能够提高整体吞吐量，这样的理论依赖于一种假设，那就是服务端所能使用的资源是无限的。我们知道，这种假设无论在什么情况下都不会成立。如果并发量超出了服务端所能承受的临界点，整个服务端将会崩溃。WCF 一方面需要允许单个服务实例并发处理接收到的多个请求，同时也需要设置一道闸门控制并发的数量。WCF 的流量限制（Throttling）体系为我们创建了这道闸门。

4.1 并发与实例上下文模式

并发的含义就是多个并行的操作同时作用于一个相同的资源或对象，或者说同一个资源或对象同时应付多个并行的请求。对于 WCF 的并发来说，这里的“资源或者对象”指的就是承载服务操作最终执行的服务实例。WCF 将服务实例封装在一个称为实例上下文（Instance Context）的对象中，所以 WCF 中的并发指的是同一个服务实例上下文同时处理多个服务调用请求。

4.1.1 同一个服务实例上下文同时处理多个服务调用请求

WCF 服务端框架的一个主要任务是将接收到的服务调用请求分发给激活的服务实例，并调用相应的服务操作返回执行结果。也就是说服务操作的执行最终还是会落实到某个具体的服务实例上。

上册的第 9 章“实例化与会话（Instancing and Session）”对 WCF 的实例化机制进行了深入的剖析。在 WCF 服务端框架体系中，激活的服务实例并不是单独存在的，而是被封装在一个被称为实例上下文的对象中。WCF 提供了三种不同的实例上下文模式（Per-Call、Per-Session 和 Single），实现了不同的服务实例上下文提供机制。

WCF 并发框架解决的是如何有效地处理被分发到同一个服务实例上下文的多个服务调用请求。这些并行的调用请求可能来自不同的客户端（服务代理），也可能来自相同的客户端。WCF 并发的本质可以通过图 4-1 体现。

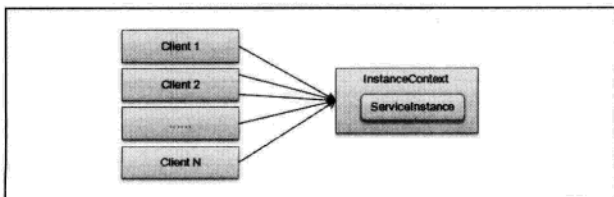


图 4-1 通过一个 InstanceContext 对多个并发请求的处理

由于 WCF 的并发处理属于服务自身的行为，因此我们通过服务行为（Service Behavior）的形式对采取的并发策略进行控制，而不同的并发策略对应着相应的并发模式（Concurrency Mode）。

1. 通过 ServiceBehaviorAttribute 特性定义并发模式

WCF 为三种典型的并发处理策略定义了 Single、Reentrant 和 Multiple 三种典型的并发模式。并发模式通过具有如下定义的 System.ServiceModel.ConcurrencyMode 枚举表示，而上述三种并发模式对应同名的枚举项。

```
public enum ConcurrencyMode
{
    Single,
    Reentrant,
    Multiple
}
```

通过 ConcurrencyMode 枚举项表示的三种不同的并发模式体现了 WCF 处理并发请求的三种策略。

- **Single:** 一个实例上下文在某个时刻只能用于对单一请求的处理，或者说针对某个实例上下文的多个并发的请求会以一种串行的方式进行处理。
- **Reentrant:** 一个实例上下文对象在某个时刻只能用于对单一请求的处理。如果服务操作在执行过程中涉及对客户端的回调 (Callback)，该实例上下文可以用于其他服务调用请求的处理。如果回调操作执行后服务实例上下文没有用于其他请求的处理，回调后的操作能够得到处理。
- **Multiple:** 一个实例上下文可以同时用于处理多个服务请求。

如果采用 Single 和 Reentrant 模式，当 WCF 服务端接收到多个针对相同实例上下文的请求时，会先确定该实例上下文是否可用（是否正在处理之前的抵达服务调用请求）。如果可用，则将接收到的第一个请求分发给它，其他请求则根据抵达的先后顺序被放入一个队列中。如果之前的请求被正常处理，队列中的第一个请求被分发给实例上下文。如果一个请求在队列中等待的时间过长，超过了设置的服务调用的超时时限，客户端会抛出 TimeoutException 异常。

并发模式的采用是服务单边的选择，是服务个人的行为，所以并发模式以服务行为的方式定义。只需要在服务类型上应用 ServiceBehaviorAttribute 特性为其 ConcurrencyMode 属性设置相应的值即可。ConcurrencyMode 属性在 ServiceBehaviorAttribute 类型中具有如下的定义。

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public ConcurrencyMode ConcurrencyMode { get; set; }
}
```

如果没有显式指定服务采用的并发模式，默认使用的是 ConcurrencyMode.Single。下面两种服务定义方式是等效的。

```
public class CalculatorService : ICalculator
{
    //省略成员
}

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Single)]
public class CalculatorService : ICalculator
{
    //省略成员
}
```

2. 回调 (Callback) 中的并发

WCF 并发解决的是同一个实例上下文在处理并发请求时采用怎样的处理策略。实例上下文不仅仅是封装真正服务实例的容器,当服务回调客户端的时候,回调对象也是封装在实例上下文中的。我们将用于封装服务实例和回调对象的实例上下文分别称为服务实例上下文和回调实例上下文。

在双向通信的场景中,如果在进行多次服务调用的时候指定相同的回调实例上下文,就有可能出现服务并发地回调同一个回调上下文的情况。不论是多个服务请求并发调用同一个服务实例上下文,还是多个服务操作并发回调同一个回调实例上下文,WCF 都采用相同的并发机制。

回调采用的并发模式通过应用在回调类型上的 `CallbackBehaviorAttribute` 特性来指定。如下面的代码所示, `CallbackBehaviorAttribute` 中同样定义了 `ConcurrencyMode` 属性。

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class CallbackBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public ConcurrencyMode ConcurrencyMode { get; set; }
}
```

在下面的代码中,我们通过在回调类型 `CalculatorCallbackService` 上应用 `CallbackBehaviorAttribute` 特性,将回调并发模式设置成 `ConcurrencyMode.Multiple`。

```
[CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
public class CalculatorCallbackService: ICalculatorCallback
{
    //省略成员
}
```

3. 事务行为与并发

在本书的第3章“事务 (Transaction)”中,我们提到可以在服务类型上面应用 `ServiceBehaviorAttribute` 特性并将 `ReleaseServiceInstanceOnTransactionComplete` 属性设成 `True`,让 WCF 在事务结束之后将服务实例上下文释放掉。不过这样的设置只有在并发模式为 `Single` 的前提下才有效。

```

[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete = true,
ConcurrencyMode = ConcurrencyMode.Multiple)]
public class BankingService : IBankingService
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void Transfer(string accountFrom, string accountTo, decimal amount)
    {
        //省略实现
    }
}

```

在上面的代码中定义了一个采用 `Multiple` 并发模式的 `BankingService` 服务，并通过 `ServiceBehaviorAttribute` 特性指定 `ReleaseServiceInstanceOnTransactionComplete` 为 `True`。当我们试图寄宿该服务的时候，会抛出如图 4-2 所示的 `InvalidOperationException` 异常，提示“服务已配置为将 `ReleaseServiceInstanceOnTransactionComplete` 设置成 `True`，但 `ConcurrencyMode` 未设置为 `Single`，`ReleaseServiceInstanceOnTransactionComplete` 需要使用 `ConcurrencyMode.Single`”。



图 4-2 `Multiple+ReleaseServiceInstanceOnTransactionComplete` 导致的异常

如果通过 `ServiceBehaviorAttribute` 特性将 `ReleaseServiceInstanceOnTransactionComplete` 属性设置为 `True`，意味着当事务被提交之后当前的实例上下文会被释放。如果当前的并发模式不是 `Single`，当前的实例上下文可能正在处理其他调用请求。对其进行贸然的释放自然是不允许的。

4.1.2 并发中的同步

WCF 提供的三种不同的并发模式，使开发者可以根据具体的情况选择不同的并发处理策略。对于这三种并发模式，`Multiple` 采用并行的执行方式，而 `Single` 和 `Reentrant` 则采用串行的执行方式。串行执行即同步执行，在 WCF 并发框架体系中同步机制是如何实现的呢？

1. `ConcurrencyMode.Single` 模式下的同步实现

WCF 并发框架体系下针对 `Single` 模式的实现非常简单，其本质就是对实例上下文进行加锁。如果采用反编译工具查看 `InstanceContext` 的定义，我们会发现它定义了如下一个类型

为 `System.Object` 的内部属性 `ThisLock`。WCF 就是通过对 `InstanceContext` 的 `ThisLock` 进行加锁，确保了对实例上下文同步的访问。

```
public sealed class InstanceContext : CommunicationObject,
    IExtensibleObject<InstanceContext>
{
    //其他成员
    internal object ThisLock
    {
        get { return base.ThisLock; }
    }
}
public abstract class CommunicationObject : ICommunicationObject
{
    //其他成员
    protected object ThisLock { get; }
}
```

WCF 服务端运行时在处理服务调用消息请求之后，利用实例上下文提供者（`InstanceContextProvider`）创建新的或者获取现有的实例上下文，然后将请求消息分发给该实例上下文对消息进行进一步处理。在处理操作执行之前，如果发现相应的服务采用 `Single` 并发模式，只有在获取了针对实例上下文的 `ThisLock` 属性对象的锁的情况下才会进行后续的操作。这样就保证了单一的实例上下文在 `Single` 并发模式下永远是以同步的方式被调用的。

2. `ConcurrencyMode.Reentrant` 模式下的同步实现

在 `Single` 并发模式下，实例上下文对某个调用请求进行处理的整个过程都是被锁定的。如果在服务操作执行过程中涉及对客户端的回调，并且回调操作采用请求/回复消息交换模式，当服务端运行时接收到从客户端返回的回调回复后，会试图再次获取服务实例上下文以进行后续的处理。在这种情况下服务端会试图再次对实例上下文进行加锁，但是实例上下文之前已经被锁住一直不曾释放，这就会导致尴尬情形出现：回调之前被锁住的实例上下文只有在整个服务操作（包括回调前、回调本身和回调后的操作）执行完成之后才会释放，但是回调后无法获得实例上下文的锁导致服务操作永远也不可能完成，这无疑就造成了死锁。

如果在服务操作执行过程中需要对客户端实施回调，要么采用单向（`One-way`）的方式进行回调，要么将服务的并发模式设置成 `Reentrant` 或 `Multiple`。否则，如图 4-3 所示的 `InvalidOperationException` 异常会在进行回调的时候抛出。从异常消息可以看出，VS 的汉化真的不敢恭维，如果要正常理解异常消息的含义，你需要知道这里的“邮件”、“可重输入”和“多个”是依次对“`Message`”、“`Reentrant`”和“`Multiple`”的翻译。

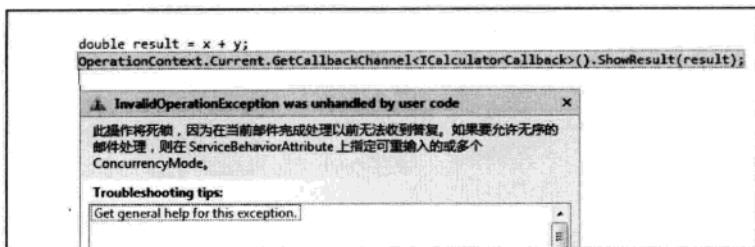


图 4-3 在 Single 模式执行回调导致的异常

如果真的需要在服务操作过程中实施基于请求/回复模式的回调,毫无疑问采用 Multiple 并发模式可以解决死锁的问题,因为 Multiple 模式根本就不存在对 InstanceContext 加锁的问题。那么在 Reentrant 模式下, WCF 并发框架体系又是如何解决这个问题的呢?

Reentrant, 翻译成汉语就是“重入”,意思是服务操作过程中完成了对外调用(Call Out)还能重新回到相应的位置继续执行。同 Single 模式一样, WCF 运行时将调用请求消息分发给相应的实例上下文之前,会先对其加锁。但是在开始实施回调之前,对实例上下文的锁定会被解除,当回调返回后再对其加锁。

对于 Reentrant 有一点需要特别说明,当服务端进行回调时,由于加在实例上下文上的锁会被释放,意味着其他服务调用请求可以被分发给该实例上下文。当回调返回的时候,如果实例上下文正用于处理另一个服务调用请求,会等待实例上下文被释放。如果等待的时间超过设定的超时时限,客户端会抛出 TimeoutException 异常。

4.1.3 并发与实例上下文模式

由于 WCF 的并发是针对实例上下文而言的,因此在不同的实例上下文模式下,会表现出不同的并发行为。接下来从具体的实例上下文模式的角度来剖析 WCF 的并发。如果对 WCF 实例上下文模式和实例上下文提供机制不了解,请参阅上册一书的第 9 章“实例化与会话(Instancing and Session)”。

为了使读者对采用不同实例上下文对并发的影响有一个深刻的认识,我会结合一个简单的例子来讲述本节的内容。我创建一个简单的 WCF 应用,并在此基础上添加监控功能,主要监控各种事件的执行时间,比如客户端服务调用的开始和结束时间,服务操作开始执行和结束执行的时间等。读者可以根据实时输出的监控信息,对 WCF 的并发处理情况有一个很直观的认识。

实例演示: 创建 WCF 监控程序 (S401)

本实例依然采用之前多次采用的三层结构,即契约、服务(定义服务类型和寄宿服务)和客户端。为了以可视化的形式实时输出监控信息,对于客户端和服务程序均采用 Windows Forms 程序。我们依然以计算服务作为例子,下面是服务契约的定义。

```
using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface ICalculator
    {
        [OperationContract]
        double Add(double x, double y);
    }
}
```

由于需要监控各种事件的时间，因此定义了一个名为 `EventType` 的枚举表示不同的事件类型。8 个枚举值分别表示开始和结束服务调用（客户端）、开始和结束服务操作执行（服务端）、开始和结束回调（服务端），以及开始和结束回调操作的执行（客户端）。

```
namespace Artech.WcfServices.Service.Interface
{
    public enum EventType
    {
        StartCall,
        EndCall,
        StartExecute,
        EndExecute,
        StartCallback,
        EndCallback,
        StartExecuteCallback,
        EndExecuteCallback
    }
}
```

然后定义了如下一个 `EventMonitor` 的静态类，该类通过两个重载的 `Send` 方法以事件触发的形式发送事件（上述 8 种类型的事件）通知。我定义了专门的事件参数类型 `MonitorEventArgs`，封装客户端 ID、事件类型和触发时间。`Send` 具有两个重载，一个具有用整数表示的客户端 ID 用于客户端，可以显式指定客户端 ID。后者需要从客户端手工添加的消息报头提取客户端 ID，该消息报头的名称和命名空间通过两个常量定义。

```
using System;
using System.ServiceModel;
using System.Threading;
namespace Artech.WcfServices.Service.Interface
{
    public static class EventMonitor
    {
        public const string ClientIdHeaderNamespace = "http://www.artech.com/";
        public const string ClientIdHeaderLocalName = "ClientId";
        public static EventHandler<MonitorEventArgs> MonitoringNotificationSended;

        public static void Send(EventType eventType)
        {
            if (null != MonitoringNotificationSended)
            {
                int clientId =
                    OperationContext.Current.IncomingMessageHeaders.GetHeader<int>(
                        ClientIdHeaderLocalName, ClientIdHeaderNamespace);
                MonitoringNotificationSended(null, new MonitorEventArgs(clientId,
```

```

        eventType, DateTime.Now));
    }
}
public static void Send(int clientId, EventType eventType)
{
    if (null != MonitoringNotificationSended)
    {
        MonitoringNotificationSended(null, new MonitorEventArgs (clientId,
            eventType, DateTime.Now));
    }
}
}
public class MonitorEventArgs : EventArgs
{
    public int ClientId { get; private set; }
    public EventType EventType { get; private set; }
    public DateTime EventTime { get; private set; }
    public MonitorEventArgs(int clientId, EventType eventType, DateTime
        eventTime)
    {
        this.ClientId = clientId;
        this.EventType = eventType;
        this.EventTime = eventTime;
    }
}
}
}

```

EventMonitor 的 Send 方法直接用在具有如下定义的 CalculatorService 的 Add 操作方法中，实时输出操作方法开始和结束执行的时间，以及当前处理的客户端的 ID。我们通过 ServiceBehaviorAttribute 特性将 UseSynchronizationContext 属性设置成 False，至于为什么要这么做，是本章后续部分需要讲述的内容。服务操作 Add 通过将当前线程挂起 5 秒钟，用于模拟一个相对耗时的操作，便于我们更好地通过监控输出的时间分析并发处理的情况。

```

using System.ServiceModel;
using System.Threading;
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Service
{
    [ServiceBehavior(UseSynchronizationContext = false)]
    public class CalculatorService : ICalculator
    {
        public double Add(double x, double y)
        {
            EventMonitor.Send(EventType.StartExecute);
            Thread.Sleep(5000);
            double result = x + y;
            EventMonitor.Send(EventType.EndExecute);
            return result;
        }
    }
}

```

然后，我们在一个 Windows Form 应用中对上面创建的 CalculatorService 进行寄宿，并将该应用作为服务端的监控器。在这个应用中，我只添加了如图 4-4 所示的简单的窗体，整个窗体仅有唯一一个 ListBox 控件，在运行的时候相应的监控信息就实时地逐条追加到该

ListBox 之中。

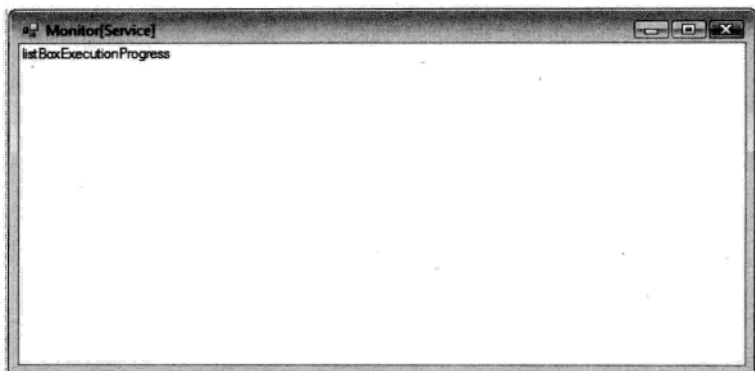


图 4-4 服务端监控窗体设计界面

我们通过注册 EventMonitor 的静态 MonitoringNotificationSended 事件实时输出服务端监控信息。同时对 CalculatorService 的寄宿实现在监控窗体的 Load 事件中。整个窗体后台代码和配置文件如下所示，需要注意的是我们采用不支持会话的 BasicHttpBinding。

MonitorForm 后台代码：

```
using System;
using System.ServiceModel;
using System.Threading;
using System.Windows.Forms;
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Service
{
    public partial class MonitorForm : Form
    {
        private SynchronizationContext _syncContext;
        private ServiceHost _serviceHost;
        public MonitorForm()
        {
            InitializeComponent();
        }

        private void MonitorForm_Load(object sender, EventArgs e)
        {
            string header = string.Format("{0, -13}{1, -22}{2}", "Client", "Time",
                "Event");
            this.listBoxExecutionProgress.Items.Add(header);
            _syncContext = SynchronizationContext.Current;
            EventMonitor.MonitoringNotificationSended +=
                ReceiveMonitoringNotification;
            this.Disposed += delegate
            {
                EventMonitor.MonitoringNotificationSended -=
                    ReceiveMonitoringNotification;
                _serviceHost.Close();
            };
            _serviceHost = new ServiceHost(typeof(CalculatorService));
        }
    }
}
```

```

        _serviceHost.Open();
    }
    public void ReceiveMonitoringNotification(object sender, MonitorEventArgs
        args)
    {
        string message = string.Format("{0, -15}{1, -20}{2}", args.ClientId,
            args.EventTime.ToLongTimeString(), args.EventType);
        _syncContext.Post(state =>
            this.listBoxExecutionProgress.Items.Add(message), null);
    }
}
}

```

配置:

```

<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService">
        <endpoint address="http://127.0.0.1:3721/calculatorservice"
            binding="basicHttpBinding"
            contract="Artech.WcfServices.Service.Interface.ICalculator"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

最后编写作为客户端程序的 Windows Form 应用。该应用既作为 CalculatorService 的客户端程序而存在,也是客户端的监控器。整个应用具有一个与图 4-4 所示一样的窗体。同样以注册 EventMonitor 的静态 MonitoringNotificationSended 事件的形式实时输出客户端监控信息。在监控窗体的 Load 事件中,利用 ThreadPool 创建 5 个服务代理以并发的形式进行服务调用。这 5 个服务代理对象对应的客户端 ID 分别为从 1 到 5,并通过消息报头的形式发送到服务端。整个监控窗体的代码如下所示,相应的配置就不再列出来了。

```

using System;
using System.ServiceModel;
using System.Threading;
using System.Windows.Forms;
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Client
{
    public partial class MonitorForm : Form
    {
        private SynchronizationContext _syncContext;
        private ChannelFactory<ICalculator> _channelFactory;
        private static int clientIdIndex = 0;
        public MonitorForm()
        {
            InitializeComponent();
        }
        private void MonitorForm_Load(object sender, EventArgs e)
        {
            string header = string.Format("{0, -13}{1, -22}{2}", "Client", "Time",
                "Event");
            this.listBoxExecutionProgress.Items.Add(header);
            _syncContext = SynchronizationContext.Current;
        }
    }
}

```

```

_channelFactory = new ChannelFactory<ICalculator>("calculatorservice");

EventMonitor.MonitoringNotificationSended +=
    ReceiveMonitoringNotification;
this.Disposed += delegate
{
    EventMonitor.MonitoringNotificationSended -=
        ReceiveMonitoringNotification;
    _channelFactory.Close();
};

for (int i = 1; i <= 5; i++)
{
    ThreadPool.QueueUserWorkItem(state =>
    {
        int clientId = Interlocked.Increment(ref clientIdIndex);
        ICalculator proxy = _channelFactory.CreateChannel();
        using (proxy as IDisposable)
        {
            EventMonitor.Send(clientId, EventType.StartCall);
            using (OperationContextScope contextScope = new
                OperationContextScope(proxy as IContextChannel))
            {
                MessageHeader<int> messageHeader = new MessageHeader
                    <int>(clientId);
                OperationContext.Current.OutgoingMessageHeaders.Add(messageHeader.GetUntypedHeader(EventMonitor.CientIdHeaderLocalName, EventMonitor.CientIdHeader
                    Namespace)); proxy.Add(1, 2);
            }
            EventMonitor.Send(clientId, EventType.EndCall);
        }
    }, null);
}

}

public void ReceiveMonitoringNotification(object sender, MonitorEventArgs
    args)
{
    string message = string.Format("{0, -15}{1, -20}{2}", args.ClientId,
        args.EventTime.ToLongTimeString(), args.EventType);
    _syncContext.Post(state =>
        this.listBoxExecutionProgress.Items.Add(message), null);
}
}
}

```

到此为止我们的监控程序就完成了，接下来就以此为基础看看在不同实例上下文模式下，WCF 服务端框架是如何处理并发的。

1. 单调 (PerCall) 实例上下文模式

由于 WCF 的并发是针对某个实例上下文而言的，对单调的实例上下文模式，不管该请求是否来自相同的客户端，WCF 服务端运行时总是创建一个全新的实例上下文来处理每一个请求，因此在单调实例上下文模式下，根本就不存在对某个实例上下文的并发调用的情况发生。

可以通过我们的监控程序来验证这一点。只需要通过下面的 `ServiceBehaviorAttribute` 特性将实例上下文模式设置成 `InstanceContextMode.PerCall`。(S402)

```
[ServiceBehavior(UseSynchronizationContext = false,
                 InstanceContextMode = InstanceContextMode.PerCall)]
public class CalculatorService : ICalculator
{
    //省略成员
}
```

如果在此基础上运行监控程序,将会得到如图 4-5 所示的输出结果。从中可以看出,虽然采用默认的并发模式 (Single),来自 5 个不同客户端 (服务代理) 的调用请求能够及时地得到处理。

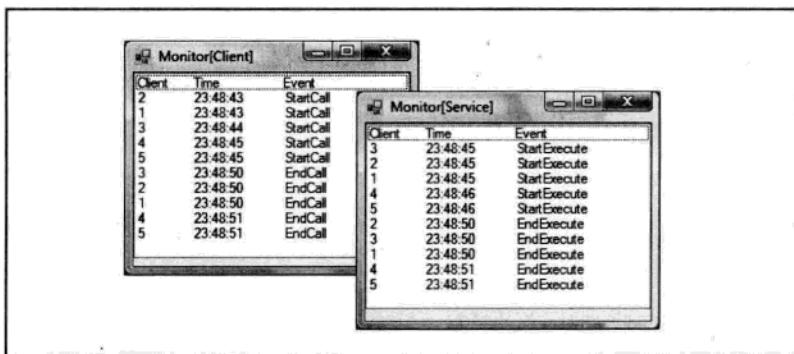


图 4-5 单调实例上下文模式下的并发事件监控输出 (不同客户端)

上面演示了 WCF 服务端处理来自不同客户端并发请求的处理。如果 5 个请求来自相同的客户端,它们是否还能够及时地得到处理呢?不妨通过我们的监控程序来说话。现在需要做的是修改客户端进行服务调用的方式,让 5 个并发的调用来自于相同的服务代理对象,相关的代码如下所示。为了便于跟踪,依然将并发的序号 1~5 通过消息报头传递到服务端。不过在这里它不代表客户端,而是代表某个服务调用。(S403)

```
ICalculator proxy = _channelFactory.CreateChannel();
for (int i = 1; i < 6; i++)
{
    ThreadPool.QueueUserWorkItem(state =>
    {
        int clientId = Interlocked.Increment(ref clientIdIndex);
        EventMonitor.Send(clientId, EventType.StartCall);
        using (OperationContextScope contextScope = new OperationContextScope(
            proxy as IContextChannel))
        {
            MessageHeader<int> messageHeader = new MessageHeader<int>(clientId);
            OperationContext.Current.OutgoingMessageHeaders.Add(
                messageHeader.GetUntypedHeader(EventMonitor.CientIdHeaderLocal
                    Name, EventMonitor.CientIdHeaderNamespace));
            proxy.Add(1, 2);
        }
    });
}
```



```

    }
    EventMonitor.Send(clientId, EventType.EndCall);
}, null);
}

```

再次运行我们的监控程序，将会得到完全不一样的输出结果（如图 4-6 所示）。从监控信息可以很清晰地看出，服务操作的执行完全是以串行化的形式执行的。对于服务端来说，似乎仍然是以同步的方式方式处理并发的服务调用请求的。

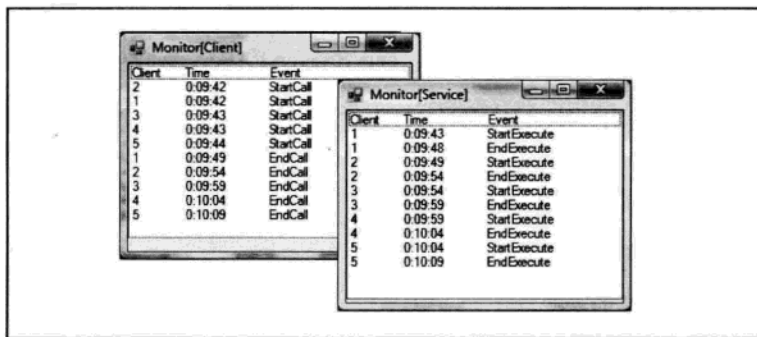


图 4-6 单调实例上下文模式下的并发事件监控输出（相同客户端）

我们说过，WCF 并发机制的同步机制是通过对实例上下文进行加锁实现的。但是对于单调实例上下文模式来说，虽然 5 个请求来自相同的客户端，但是对应的实例上下文却是不同的。难道前面的结论都是错误的吗？

实际上出现如图 4-6 所示的监控输出与 WCF 并发框架体系采用的同步机制一点关系都没有。在说明原因之前，我们先给出解决方案。只需要按照如下的方式在进行服务调用之前调用 `Open` 方法显式地开启服务代理，就会得到与图 4-5 类似的输出结果。（S404）

```

ICalculator proxy = _channelFactory.CreateChannel();
(proxy as ICommunicationObject).Open();
for (int i = 1; i < 6; i++)
{
    //省略其他代码
}

```

上面的问题涉及 WCF 一个很隐晦的机制，相信不会有太多人知道它的存在。如果直接通过创建出来的服务代理对象（并没有显式开启服务代理）进行服务调用，WCF 客户端框架会通过相应的机制确保服务代理的开启，可以将这种机制称为服务代理自动开启。在内部 WCF 实际上是将本次调用放入一个队列之中，等待上一个放入队列的调用结束。也就是说，针对一个没有被显式开启的服务代理的并发调用实际上是以同步或者串行的方式被调用的。

但是如果在进行服务调用之前通过上面代码所示的方式显式地开启服务代理，基于该代理的服务调用就能得到及时处理。所以当真的需要执行基于相同服务代理的并发调用时，请务必对服务代理进行显式开启。

2. 会话 (PerSession) 实例上下文模式

在基于会话的实例上下文模式下，被创建出来的服务实例上下文与会话（客户端或者服务代理）绑定在一起。也就是说，实例上下文和服务代理具有一一对应的关系。基于我们前面介绍的对实例上下文加锁的同步机制，如果服务端接收到的并发调用是基于不同的客户端，那么它们会被分发给不同的实例上下文，所以对它们的处理是并行的。因此我们主要探讨的是针对相同客户端的并发调用的问题。

在上册的第9章“实例化与会话 (Instancing and Session)”中，我们对 WCF 的会话进行过深入的剖析。如果读者对其中的内容还熟悉，一定知道 WCF 的会话最终取决于以下三个方面的因素：

- 服务契约采用 Allowed 或者 Required 的会话模式；
- 服务采用 PerSession 的实例上下文模式；
- 终结点的绑定提供对会话的支持。

即使我们通过 ServiceBehaviorAttribute 特性将服务的实例上下文模式设置成 PerSession，如果不满足其余两个条件，WCF 仍然采用的是基于单调的实例上下文提供机制，那么表现出来的并发处理行为就与单调模式并无二致了。

如果同时满足上述的三个条件，来自于相同客户端的并发请求分发到相同的实例上下文。在这种情况下，WCF 将按照相应并发模式语义上体现的行为来处理这些并发的请求，即 Single 和 Multiple 体现的分别是串行化和并行化的处理方式。

3. 单例 (Single) 实例上下文模式

对于采用单例实例上下文模式，所有的服务调用请求，不论它来自于那个客户端，最终都会被分发给同一个服务实例上下文。毫无疑问，不论并发请求是否来自相同的客户端，真正表现出来的并发行为和指定的并发模式是一致的。表 4-1 体现了针对来自于相同客户端（服务代理）的并发请求，不同的实例上下文模式在 Single 和 Multiple 并发模式下的操作执行是按照并行还是串行的方式执行的。

表 4-1 针对来自相同客户端的并发请求，不同实例上下文执行方式

	Single		Multiple
PerCall	并行		并行
PerSession	SessionMode.Allowed/Required AND Sessionful Binding	串行	并行
	SessionMode.NotAllowed OR Non-Sessionful Binding	并行	
Single	串行		并行

实例演示：并发模式对回调的影响（S405，S406）

在回调场景中，我们将回调对象封装到实例上下文中。当服务操作过程中执行回调操作的时候，回调消息最终也是分发到回调实例上下文中。从消息分发与并发处理的机制来看，这两种请求并没有本质的不同。接下来通过一个实例将两者结合起来，综合分析 WCF 对并发服务调用和并发回调的处理机制。

需要对上面给出的监控程序进行相应的修改。首先需要修改的是服务契约 ICalculator。现在我们通过回调的形式来重写计算服务，所以需要将 Add 的返回类型改成 void，计算结果通过执行回调操作的形式在客户端显示。

```
[ServiceContract(Namespace="http://www.artech.com/",
    CallbackContract = typeof(ICalculatorCallback))]
public interface ICalculator
{
    [OperationContract]
    void Add(double x, double y);
}
```

作为回调契约的 ICalculatorCallback 接口定义如下。它通过 ShowResult 方法将计算结果显示出来。在一般情况下我们会将 Add 和 ShowResult 操作定义为单向（One-way），但是这里我并没有这么做，所以无论是服务操作 Add 还是回调操作 ShowResult，均采用请求/回复消息交换模式。

```
using System.ServiceModel;
namespace Artech.WcfService.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface ICalculatorCallback
    {
        [OperationContract]
        void ShowResult(double result);
    }
}
```

在本例中，CalculatorService 采用单例实例上下文模式（Single），并将并发模式设置成 Reentrant。在 Add 操作中，可以将整个执行过程分成三个阶段，即 PreCallback、Callback 和 PostCallback，而且 PreCallback 和 PostCallback 执行时间为 5 秒。在开始和结束执行 Add 操作，以及开始与结束回调的时候都是通过 EventMonitor 发送相应的事件通知的。修改后的 CalculatorService 如下面的代码所示。

```
[ServiceBehavior(UseSynchronizationContext = false,
    InstanceContextMode = InstanceContextMode.Single,
    ConcurrencyMode = ConcurrencyMode.Reentrant)]
public class CalculatorService : ICalculator
{
    public void Add(double x, double y)
    {
        //PreCallback
        EventMonitor.Send(EventType.StartExecute);
```

```

Thread.Sleep(5000);
double result = x + y;

//Callback
EventMonitor.Send(EventType.StartCallback);
int clientId =
    OperationContext.Current.IncomingMessageHeaders.GetHeader<int>(
        EventMonitor.CientIdHeaderLocalName,
        EventMonitor.CientIdHeaderNamespace);
MessageHeader<int> messageHeader = new MessageHeader<int>(clientId);
OperationContext.Current.OutgoingMessageHeaders.Add(
    messageHeader.GetUntypedHeader(EventMonitor.CientIdHeaderLocalName,
        EventMonitor.CientIdHeaderNamespace));
OperationContext.Current.GetCallbackChannel<ICalculatorCallback>().
    ShowResult(result);
EventMonitor.Send(EventType.EndCallback);

//PostCallback
Thread.Sleep(5000);
EventMonitor.Send(EventType.EndExecute);
}
}

```

对于服务寄宿程序不需要做任何修改,但是需要采用支持双向通信的绑定类型以实现
对回调的支持。在这里采用的是 NetTcpBinding。为了降低安全协商 (Negotiation) 代码的时延,
我特意将绑定的安全模式设置成 None。下面是更新后的服务端配置,客户端需要进行相应
的修改。

```

<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="nonSecureBinding">
          <security mode="None" />
        </binding>
      </netTcpBinding>
    </bindings>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService">
        <endpoint address="net.tcp://127.0.0.1:3721/calculatorservice"
          binding="netTcpBinding"
          bindingConfiguration="nonSecureBinding"
          contract="Artech.WcfServices.Service.Interface.ICalculator" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

由于回调操作在客户端执行,所以客户端需要实现回调契约接口创建回调类型。实现回
调契约接口 ICalculatorCallback 的回调类型定义在如下的 CalculatorCallbackService 类型中。
由于在本例中我们需要的仅仅是监控回调操作执行的时间,并不是真的需要显示出运算的最
终结果,因此仅仅是通过挂起当前线程模拟一个耗时的回调操作 (10 秒),在回调操作开始
和结束执行的时候通过 EventMonitor 发送相应的事件通知。

```

using System.Threading;
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Client
{
    public class CalculatorCallbackService : ICalculatorCallback
    {
        public void ShowResult(double result)
        {
            EventMonitor.Send(EventType.StartExecuteCallback);
            Thread.Sleep(10000);
            EventMonitor.Send(EventType.EndExecuteCallback);
        }
    }
}

```

最后一个步骤是对客户端按照回调的方式进行相应的修改。首先创建 `CalculatorCallbackService` 对象，并以此创建一个 `InstanceContext` 作为回调实例上下文。然后通过该 `InstanceContext` 创建 `DuplexChannelFactory<TChannel>`。最后通过 `ThreadPool` 并发地执行两次服务代理的创建和服务调用的操作，客户端 ID 作为消息报头被传送到服务端。(S405)

```

public partial class MonitorForm : Form
{
    private SynchronizationContext _syncContext;
    private DuplexChannelFactory<ICalculator> _channelFactory;
    private InstanceContext _callbackInstance;
    private int _clientId = 0;
    private void MonitorForm_Load(object sender, EventArgs e)
    {
        string header = string.Format("{0,-13}{1,-22}{2}", "Client", "Time",
            "Event");
        this.listBoxExecutionProgress.Items.Add(header);
        _syncContext = SynchronizationContext.Current;
        _callbackInstance = new InstanceContext(new CalculatorCallbackService());
        _channelFactory = new DuplexChannelFactory<ICalculator>(_callbackInstance,
            "calculatorservice");

        EventMonitor.MonitoringNotificationSended += ReceiveMonitoring
            Notification;
        this.Disposed += delegate
        {
            EventMonitor.MonitoringNotificationSended -=
                ReceiveMonitoringNotification;
            _channelFactory.Close();
        };
        for (int i = 0; i < 2; i++)
        {
            ThreadPool.QueueUserWorkItem(state =>
            {
                int clientId = Interlocked.Increment(ref _clientId);
                EventMonitor.Send(clientId, EventType.StartCall);
                ICalculator proxy = _channelFactory.CreateChannel();
                using (OperationContextScope contextScope = new
                    OperationContextScope(proxy as IContextChannel))
                {
                    MessageHeader<int> messageHeader = new
                        MessageHeader<int>(clientId);
                    OperationContext.Current.OutgoingMessageHeaders.Add(

```

```

        messageHeader.GetUntypedHeader(
            EventMonitor.ClientIdHeaderLocalName,
            EventMonitor.ClientIdHeaderNamespace));
        proxy.Add(1, 2);
    }
    EventMonitor.Send(clientId, EventType.EndCall);
}, null);
}
}

public void ReceiveMonitoringNotification(object sender, MonitorEventArgs
args)
{
    string message = string.Format("{0, -15}{1, -20}{2}", args.ClientId,
        args.EventTime.ToLongTimeString(), args.EventType);
    _syncContext.Post(state =>
        this.listBoxExecutionProgress.Items.Add(message), null);
}
}
}

```

现在重新运行更新后的监控程序，将会得到如图 4-7 所示的输出结果。如果仔细分析服务端和客户端输出的结果，将会看到 Add 操作的整个执行时间有一段是重迭的，也就是说整个服务操作存在并发执行的情况。但是单看 PreCallback 和 PostCallback，则不存在并发执行的情况。从客户端的角度来看，回调操作也不存在并发执行的情况。

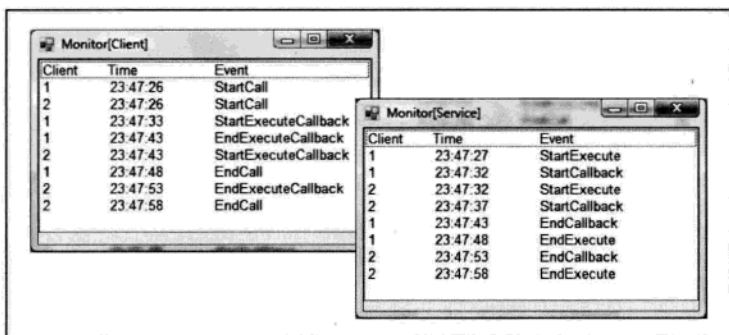


图 4-7 Reentrant (Service) + Single (Callback) 监控结果

可能上面的输出结果还不是很直观，现在我们通过时间轴的形式来描述通过输出结果表现出的执行情况。我们忽略掉客户端和服务通信及 WCF 消息分发导致的时延，两次服务调用执行的情况如图 4-8 所示。

假设服务端在 0s 接收到两个并发的调用请求，一个请求被分发给服务实例上下文，另一个则被放到等待队列。到 5s 的时候，第一个请求完成 PreCallback 的操作后进行回调，此时服务实例上下文被释放出来，使得它可以用于处理等待着的第二个请求。

到 10s 的时候，第二个请求完成了 PreCallback 操作准备进行回调，但是回调实例上下文正在处理第一个回调请求，所以自己再一次等待。直到 15s 时第一个回调请求处理完毕，回到服务实例上下文进行 PostCallback 操作，此时第二个回调请求得到处理。

第 20s，针对第一个请求的整个操作执行完毕，而此时第二个请求则正在执行回调操作。

5s 之后回调结束，回到服务实例上下文执行 PostCallback 操作。从时间轴上看，不论是服务实例上下文，还是回调实例上下文，在某个时刻只能处理一个请求。

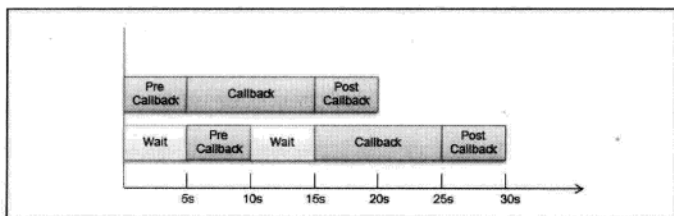


图 4-8 Reentrant (Service) + Single (Callback) 监控结果时间轴描述

如果同时将服务和回调采用的并发模式均换成 Multiple，那么无论是作用于服务实例上下文的 PreCallback 和 PostCallback 操作，还是作用于回调实例上下文的 Callback，都可以并发地执行。只需要对分别应用于 CalculatorService 和 CalculatorCallbackService 的 ServiceBehaviorAttribute 和 CallbackBehaviorAttribute 的两个特性稍加修改，按照如下的方式将 ConcurrencyMode 属性设置成 ConcurrencyMode.Multiple 即可。(S406)

```
[ServiceBehavior(UseSynchronizationContext = false,
    InstanceContextMode = InstanceContextMode.Single,
    ConcurrencyMode = ConcurrencyMode.Multiple)]
public class CalculatorService : ICalculator
{
    //省略成员
}

[CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
public class CalculatorCallbackService : ICalculatorCallback
{
    //省略成员
}
```

再次运行监控程序，得到如图 4-9 所示的输出。可以看出这正是我们想要的结果，无论是作用于服务实例上下文还是回调实例上下文的操作，都是并发执行的。

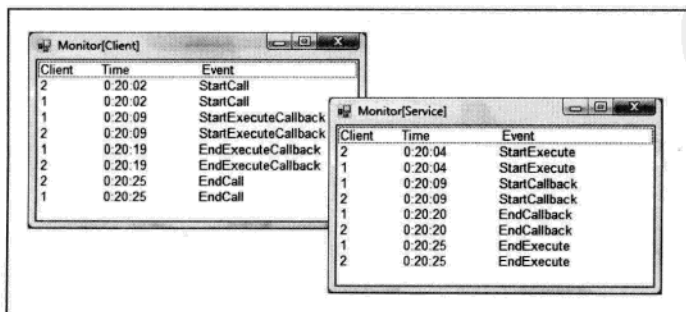


图 4-9 Multiple (Service) + Multiple (Callback) 监控结果

4.2 同步上下文与线程亲和性

在上一节中通过实例演示的方式讲述了基于不同实例上下文模式的并发行为。对于这个实例中的服务类型 `CalculatorService`, 读者应该还记得我们通过 `ServiceBehaviorAttribute` 特性将属性 `UseSynchronizationContext` 设置成 `False`。至于为何要这么做, 就是本节将讲述的内容。

4.2.1 倘若去除 `ServiceBehaviorAttribute` 的 `UseSynchronizationContext` 属性

现在我们对监控程序实例中的 `CalculatorService` 进行了一些小小的改动, 去除 `ServiceBehaviorAttribute` 特性的 `UseSynchronizationContext` 属性定义。修改后的代码如下所示, 采用单调实例上下文模式。(S407)

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall,
ConcurrencyMode=ConcurrencyMode.Multiple)]
public class CalculatorService : ICalculator
{
    //省略成员
}
```

为了让读者得到更多关于并发处理的信息, 我们让最终输出的监控信息包含当前线程的 ID。为此需要为事件参数类型 `MonitorEventArgs` 添加如下一个 `ThreadId` 属性。在构造 `MonitorEventArgs` 对象的时候, 该属性取当前线程 ID。

```
public class MonitorEventArgs : EventArgs
{
    //其他成员
    public int ThreadId
    { get; private set; }

    public MonitorEventArgs(int clientId, EventType eventType, DateTime
eventTime)
    {
        //其他属性赋值
        this.ThreadId = Thread.CurrentThread.ManagedThreadId;
    }
}
```

然后我们在寄宿服务的监控窗口中, 通过修改用于接收监控信息的方法 `ReceiveMonitoringNotification`, 将当前事件对应的线程 ID 输出, 相应的改动如下所示。

```
public partial class MonitorForm : Form
{
    //其他成员
    private void MonitorForm_Load(object sender, EventArgs e)
    {
        string header = string.Format("{0,-13}{1,-22}{2,-20}{3,-20}", "Client",
            "Time", "Thread", "Event");
```

```

        this.listBoxExecutionProgress.Items.Add(header);
        //其他操作
    }

    public void ReceiveMonitoringNotification(object sender, MonitorEventArgs
args)
    {
        string message = string.Format("{0, -13}{1, -22}{2, -20}{3, -20}",
            args.ClientId, args.EventTime.ToLongTimeString(), args.ThreadId,
            args.EventType);
        _syncContext.Post(state => this.listBoxExecutionProgress.Items.Add(
            message), null);
    }
}

```

如果现在运行监控程序，将会得到如图 4-10 所示的输出结果。该监控结果反映了两个重要的信息：

- 服务操作的执行是串行化执行的；
- 服务端是采用同一个线程执行的（线程 ID 相同）。

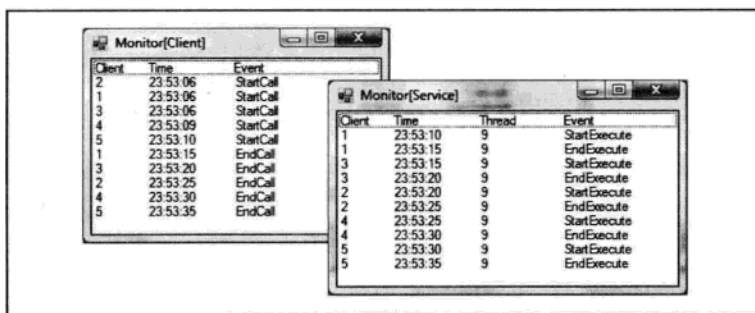


图 4-10 去除 UseSynchronizationContext 得到的监控结果

正是因为所有服务操作都是在同一个线程中执行的，才会表现出串行化执行的行为。那么是什么导致客户端并发服务请求最终被分发到同一个线程上呢？通过上面的分析，我们知道这不可能是 WCF 并发体系的同步机制所致，因为该同步机制是通过对服务实例上下文的锁定来实现的。由于 CalculatorService 采用的是单调实例上下文模式，每一个服务调用请求都会分发给一个全新的实例上下文。

通过实例可以很清楚地看到，通过去除 ServiceBehaviorAttribute 特性的 UseSynchronizationContext 属性定义让我们的服务端失去了并发执行的能力。接下来我们将着力剖析其背后的原因。不过在这之前，需要了解一下 UseSynchronizationContext 属性中设置的 SynchronizationContext，即同步上下文是什么。

4.2.2 什么是同步上下文（SynchronizationContext）

在一个多线程的应用中，经常会遇到这样的场景：在一个异步执行的方法中，需要将部

分操作递交给其他某个线程执行。比如说在一个基于 Windows Form 的应用中, 如果异步方法调用涉及对某个窗体中的某个控件的操作, 需要将该操作递交给 UI 线程中执行, 因为控件只能在自己被创建的线程中操作。可以采用两种解决方案:

- 调用 System.Windows.Forms.Control 的 Invoke 或者 BeginInvoke 方法, 将相应的操作通过委托的方式传入该方法中执行;
- 利用同步上下文 (SynchronizationContext)。

细心的朋友, 应该已经注意到了在我们前面广泛使用到的监控程序中, 不论是在客户端还是在服务端, 写入事件监控信息时就使用到了 SynchronizationContext 对象。同步上下文帮助将某个操作封送 (Marshal) 到某个指定的线程, 使其在目标线程上下文中被执行。同步上下文是在 .NET Framework 2.0 中被引入的一种多线程机制, 通过具有如下定义的抽象类 System.Threading.SynchronizationContext 表示。

```
public class SynchronizationContext
{
    //其他成员
    public virtual void Post(SendOrPostCallback d, object state);
    public virtual void Send(SendOrPostCallback d, object state);
    public static SynchronizationContext Current { get; }
}
```

SynchronizationContext 与某个线程绑定, 属于线程执行上下文 (Execution Context) 的一部分, 存储于线程本地存储 (TLS, Thread Local Storage) 中。在 SynchronizationContext 的所有成员中, 最重要的就是 Send 和 Post 两个方法, 它们以 System.Threading.SendOrPostCallback 委托的形式将相应的操作封送到 SynchronizationContext 对应的线程中执行。Send 和 Post 具有相同的方法签名, 它们之间的不同之处在于 Send 是按照同步的方式执行的, 而 Post 则是按照异步的方式执行的。静态只读属性 Current 获取存储于当前 TLS 的 SynchronizationContext, 如果不存在则返回 NULL。

再次回到前面创建的监控程序的例子。对于服务端来说, 接收监控事件通知操作和服务操作在相同的线程中执行。如果将应用在服务类型上的 ServiceBehaviorAttribute 特性的 UseSynchronizationContext 属性设置成 False, 那么该线程就不是服务寄宿的 UI 线程。对于客户端来说, 由于服务调用是以异步的方式进行的, 因此接收监控事件通知操作也不在 UI 线程上执行。在输出监控信息的时候, 需要对监控窗体的控件 (ListBox) 进行操作。由于控件是在 UI 线程上被创建的, 因此不能在监控线程中对其进行直接操作。异步线程对 UI 线程的操作, 就是通过获取 UI 线程的 SynchronizationContext 实现的。Windows Forms 应用采用的 SynchronizationContext 是一个 System.Windows.Forms.WindowsFormsSynchronizationContext 对象。

为了让读者更加容易地理解 SynchronizationContext 在 WCF 并发处理体系中的影响, 我们来做一个相关的演示实例。创建一个 Windows Forms 应用, 添加一个类似于我们的监控程

序中的窗体，里面仅仅包含用于输出进度信息的 `ListBox`。然后在窗体的 `Load` 事件中编写如下的代码。

```
int index = 0;
SynchronizationContext syncContext = SynchronizationContext.Current;
this.listBoxExecutionProgress.Items.Add(string.Format("{0, -10}{1,-10}{2}",
    "Task", "Thread", "Time"));
for(int i=0; i<5;i++)
{
    int taskSequence = Interlocked.Increment(ref index);
    ThreadPool.QueueUserWorkItem(state1 =>
    {
        syncContext.Post(state2 =>
        {
            Thread.Sleep(5000);
            string message = string.Format("{0, -10}{1,-10}{2}",
                taskSequence, Thread.CurrentThread.ManagedThreadId,
                DateTime.Now.ToLongTimeString());
            this.listBoxExecutionProgress.Items.Add(message);
        }, null);
    }, null);
}
```

上面一段简单的程序模拟这样的场景：通过 `ThreadPool` 以异步的方式调用 5 个相对耗时的操作（每一个操作耗时 5 秒，通过让线程休眠实现），并在操作执行结束后打印出当前时间和线程 ID。但是这 5 个并行操作最终却是在 UI 线程的 `SynchronizationContext` 中执行的。程序运行后将会得到如图 4-11 所示的输出结果。

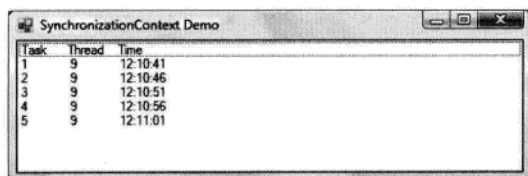


图 4-11 并行操作在相同 `SynchronizationContext` 中的执行结果

图 4-11 反映出来的结果与前面去除应用在 `CalculatorServiceAttribute` 的 `UseSynchronizationContext` 属性定义后服务端得到的监控结果比较类似（图 4-10）。5 个本应该在不同线程中并行执行的操作最终却是在相同的线程（实际上就是 UI 线程）中串行执行的。这 5 个并行处理操作可以看成是并发请求对应的 5 个服务操作。这种串行化执行并发请求的服务操作是如何产生的呢？

4.2.3 WCF 中的同步上下文与线程亲和性

WCF 服务端采用线程池并发地处理服务调用请求，所以同时抵达的服务调用请求消息能够得到及时的处理。但是服务操作具体在哪个线程线程执行，则是通过 WCF 的并发处理体系决定的。

在默认的情况下, 如果服务寄宿 (IIS/WAS 寄宿方式除外) 线程存在同步上下文, 会将其保存在服务端分发运行时中。在执行服务操作的时候, WCF 会判断分发运行时的同步上下文是否存在, 如果不存在, 则在各个线程中执行服务操作, 否则服务操作会被封送到该同步上下文中执行。

如果将某个服务寄宿于一个控制台应用, 由于控制台程序的当前同步上下文为空, 所有的请求操作会在各自的线程中并行地执行。所以在流量允许的范围内, 并发的请求能够得到及时的处理。如果将 Windows Forms 应用作为某个服务的宿主, 由于 UI 线程总是具有一个类型为 `WindowsFormsSynchronizationContext` 的同步上下文, 服务操作的执行永远以同步的方式执行。我们将服务操作与服务寄宿程序线程自动绑定的现象称为服务的线程亲和性 (Thread Affinity)。

线程亲和性使我们采用 Windows Forms 应用作为服务宿主的时候, 服务操作可以直接操作窗体上的控件。但是如果希望服务操作能够并发地被执行, 就不得不打破这种线程亲和性。这可以通过在服务类型上应用 `ServiceBehaviorAttribute` 特性并将 `UseSynchronizationContext` 属性设置成 `False` 来实现。`UseSynchronizationContext` 属性在 `ServiceBehaviorAttribute` 中具有如下的定义。

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public bool UseSynchronizationContext { get; set; }
}
```

如果你在服务类型上应用了 `ServiceBehaviorAttribute` 并将其 `UseSynchronizationContext` 属性设置为 `False`, 针对窗体控件的操作就需要调用 `Control` 类型的 `Invoke` 或者 `BeginInvoke` 方法, 或者按照我们监控程序的方式借助于 `SynchronizationContext` 对象 (调用 `Send` 或者 `Post` 方法) 了。

1. 同步上下文如何影响服务操作的执行

当服务端信道层成功接收到来自客户端的请求消息后, 会将该消息递交给相应信道监听器 (`ChannelListener`) 所在的信道分发器 (`ChannelDispatcher`)。信道分发器则根据相应的消息筛选 (`MessageFilter`) 将消息进一步分发给匹配的终结点分发器 (`EndpointDispatcher`)。终结点分发器根据自己的分发运行时 (`DispatchRuntime`) 设定的处理行为对请求消息执行进一步的处理。关于消息分发、筛选机制, 以及分发运行时的创建, 在本书的第 9 章“扩展 (Extension)”中有详细的介绍。

分发运行时控制了终结点分发器进行消息分发处理的行为, 实际上大部分作用于服务端的自定义行为 (契约行为、操作行为、服务行为和终结点行为) 都是通过对该运行时进行相应的定制, 使得 WCF 服务端框架按照我们希望的方式处理请求的消息。在表示分发运行时

的 `DispatchRuntime` 类型中定义了如下一个类型为 `SynchronizationContext` 的可读写的 `SynchronizationContext` 属性。

```
public sealed class DispatchRuntime
{
    //其他成员
    public SynchronizationContext SynchronizationContext { get; set; }
}
```

在 `DispatchRuntime` 初始化过程中, 会按照如下所示的伪代码对 `SynchronizationContext` 进行初始化。首先会遍历当前 `AppDomain` 中是否加载了名称以“`System.Web,`”为前缀的程序集, 如果这样的程序集被成功加载, 并且 `HostingEnvironment` 的 `IsHosted` 返回值为 `True`, 则将 `SynchronizationContext` 设置为 `Null`。该步骤主要判断服务寄宿的方式是否为 `IIS`, 因为这样的寄宿方式不需要同步上下文。实际上如果你采用 `ASP.NET` 应用作为宿主, 下面的代码也进入与 `IIS` 寄宿一样的逻辑分支。

如果不满足上面的条件, 则将当前线程的同步上下文赋值给 `SynchronizationContext` 属性。所以对于 `Windows Forms` 应用作为服务的宿主, `DispatchRuntime` 的 `SynchronizationContext` 将会被初始化成一个 `WindowsFormsSynchronizationContext` 对象。

```
public sealed class DispatchRuntime
{
    //其他成员
    public SynchronizationContext SynchronizationContext { get; set; }
    public DispatchRuntime()
    {
        //其他操作
        Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
        for (int i = 0; i < assemblies.Length; i++)
        {
            if (string.Compare(assemblies[i].FullName, 0, "System.Web,", 0,
                               "System.Web,", .Length, StringComparison.OrdinalIgnoreCase) == 0)
            {
                if (HostingEnvironment.IsHosted)
                {
                    this.SynchronizationContext = null;
                    return;
                }
            }
        }
        this.SynchronizationContext = SynchronizationContext.Current;
    }
}
```

而在执行服务行为 `ServiceBehaviorAttribute` 的 `ApplyDispatchBehavior` 方法的过程中, 会按照如下的方式根据 `UseSynchronizationContext` 属性值对分发运行时的 `SynchronizationContext` 属性进行重新设定。如果 `UseSynchronizationContext` 属性为 `False`, 会将 `SynchronizationContext` 设置为 `NULL`。

```
public class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
```

```

//其他成员
public bool UseSynchronizationContext { get; set; }
public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase)
{
    //其他操作
    if (!UseSynchronizationContext)
    {
        foreach (ChannelDispatcher channelDispatcher in
            serviceHostBase.ChannelDispatchers)
        {
            foreach (EndpointDispatcher endpointDispatcher in
                channelDispatcher.Endpoints)
            {
                endpointDispatcher.DispatchRuntime.SynchronizationContext = null;
            }
        }
    }
}
}

```

当真正的服务调用请求被分发给终结点分发器后，会先判断分发运行时的 `SynchronizationContext` 是否存在。如果返回结果为 `NULL`，请求消息会在各自的线程中进行处理，否则会将后续的消息处理操作封送到该 `SynchronizationContext` 表示的同步上下文中执行。因此在分发运行时的 `SynchronizationContext` 存在的情况下，后续的消息处理过程都是以同步的方式执行的。终结点分发器对请求消息的处理大体上可以通过下面一段伪代码表示。

```

public void ProcessMessage(Message message)
{
    SendOrPostCallback messageProcessCallback = GetMessageProcessCallback();
    DispatchRuntime dispatchRuntime = GetCurrentRuntime();
    SynchronizationContext context = dispatchRuntime.SynchronizationContext;
    if (context != null)
    {
        context.Post(messageProcessCallback, message);
    }
    else
    {
        IOThreadScheduler.ScheduleCallback(messageProcessCallback, message);
    }
}

```

2. 同步上下文如何影响回调操作的执行

对于非 IIS/WAS 寄宿方式，如果在进行服务寄宿的时候当前线程存在同步上下文（比如 Windows Forms 应用作为宿主），服务操作最终在该同步上下文中执行。相似的情况同样发生在回调操作的执行上。

在回调场景中，客户端进行服务调用的时候，如果当前线程存在同步上下文，那么当服务端进行回调的时候，回调操作会自动被封送到该同步上下文中执行。也就是说回调操作与客户端程序也存在一种线程关联性。

在服务端可以通过在服务类型上面应用 `ServiceBehaviorAttribute` 特性并将 `UseSynchronizationContext` 属性设置成 `False`, 来解除服务操作与服务寄宿程序之间的线程关联性。在客户端也可以采用特性标注的方式解除回调操作与客户端程序之间的线程关联性, 而这个特性就是我们之前提到过的 `CallbackBehaviorAttribute`。如下面的代码所示, `CallbackBehaviorAttribute` 特性同样具有一个 `UseSynchronizationContext` 属性。

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class CallbackBehaviorAttribute : Attribute, IEndpointBehavior
{
    //其他成员
    public bool UseSynchronizationContext { get; set; }
}
```

只需要按照如下的方式在回调类型上应用 `CallbackBehaviorAttribute` 特性, 并将 `UseSynchronizationContext` 设置成 `False`, 就可以解除回调操作与客户端程序之间的线程关联性。在这种情况下, 回调操作将会在接受回调请求的线程中执行。

```
[CallbackBehavior(UseSynchronizationContext = false)]
public class CalculatorCallbackService : ICalculatorCallback
{
    //省略成员
}
```

4.3 流量限制 (Throttling)

WCF 是一个基于多线程的消息监听、接收和处理框架体系, 能够同时应付多个来自相同或者不同客户端的服务调用请求, 并提供完善的同步机制确保状态的一致性。我们期望 WCF 服务端能够处理尽可能多的并发请求, 但是资源的有限性决定了并发量有一个最大值。如果 WCF 不控制进入消息处理系统的并发量, 试图处理所有抵达的并发请求, 那么一旦超过这个临界值, 整个服务端将会由于资源耗尽而崩溃。

我们需要在 WCF 的消息接收系统和消息处理系统之间设置一道闸门, 将流入消息处理系统的请求控制在一个最佳的范围, 以实现对现有资源的有效利用, 从而达到确保服务的可用性和提高整体吞吐量的目的。WCF 的流量限制为我们设置了这道闸门, 可以根据现有的软硬件环境对该闸门准入的并发流量进行动态的配置。

4.3.1 如何进行限流控制

和并发策略的选择一样, 对于流量的控制也属于服务自身的行为。所以 WCF 对限流的控制是通过一个服务行为的方式实现的。该服务行为就是具有如下定义的 `System.ServiceModel.Description.ServiceThrottlingBehavior`。

```
public class ServiceThrottlingBehavior : IServiceBehavior
{
    //其他成员
    public int MaxConcurrentCalls { get; set; }
    public int MaxConcurrentInstances { get; set; }
    public int MaxConcurrentSessions { get; set; }
}
```

从上面的代码中可以看到, `ServiceThrottlingBehavior` 定义了 `MaxConcurrentCalls`、`MaxConcurrentInstances` 和 `MaxConcurrentSessions` 可读写属性, 它们分别代表流量控制的三个阈值。我们所说的限流就是通过设置这三个值控制能够处理的并发量。这三个属性所代表的数值是针对某个 `ServiceHost` 而言的, 分别具有如下含义。

- `MaxConcurrentCalls`: 当前 `ServiceHost` 能够处理的最大并发消息数量, 默认值为 16;
- `MaxConcurrentInstances`: 当前 `ServiceHost` 允许存在的服务实例上下文的最大数量, 默认值为 116;
- `MaxConcurrentSessions`: 当前 `ServiceHost` 允许的最大并发会话数量, 默认值为 100。

值得一提的是, 对于 WCF 3.0 和 3.5 来说, 上面这 3 个限流指标的默认值分别为 16、116 和 100。除此之外, 对于 WCF 4.0 来说, 这些指标的值是针对单个处理器而言的。也就是对于双核处理而言, 真正的并发量是所设置数值的两倍。由于控制流量的 `ServiceThrottlingBehavior` 是一个服务行为, 因此可以通过编程和配置的方式对上述的三个允许的最大并发值进行设置。

1. 通过编程的方式设置最大并发值

控制流量的 `ServiceThrottlingBehavior` 是一个服务行为, 如果采用自我寄宿的方式, 可以直接通过编程的方式将该服务行为添加到服务描述的行为列表之中。在下面的服务寄宿代码中, 将 `CalculatorService` 服务的 `MaxConcurrentCalls`、`MaxConcurrentSessions` 和 `MaxConcurrentInstances` 分别设置成 50、30 和 20。

```
using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    ServiceThrottlingBehavior throttlingBehavior =
        host.Description.Behaviors.Find<ServiceThrottlingBehavior>();
    if (null == throttlingBehavior)
    {
        throttlingBehavior = new ServiceThrottlingBehavior();
        host.Description.Behaviors.Add(throttlingBehavior);
    }

    throttlingBehavior.MaxConcurrentCalls = 50;
    throttlingBehavior.MaxConcurrentInstances = 30;
    throttlingBehavior.MaxConcurrentSessions = 20;

    host.Open();
    //...
}
```


2. 通过配置的方式设置最大并发量

基本上所有服务行为均可通过配置的方式应用到相应的服务上面,上述关于限流的三个最大并发量指标通常都是采用配置的方式进行设置的。需要采用配置方式应用到目标服务的行为都具有一个对应的配置元素类型。而 `ServiceThrottlingBehavior` 服务行为的配置元素类是一个具有如下定义的 `System.ServiceModel.Configuration.ServiceThrottlingElement` 类。

```
public sealed class ServiceThrottlingElement : BehaviorExtensionElement
{
    //其他成员
    [ConfigurationProperty("maxConcurrentCalls", DefaultValue = 0x10)]
    public int MaxConcurrentCalls { get; set; }

    [ConfigurationProperty("maxConcurrentInstances", DefaultValue = 0x74)]
    public int MaxConcurrentInstances { get; set; }

    [ConfigurationProperty("maxConcurrentSessions", DefaultValue = 100)]
    public int MaxConcurrentSessions { get; set; }
}
```

`ServiceThrottlingElement` 的定义暴露了 `ServiceThrottlingBehavior` 对应配置项的结构。整个配置项由单纯的三个配置属性构成,分别代表上述的三个限流阈值。此外 `ServiceThrottlingElement` 还透露了一个重要的信息,就是这三个最大并发量的默认值。`MaxConcurrentCalls`、`MaxConcurrentInstances` 和 `MaxConcurrentSessions` 在默认情况下的值为 16 (0x10)、116 (0x74) 和 100,这和上面的介绍是一致的。

如果通过配置的方式控制限流,只需要将通过 `ServiceThrottlingElement` 定义的配置元素定义在相应的服务行为配置中即可。`ServiceThrottlingElement` 配置项的名称为 `serviceThrottling`。在下面的配置中,我们将寄宿服务的三个限流阈值 (`MaxConcurrentCalls`、`MaxConcurrentInstances` 和 `MaxConcurrentSessions`) 分别设置为 50、30 和 20。

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="throttlingBehavior">
          <serviceThrottling maxConcurrentCalls="50"
                             maxConcurrentInstances="30"
                             maxConcurrentSessions="20" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service behaviorConfiguration="throttlingBehavior" ...>
...
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

4.3.2 WCF 限流控制是如何实现的

通过上面的介绍,我们知道了如何通过编程和配置的方式设置相应的最大并发量,从而指导 WCF 的限流体系按照设定的值对并发的服务调用请求进行限流控制。那么在 WCF 框架体系内部,整个过程是如何实现的呢?

1. 信道分发器 (ChannelDispatcher) 与 ServiceThrottle

从服务端运行时框架的角度看,限流控制体现在信道分发器上。在信道分发器将接收到的消息分发给相应的终结点分发器之前,会进行流量的检测。至于实现流量控制的原理,会在后面讨论。在这里我们需要知道 WCF 通过具有如下定义的 `System.ServiceModel.Dispatcher.ServiceThrottle` 实现对流量的控制,而它同样具有代表上述三个并发量指标的属性定义。

```
public sealed class ServiceThrottle
{
    //其他成员
    public int MaxConcurrentCalls { get; set; }
    public int MaxConcurrentInstances { get; set; }
    public int MaxConcurrentSessions { get; set; }
}
```

如果查看 `ChannelDispatcher` 的成员列表,会看到它具有如下一个类型为 `ServiceThrottle` 的 `ServiceThrottle` 可读写属性。在 `ChannelDispatcher` 进行消息分发之前对限流的控制就是通过该属性表示的 `ServiceThrottle` 对象实现的。

```
public class ChannelDispatcher : ChannelDispatcherBase
{
    // 其他成员
    public ServiceThrottle ServiceThrottle { get; set; }
}
```

服务行为 `ServiceThrottlingBehavior` 对限流的控制最终体现在根据自身的设置(三个最大并发量指标)为信道分发器定义定制相应的 `ServiceThrottle`。由于服务行为是针对服务级别的,因此如果一个 `ServiceHost` 具有若干个信道分发器, `ServiceThrottlingBehavior` 服务行为会为每一个信道分发器进行相同的设置。`ServiceThrottlingBehavior` 对信道分发器 `ServiceThrottle` 的设置实现在 `ApplyDispatchBehavior` 方法中,大概的逻辑如下面的伪代码所示。

```
public class ServiceThrottlingBehavior : IServiceBehavior
{
    //其他成员
    void ApplyDispatchBehavior(ServiceDescription description, ServiceHostBase
        serviceHostBase)
    {
        ServiceThrottle serviceThrottle = new ServiceThrottle(serviceHostBase)
        serviceThrottle.MaxConcurrentCalls = this.MaxConcurrentCalls;
        serviceThrottle.MaxConcurrentInstances = this.MaxConcurrentInstances;
        serviceThrottle.MaxConcurrentSessions = this.MaxConcurrentSessions;
        foreach(ChannelDispatcher channelDispatcher in
```

```

        serviceHostBase.ChannelDispatchers)
    {
        channelDispatcher.ServiceThrottle = serviceThrottle;
    }
}
}

```

由于服务的限流控制最终是通过信道分发器的 `ServiceThrottle` 对象实现的, 因此我们可以通过信道分发器的 `ServiceThrottle` 的属性, 获取到我们通过编程或配置方式设置的三个最大并发量的值。假设通过配置的方式为寄宿的服务进行了如下的限流设置。

```

<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="throttlingBehavior">
          <serviceThrottling maxConcurrentCalls="50"
                             maxConcurrentInstances="30"
                             maxConcurrentSessions="20" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service name="Artech.WcfService.Service.CalculatorService"
               behaviorConfiguration="throttlingBehavior">
        <endpoint address="net.tcp://127.0.0.1:8888/calculatorservice"
                  binding="netTcpBinding"
                  contract="Artech.WcfService.Service.Interface.ICalculator"/>
        <endpoint address="net.tcp://127.0.0.1:9999/calculatorservice"
                  binding="netTcpBinding"
                  contract="Artech.WcfService.Service.Interface.ICalculator" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

在寄宿过程中, 可以通过如下的方式得到 `ServiceHost` 的每个信道分发器所有的 `ServiceThrottle` 对象, 并将 `MaxConcurrentCalls`、`MaxConcurrentInstances` 和 `MaxConcurrentSessions` 三个最大并发量打印出来。输出的结果和上面的配置是一致的。

```

using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    host.Open();
    for (int i = 0; i < host.ChannelDispatchers.Count; i++)
    {
        ChannelDispatcher channelDispatcher =
            (ChannelDispatcher)host.ChannelDispatchers[i];
        ServiceThrottle serviceThrottle = channelDispatcher.ServiceThrottle;
        Console.WriteLine("ChannelDispatcher {0}: MaxConcurrentCalls =
            {1};MaxConcurrentInstances = {2};MaxConcurrentSessions = {3}", i + 1,
            serviceThrottle.MaxConcurrentCalls,
            serviceThrottle.MaxConcurrentInstances,
            serviceThrottle.MaxConcurrentSessions);
    }
}

```

输出结果:

```
ChannelDispatcher 1: MaxConcurrentCalls = 50;MaxConcurrentInstances = 30;MaxConc
urrentSessions = 20
ChannelDispatcher 2: MaxConcurrentCalls = 50;MaxConcurrentInstances = 30;MaxConc
urrentSessions = 20
```

2. ServiceThrottle 对限流实现原理揭秘

WCF 对限流控制的实现原理,相对来说比较复杂。由于涉及很多的内部对象,因此要将限流控制机制具体的实现讲清楚,是一件不太容易的事情。接下来,我尽量用比较直白的描述简单地介绍一下 WCF 限流框架体系是如何将递交处理的请求控制在我们设置的范围内的。

无论是基于对并发会话的控制,还是对并发调用及并发实例上下文的控制,都采用的是相同的实现机制。WCF 为此专门设计了一个内部组件,我们可称其为流量限制器 (FlowThrottle)。

(1) 流量限制器

流量限制器的设计大体上如图 4-12 所示。它具有一个 Capacity 属性,表示最大流量。其内部维护一个队列和一个计数器,该队列被称为等待队列。当流量限制器初始化的时候,最大容量会被指定,等待队列为空,计数器置为零。

在要处理需要进行流量控制的请求的时候,调用者将请求递交给该流量限制器。流量限制器判断当前的计数器是否大于最大容量,如果不是则将其递交到相应的处理组件进行处理,与此同时计数器加 1。如果计数器超出最大容量,则将请求放到等待队列中。如果之前的处理被正常完成,流量限制器的计数器会减 1。如果此时等待队列不为空,则会提取第一个请求进行处理。

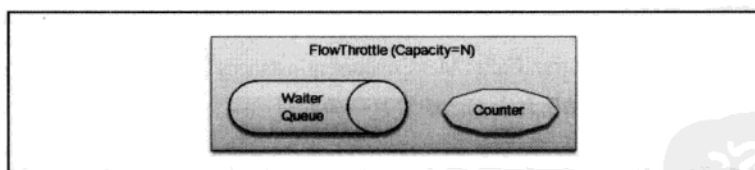


图 4-12 流量限制器设计

(2) ServiceThrottle 与流量限制器

由于 WCF 的限流通过三个指标 (最大并发请求、最大并发实例上下文和最大并发会话) 来控制,因此 ServiceThrottle 内部会维护三个不同的流量限制器,即 CallFlowThrottle、InstanceFlowThrottle 和 SessionFlowThrottle。这三个流量限制器的最大容量就是我们通过 ServiceThrottlingBehavior 设置的三个最大并发量属性 (MaxConcurrentCalls、MaxConcurrentInstances 和 MaxConcurrentSessions)。图 4-13 揭示了信道分发器、ServiceThrottle 和流量限制器之间的关系。

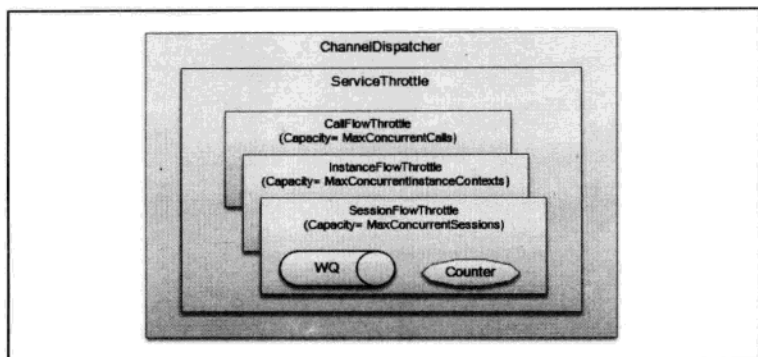


图 4-13 信道分发器、ServiceThrottle 和流量限制器之间的关系

(3) 限流控制实现

ServiceThrottle 三个流量限制器就像是设置在信道分发器中的三道闸门。当信道监听器监测到请求消息的时候，会创建信道栈接收消息。在将消息分发给相应的终结点分发器之前必须经过这三道闸门。如果一道闸门不放行，将不能再进行后续的处理，必须等到之前的操作结束，使并发的操作小于闸门限制的容量。

第一道闸门是限制并发会话的流量限制器。当信道监听器监听到抵达的详细请求后，创建信道栈对消息进行接收。如果创建的信道是会话信道，并发会话流量限制器会参与进来。并发会话流量限制器内部维护着一个会话信道计数器，如果该计数器超过通过 ServiceThrottlingBehavior 的 MaxConcurrentSessions 属性设置的最大并发量，如果没有超过则继续处理，否则将请求添加到自己的等待队列中。

如果并发会话的流量限制器放行，对请求消息的处理进入第二道屏障，即并发调用流量限制器。原理与上面相似，如果该流量限制器的并发请求数超出了通过 ServiceThrottlingBehavior 行为的 MaxConcurrentCalls 属性设置的最大并发量，请求将会被添加到自己的等待队列中，否则继续处理。

如果上面两个屏障顺利通过，WCF 会通过实例上下文提供器 (InstanceContext Provider) 获取现有的或者创建新的实例上下文。此时第三道屏障，即并发实例上下文流量控制器，开始发挥它的限流作用。与前面的并发限流机制一样，该流量限制器判断自身维护的并发实例上下文计数器是否超过了通过 ServiceThrottlingBehavior 的 MaxConcurrentInstances 属性设置的最大并发量，如果没有超过则继续处理，否则将请求添加到并发实例上下文流量控制器的等待队列中。

第5章 可靠会话

(Reliable Sessions)

作为一个通信基础平台，WCF 必须保证通信的可靠性。由于网络环境的限制，网络层不能百分之百地确保对消息的有效交付。如何克服中间环节的制约，确保从一端发送的消息能够被有效地交付给另一端，就是可靠消息传输（Reliable Messaging）需要解决的问题。WCF 通过可靠会话（Reliable Sessions）实现了端到端（End to End）的可靠消息传输。

新华书店
PDF

5.1 可靠消息传输

可以将一个通过 WCF 构建的分布式应用划分为两个部分,即客户端应用和服务端应用,它们之间的交互方式是采用某种 MEP 的消息交换。我们需要通过可靠消息传输机制确保从客户端应用发送的消息能够成功地被服务端应用接收,反之亦然。也就是说可靠消息传输提供的是一种端到端的消息传输确保机制,而不管两个终端之间是否具有相应的中间节点(Intermediary),比如路由器、防火墙和代理之类。除了确保对消息的可靠交付,可靠消息传输还需要解决以下两个问题。

- 重复消息(Duplicate Message):对于客户端发出的消息,服务端可能接收到两个以上相同的副本。可靠消息传输机制需要具有对重复消息的识别能力。
- 无序交付(Disordered Delivery):服务端接收到的消息序列可能与消息发送序列不一致。在某些情况下,我们要求 WCF 服务端框架严格按照消息在客户端应用中被发送的顺序交付给服务端应用,这需要消息传输机制提供有序消息交付(Ordered Message Delivery)的功能。

5.1.1 从 TCP 对报文段的可靠交付机制说起

IP 协议是 TCP/IP 协议簇中最为核心的协议。对于协议分层(链路层、网络层、传输层和应用层)来说,IP 协议属于网络层协议。传输层协议 TCP、UDP、ICMP 及 IGMP 协议均建立在 IP 协议之上。IP 协议传输的数据单位是数据报(Datagram),数据报的首部(一般 20 个字节)包含有源和目标的 IP 地址及其他相应的寻址和控制信息。IP 协议具有以下两个显著的特点。

- 不可靠(Unreliable):IP 协议只能尽可能提供最好的传输服务,但不能保证数据报成功地抵达目的地。如果发生某种错误,会丢弃数据报,然后发送通知信息给发送端。
- 无连接(Connectionless):发送方进行数据报发送之前并不会建立与接收方之间的连接,所以 IP 协议并不维护任何关于后续数据报的状态信息,每一个数据报都是被独立处理的。

作为传输层的 TCP 协议则是一个可靠、基于连接的协议。TCP 协议传输的数据单位被称为报文段(Segment),TCP 协议能够确保发出的报文段能够成功地抵达目的地。那么建立在不可靠的 IP 协议上的 TCP 协议是如何实现报文段的可靠交付的呢?大体上说,下面两种机制确保了 TCP 协议对“使命必达”的承诺。

- 消息确认(Message Acknowledgement):当接收端 TCP 成功接收到 TCP 报文段之后,会向发送端发送一个表明该报文段已经被成功接收的确认消息(Acknowledgement)。

- 超时重传: 发送端具有一个存储报文段的缓冲区 (Buffer), 一般称为发送端窗口 (Sender Window), 用于存放已经发送但是尚未接收到确认的报文段。如果在一定的超时时限内没有接收到确认消息, 会认为相应的报文段发送失败, 此时发送端 TCP 会从发送端窗口中提取相应的报文段进行重新发送。

对于上述的消息确认和超时重传的机制, 细心的读者会发现, 这会出现两个问题。

(1) 如果接收端 TCP 成功接收到某个报文段, 并且成功发送了确认消息, 但是如果确认消息丢失, 发送端 TCP 会对相应的报文段进行重传, 意味着接收端有可能接收到两份重复的报文段。很显然接收端 TCP 不能将重复的报文段向上层交付, 那么如何解决重复报文段的问题呢?

(2) 报文段在发送端 TCP 发送的节奏和在接收端 TCP 被接收的节奏是不同的, 所以不可能保证报文段完全以发送的顺序被接收。但是接收端 TCP 必须保证交付给上层的报文段的顺序是报文段被发送的顺序, 这又是如何做到的呢?

要解决上述的两个问题, 首先需要解决的是对报文段的识别机制。对于 TCP 协议来说, 每一个报文段具有一个序列号, 一般代表报文段承载的数据在整个发送的数据块所处的位置 (以字节为单位), 通过这个序列号就可以确定报文段发送的顺序。

除了发送端 TCP 具有一个发送端窗口外, 接收端 TCP 同样具有自己的接收端窗口, 用于存放已经接收但尚未交付的报文段。如果具有大序列号的报文段被先接收到, 则会先被置于接收端窗口中, 只有等到前面所有的报文段都抵达之后, 接收端 TCP 才会按照序列号的顺序依次对报文段实施交付。如果接收到的报文段的序列号小于或者等于接收端窗口的任何一个报文段的序列号, 接收端 TCP 会将其作为重复报文段而丢弃。

从上面的介绍可以看出, TCP 协议已经解决了我们之前提出的关于可靠消息传输的三个难题, 即消息的可靠交付、重复消息处理和有序交付。实际上, WCF 基于可靠会话机制的可靠消息传输的实现原理和 TCP 协议基本一致, 如果硬是要找出不一致的地方, 主要表现在以下 4 点。

- WCF 可靠消息传输是基于 SOAP 消息级别的, TCP 则是基于报文段级别;
- WCF 可靠消息传输是与传输协议无关的, 并不限于 TCP 协议;
- WCF 的可靠消息传输并没有具体传输会话 (Transport Session) 的限制, 可以跨越多个传输连接或者会话;
- TCP 在当前 TCP 连接范围内提供端到端的可靠传输, 而 WCF 的可靠消息传输在两个 SOAP 终结点之间提供可靠传输, 并不受传输连接 (Transport Connection) 的限制。

由于可靠消息传输对于 SOA 的重要性, 在 WS-* 协议簇中具有专门的规范, 即我们即将介绍的 WS-Reliable Messaging, 简称 WS-RM。

5.1.2 WS-RM 简介

WS-RM 是 WS-Reliable Messaging 的简称, 是 WS-* 大家庭的一个重要成员。和前面介绍的 WS-Coordination、WS-AT 一样, WS-RM 的制定者是结构化信息标准促进组织 (OASIS, Organization for the Advancement of Structured Information Standards)。

制定 WS-RM 的一个主要目的就是实现一种模块化的可靠消息传输机制。WS-RM 定义了一种消息传输协议, 以实现在可靠消息传输过程中对消息的识别、追踪和管理。到目前为止, WS-RM 先后出了 WS-RM 1.0 和 WS-RM 1.1 两个官方版本。接下来对 WS-RM 的介绍完全基于 WS-RM 1.1 版本。WS-RM 1.1 的命名空间为 <http://docs.oasis-open.org/ws-rx/wsrml/200702>, 也可以直接通过命名空间表示的 URL 查看 WS-RM 官方文档。

1. 可靠消息传输模型

可靠消息传输满足以下三个可靠性诉求, 即接收保障、重复筛选和有序交付。接收保障确保从消息源发送的消息能够成功地抵达目的地。重复筛选意味着消息的接收端能够识别每一个接收到的消息, 自动丢弃重复的消息。而有序交付要求消息的接收端能够完全按照消息发送的顺序对消息进行交付。

整个可靠消息传输模型大体上如图 5-1 所示。如果我们将发送消息的称为应用源 (Application Source), 将接收消息的应用称为应用目的地 (Application Destination), 那么可靠消息传输题是为了确保应用源和应用目的地之间消息传输的可靠性而存在的。

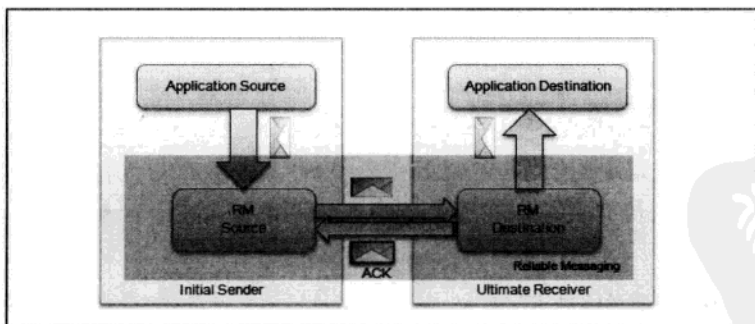


图 5-1 可靠消息传输模型

在消息发送端, 应用源将消息发送给本地的可靠消息传输体系, 即可靠消息传输源 (RM Source, 以下简称 RM 源)。RM 源将消息发送到目的地。可靠消息传输目的地 (RM Destination, 以下简称 RM 目的地) 接收到消息之后, 会向 RM 源发送确认 (ACK) 消息, 表明消息已经被成功接收。如果消息准确无误 (这里主要指消息序号是否和上次交付的消息相邻), 则

将消息交付给应用目的地。

和 TCP 实现对报文段的可靠传输一样, 在这里依然采用消息缓存、消息确认和超时重传的机制提供对可靠消息传输的三个目标的实现。每个消息会被赋予一个消息序号, 该序号在该可靠消息传输上下文中是唯一的。RM 源和目的地具有各自的消息缓冲区 (或者说消息窗口) 用于缓存消息。RM 源用该消息窗口存放已经发送但是尚未接收到确认的消息, 我们可以将这样的消息称为状态未决 (In-Doubt) 消息。RM 目的地则利用消息窗口存放成功接收但是尚未向应用目的地交付的消息。

对于某个已经发送的消息, 如果在设定的超时时限内没有成功接收到相应的确认, RM 源会认为该消息发送失败。它会从自己的消息窗口中选择对应的消息进行重新发送。只有在成功接收到确认消息的情况下, RM 源才会将消息从消息窗口中移除。

当 RM 目的地成功接收到消息后, 如果消息的序号和上次交付消息的序号相邻, 它会将消息交付给应用目的地。否则表明之前发送出来的消息尚未抵达, 此时 RM 目的地会将该消息放到自己的消息窗口中。只有等到之前所有的消息全部成功接收后, RM 目的地才会按照消息的序号对消息实施交付, 从而保证了对消息的有序交付。如果接收到的消息序号小于或者等于已经交付的消息序号, 或者等于消息窗口的某个消息的序号, RM 目的地将其视为重复消息予以丢弃。

WS-RM 仅对消息窗口所应提供的功能进行规定, 并没有对消息缓存的具体实现做出任何的限制。也就是说, 我们可以采用基于内存的方式将消息缓存在当前进程的内存之中, 这样可以带来更好的性能提升。也可以采用持久化的存储方式, 将序列化后的消息存储于物理存储介质中, 这样可以实现跨进程的数据共享以及程序中断后的数据恢复。前者被 WCF 中的可靠会话所采用。

2. 从消息交换来看可靠消息传输的处理流程

接下来我们将目光聚焦到具体的消息交换层面, 看看 WS-RM 采用怎样的消息交换方式提供对消息确认、超时重传的实现。在这之前先介绍一个非常重要的概念——序列 (Sequence)。

可靠消息传输致力于在两种终结点 (广义 Web 服务终结点) 之间提供对接收保障、重复筛选和有序交付的实现。但是可靠消息传输具有一个执行范围, 或者说可靠消息传输的实现是基于某个上下文环境的。这相当于是一个会话 (Session), 而该会话在 WS-RM 的词汇中被称为序列。两个终结点之间能够实现可靠消息传输之前, 需要先在他们之间创建一个序列, 该序列为可靠消息传输提供一个执行上下文。

同样用 TCP 对报文段的可靠传输作为类比。TCP 是一个完全基于连接的协议, 在利用 TCP 进行报文传输之前, 两个 TCP 端点之间需要通过三次“握手”建立连接。也就是说 TCP

对报文段的可靠传输是在一个确立的连接中进行的，TCP 连接承担执行上下文的作用。所以 TCP 连接之于 TCP 报文的可靠传输，就相当于序列之于 WS-RM 可靠消息传输。

可靠消息传输机制的核心是“消息窗口”和“消息识别”。为了让 RM 源和目的地能够有效地识别每一个消息，每一个消息会被赋予一个消息序号（Message Number）。该消息序号从 1 开始，并在可靠消息序列这个上下文中是唯一的。

对 WS-RM 下的序列有了一个大致的了解之后，我们结合图 5-2 所示的序列图讨论一下 WS-RM 下的消息确认机制和超时重传是如何实现的。基于 WS-RM 消息交换的两个终端是 Endpoint A 和 Endpoint B，它们分别是消息的最初发送者和最终接收者。整个过程由 10 个步骤完成。

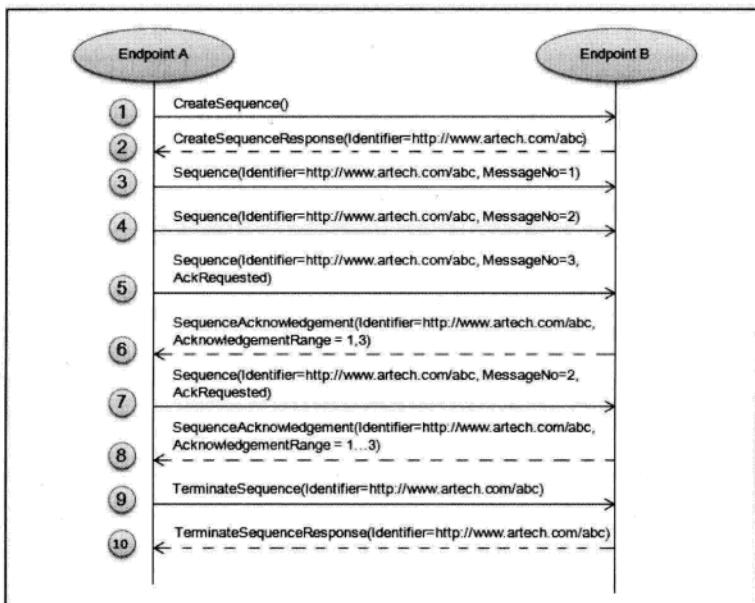


图 5-2 WS-RM 消息交换

(1) Endpoint A 向 Endpoint B 发送一个创建 CreateSequence 的请求，Endpoint B 接收到请求之后创建一个全新的序列，并将序列的唯一标识（http://www.artech.com/abc）以 CreateSequenceResponse 的形式返回给 Endpoint A（步骤 1~2）。

(2) Endpoint A 连续向 Endpoint B 发送三个消息。这三个消息均具有 WS-RM 相关的报头，主要包括序列的唯一标识和消息序列（消息序号分别为 1、2 和 3）。为了避免频繁的网络传输，WS-RM 的消息会采用一种批量确认的机制。也就是说 RM 目的地对它所接收的消息并不是逐个确认的，而是将之前接收到的消息序号放入一个确认范围（Acknowledgement Range）中，一并发送给 RM 源进行批量确认。

反映在 RM 源上，如果它期望在某次消息发送后接收到对方的确认，就需要在该消息中

插入一个 AckRequested 报头。步骤 5 中发送的第三个消息就包含了这样一个消息报头 (步骤 3~5)。

(3) 假设 Endpoint B 先后接收到序号为 1 和 3 的两个消息, 第二个消息则传输失败。当接收到包含有 AckRequested 报头的消息之后, 会向 Endpoint A 发送确认消息。在确认消息中的确认范围中, 包括 1 和 3 (步骤 6)。

(4) Endpoint A 接收到确认消息后, 分析确认范围中的消息序号, 发现确认消息序号中只有 1、3 而没有 2, 知道序号为 2 的消息发送失败。于是会从发送窗口中提取序号为 2 的消息进行重传, 该消息同样具有 AckRequested 报头, 因为它期望及时接收到对重传消息的确认 (步骤 7)。

(5) Endpoint B 成功接收到重传的序号为 2 的消息后, 发送确认消息进行确认, 需要注意的是确认范围中不仅仅包含 2, 还包括之前成功接收的消息序号 1 和 3 (步骤 8)。

(6) Endpoint A 接收到确认。如果此时不需要进行后续的消息交换工作, 可以发送 TerminateSequence 消息请求终止序列, Endpoint B 终止序列, 并返回 TerminateSequence Response 消息确认序列的终止 (步骤 9~10)。

3. WS-RM 消息

我们认识了从序列创建到终止过程中消息交换的大致流程。接下来进一步将关注点聚焦到单个消息上, 看看在整个基于序列的上下文中不同类型的消息具有怎样的结构。

(1) CreateSequence 和 CreateSequenceResponse

WS-RM 的可靠消息传输从序列的创建开始。为了创建序列, RM 源 (RM Source) 向 RM 目的地 (RM Destination) 发送一个主体包含 <CreateSequence> 元素的 SOAP 消息。该 <CreateSequence> 元素携带序列相关的属性, 比如序列过期时限及序列相关的其他信息。成功接收到序列创建请求后, RM 目的地成功创建序列, 将序列相关信息封装到 <CreateSequenceResponse> 元素中, 并最终通过 SOAP 消息返回。

```
<wsrm:CreateSequence ...>
  <wsrm:AcksTo> wsa:EndpointReferenceType </wsrm:AcksTo>
  <wsrm:Expires ...> xs:duration </wsrm:Expires> ?
  <wsrm:Offer ...>
    <wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
    <wsrm:Endpoint> wsa:EndpointReferenceType </wsrm:Endpoint>
    <wsrm:Expires ...> xs:duration </wsrm:Expires> ?
    <wsrm:IncompleteSequenceBehavior>
      wsrm:IncompleteSequenceBehaviorType
    </wsrm:IncompleteSequenceBehavior> ?
    ...
  </wsrm:Offer> ?
  ...
</wsrm:CreateSequence>
```

上面的 XML 片段展示了 CreateSequence 元素的结构。其中 <wsrm:AcksTo> 是必需的,

它表示 RM 目的地发送确认消息的目的终结点引用。而<wsrm:Expires>表示请求创建的可靠消息传输序列的过期时限。如果该值为 PT0S（默认值），则表明可靠消息传输序列永不过期。

WS-RM 中某个 RM 序列只能保证单向的消息传输的可靠性。也就是说确保从终结点 A 到终结点 B 的可靠消息传输的 RM 序列不能提供从终结点 B 到终结点 A 的可靠消息传输保障。要想解决这种双向（Two-Way）可靠消息传输，需要借助于两个 RM 序列。所以请求-回复模式和双工模式下的可靠消息传输需要双 RM 序列的支持。

WS-RM 通过序列提供机制（Sequence Offering）对此提供支持。如果两个终结点之间存在请求-回复或者双工消息交换模式，RM 源会在本地创建 RM 序列，并将创建的序列封装到 CreateSequence 消息的<Offer>元素中，提供给 RM 目的地。

RM 目的地接收到序列创建请求的 CreateSequence 消息后，会创建序列并将序列的相关信息封装到回复消息中。当 RM 源接收到了回复之后，意味着它可以在指定的序列（通过序列的标识）所在的可靠传输上下文中发送消息了。可靠消息传输回复消息的主体部分包含一个具有如下结构的<CreateSequenceResponse>元素。

```
<wsrm:CreateSequenceResponse>
  <wsrm:Identifier> xs:anyURI </wsrm:Identifier>
  <wsrm:Expires> xs:duration </wsrm:Expires> ?
  <wsrm:IncompleteSequenceBehavior>
    wsrm:IncompleteSequenceBehaviorType
  </wsrm:IncompleteSequenceBehavior> ?
  <wsrm:Accept>
  <wsrm:AcksTo> wsa:EndpointReferenceType </wsrm:AcksTo>
  ...
  </wsrm:Accept> ?
  ...
</wsrm:CreateSequenceResponse>
```

其中<wsrm:Identifier>中的 URI 为创建序列的唯一标识。<wsrm:Expires>为序列过期时间，一般来说被创建的序列过期时间小于或者等于 CreateSequence 请求中指定的 RM 源所期望的过期时间。如果该值为 PT0S（默认值），意味着序列永不过期。<wsrm:Accept>元素封装的部分表示接受在 CreateSequence 请求中提供的序列。

（2）CloseSequence 和 CloseSequenceResponse

这里讲的序列的关闭（Close），并不是指前面消息交换流程中的序列的终止（Terminate）。这是一个前面没有提及的概念。在可靠消息传输序列被使用过程中，RM 源和 RM 目的地都可以请求停止使用该序列。而对序列的终止会使 RM 源和目的地失去其现有的状态，所以为了确保可靠消息序列能够以一种可知的最终状态结束，RM 源和 RM 目的地均可以在最终终止序列之前选择将其关闭。

如果一方（RM 源或者 RM 目的地）希望关闭现有的序列，它会向另一方发送序列关闭请求消息。该消息主体部分包含一个具有如下结构的<CloseSequence>元素。<wsrm:Identifier>的值为序列的唯一标识。<wsrm:LastMsgNumber>是最后发送的消息序号，也是在该序列上

下文中发送的最后一个消息的序号。

```
<wsrm:CloseSequence>
  <wsrm:Identifier> xs:anyURI </wsrm:Identifier>
  <wsrm:LastMsgNumber> wsrm:MessageNumberType </wsrm:LastMsgNumber> ?
  ...
</wsrm:CloseSequence>
```

当 CloseSequence 请求消息被另一方接收之后,表明它不再接收该序列范围内的其他消息。作为回复,它会创建一个主体部分包含具有如下结构的<CloseSequenceResponse>元素的 SOAP 消息。在<CloseSequenceResponse>元素中,只有包含有序列标识的<wsrm:Identifier>是必需的。除了主体部分包含<CloseSequenceResponse>元素之外,序列关闭回复消息还必须包含一个具有<Final>子元素的<SequenceAcknowledgement>确认 SOAP 报头,代表对最后一个消息的确认。

```
<wsrm:CloseSequenceResponse>
  <wsrm:Identifier> xs:anyURI </wsrm:Identifier>
  ...
</wsrm:CloseSequenceResponse>
```

(3) TerminateSequence 和 TerminateSequenceReponse

当 RM 源完成所有消息传输工作之后,它可以向 RM 目的地发送一个主体部分包含<TerminateSequence>元素的消息,通知对方当前序列已经完成,并且不会继续发送基于该序列的消息。成功接收到序列终止请求消息后,RM 目的地可以进行针对当前序列相关的资源回收工作。在正常的情况下,RM 源在接收到所有消息的确认后才会向 RM 目的地发送序列终止的请求,但是 WS-RM 对此并没有严格的限制。也就是说无论所有消息的确认是否成功接收,RM 源都可以强行终止对应的序列。此外除了 RM 源,RM 目的地也可以主动终止当前序列。

为了帮助 RM 目的地确定它是否成功接收到即将被终止的序列关联的所有消息,RM 源会将自己在序列上下文中发送的最后一个消息的序号发送给它。所以<TerminateSequence>元素除了包含封装有序列标识的<Identifier>子元素之外,还具有必须包含最后消息序号的<LastMsgNumber>元素。下面的 XML 代表了<TerminateSequence>元素的结构。对于通过<LastMsgNumber>表示的消息序号,还有一点需要特别说明,那就是该消息序号必须和序列关闭请求消息中<CloseSequence>元素下的同名元素具有相同的值。

```
<wsrm:TerminateSequence>
  <wsrm:Identifier> xs:anyURI </wsrm:Identifier>
  <wsrm:LastMsgNumber> wsrm:MessageNumberType </wsrm:LastMsgNumber> ?
  ...
</wsrm:TerminateSequence>
```

当序列终止请求被接收方(RM 源或者 RM 目的地)接受后,它会回复一个主体部分包含具有如下结构的<TerminateSequenceReponse>元素的消息。<TerminateSequenceReponse>元素结构非常简单,仅包含一个表示序列标识的<Identifier>元素。

```
<wsrm:TerminateSequenceResponse>
  <wsrm:Identifier> xs:anyURI </wsrm:Identifier>
  ...
</wsrm:TerminateSequenceResponse>
```

(4) Sequence、AckRequested 和 SequenceAcknowledgement

基于 WS-RM 的消息交换是在实现创建的 RM 序列上下文中进行的。为了让接收方清楚地知道它所接收的消息具体属于哪一个 RM 序列,需要发送方在发送之前向消息的报头集合中添加相应的序列关联信息。与序列关联信息一并添加的,还有消息的序号。WS-RM 将这两组信息封装在一个具有如下结构的<Sequence>报头中。其中<Identifier>元素的值为 RM 序列的唯一标识,而<MessageNumber>即为消息在当前序列上下文中的序号。

```
<wsrm:Sequence>
  <wsrm:Identifier> xs:anyURI </wsrm:Identifier>
  <wsrm:MessageNumber> wsrm:MessageNumberType </wsrm:MessageNumber>
  ...
</wsrm:Sequence>
```

如果 RM 源希望 RM 目的地在接收到某个消息之后对所接收的所有消息进行确认,它会在该消息的报头集合中添加一个具有如下结构的<AckRequested>报头。该报头中仅包含一个必需的表示 RM 序列标识的<Identifier>元素。

```
<wsrm:AckRequested>
  <wsrm:Identifier> xs:anyURI </wsrm:Identifier>
  ...
</wsrm:AckRequested>
```

如果 RM 目的地接收到一个包含有<AckRequested>报头的消息,会对之前成功接收的所有消息进行批量确认。确认的信息被封装到一个具有如下结构的<SequenceAcknowledgement>报头中,该报头内部包含所有成功接收到的消息的序号。

```
<wsrm:SequenceAcknowledgement ...>
  <wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
  [ [ [ <wsrm:AcknowledgementRange ...
    Upper="wsrm:MessageNumberType"
    Lower="wsrm:MessageNumberType"/> +
    | <wsrm:None/> ]
    <wsrm:Final/> ? ]
    | <wsrm:Nack> wsrm:MessageNumberType </wsrm:Nack> + ]
  ...
</wsrm:SequenceAcknowledgement>
```

上面的 XML 片段可能看起来有点头晕,为了使读者能够很直观地了解<SequenceAcknowledgement>报头的样式,我们给出几种典型的实例。下面的报头意味着对序号 1~10 的消息的接收确认。

```
<wsrm:SequenceAcknowledgement>
  <wsrm:Identifier>http://www.artech.com/abc</wsrm:Identifier>
  <wsrm:AcknowledgementRange Upper="10" Lower="1"/>
</wsrm:SequenceAcknowledgement>
```

下面的<SequenceAcknowledgement>报头则表示对序号从1~2, 4~6及8~10的所有消息的接收确认。

```
<wsrm:SequenceAcknowledgement>
  <wsrm:Identifier>http://www.artech.com/abc</wsrm:Identifier>
  <wsrm:AcknowledgementRange Upper="2" Lower="1"/>
  <wsrm:AcknowledgementRange Upper="6" Lower="4"/>
  <wsrm:AcknowledgementRange Upper="10" Lower="8"/>
</wsrm:SequenceAcknowledgement>
```

下面的<SequenceAcknowledgement>报头是 RM 目的地对序号为3的消息的确认, 不过这是一个负面确认 (NACK), 意味着序号为3的消息没有被接收到。

```
<wsrm:SequenceAcknowledgement>
  <wsrm:Identifier>http://www.artech.com/abc</wsrm:Identifier>
  <wsrm:Nack>3</wsrm:Nack>
</wsrm:SequenceAcknowledgement>
```

当 RM 目的地进行消息确认的时候, 可以创建一个单独的空消息, 并附加上<SequenceAcknowledgement>报头发送给序列确认目标终结点引用 (即序列创建请求的 CreateSequence 元素中 AckTo 中指定的终结点引用)。如果 RM 目的地需要向确认目标终结点引用发送消息, 它也可以直接将<SequenceAcknowledgement>报头附加在该消息之上。这种为了避免创建新的消息, 直接将相应的信息服务加到另一个现有的具有相同目标终结点引用的消息上的方式被称为“背负 (piggy-back)”机制。

5.2 编写可靠会话服务

作为 WCF 对 WS-RM 的支持, 可靠会话 (Reliable Sessions) 是通过在信道层建立起一个可靠消息传输上下文来实现的。可靠消息传输的消息交换是在一个 RM 序列中进行的, RM 序列为整个可靠消息传输过程建立起一个上下文环境。完全可以将 RM 序列看成一个会话 (Session), 即一个消息发送方和接收方为了实现对消息的可靠传输而建立的会话。所以对 WS-RM 的实现取名为可靠会话是非常贴切的。

在本节中, 将重点介绍可靠会话编程方面的内容。为了使读者能够对可靠会话的基本编程方式和达到的功效有一个深刻的理解, 我们先来做一个实例演示。

5.2.1 实例演示: 通过 WCF 服务传输图片 (S501)

我们将要演示的实例是对可靠会话确保 WCF 消息传输的可靠性的一个直观反映, 也是早年微软推广 WCF 技术频繁使用的案例——图片传输。我们在客户端选择一张图片, 并对它进行切片, 最后通过调用 WCF 服务将每一个切片依次传输到服务端。服务端则按照切片被接收到的顺序重新组装成一张完整的图片。

如果中间有任何一张切片丢失，服务端最终组装图片将不会完整。如果服务端切片接收的次序和发送顺序不一致，将会造成组装后的图片并不能还原其发送前的模样。在这里充分利用了 WCF 中的可靠会话提供了可靠而有序的消息交付。

不稳定的网络是造成消息丢失的最主要的因素，但是在本机环境下模拟不稳定的网络是一件比较困难的事情。虽然我们不能让消息在网络传输层中丢失，但是可以让它在 WCF 的信道层中丢失。

步骤一：通过自定义信道模拟不稳定的网络

为了能够对网络传输过程中的丢包率进行动态控制，我们创建一个具有如下定义的类型 `MessageInspector`（与 WCF 的 `ClientMessageInspector` 和 `DispatchMessageInspector` 无关）。只读属性 `DropRate` 表示丢包率，`ProcessMessage` 对传入的消息进行处理，如果返回为 `null`，则意味着消息的丢失。

```
using System;
using System.ServiceModel.Channels;
namespace Artech.ImageTransfer.Extensions
{
    public class MessageInspector
    {
        public int DropRate { get; private set; }
        public Random Randomizer { get; private set; }
        public MessageInspector(int dropRate)
        {
            this.DropRate = dropRate;
            this.Randomizer = new Random();
        }
        public virtual void ProcessMessage(ref Message message)
        {
            int randomNumber = this.Randomizer.Next(100);
            if (randomNumber <= this.DropRate)
            {
                message = null;
            }
        }
    }
}
```

接下来创建如下一个用于模拟不稳定网络环境的自定义信道 `UnreliableNetworkSimulateChannel`。我们的实例采用 TCP 传输方式，所以让它实现了 `IDuplexSessionChannel` 接口。`UnreliableNetworkSimulateChannel` 通过 `MessageInspector` 对象对传入的消息进行加工（根据丢包率随机地丢弃）。`MessageInspector` 在构造函数中创建，而丢包率通过参数传入。除了 `Send` 方法，几乎所有的成员都是调用 `InnerChannel` 相应的方法或者返回同名的属性。由于在上册的第 3 章“绑定（Binding）”中有过对如何自定义信道的专门介绍，在这里我们就不再多做重复的讲述了。如果你希望了解整个 `UnreliableNetworkSimulateChannel` 的定义，可以下载本实例的源代码（S501）。

```

using System;
using System.ServiceModel.Channels;
namespace Artech.ImageTransfer.Extensions
{
    public class UnreliableNetworkSimulateChannel : IDuplexSessionChannel
    {
        public IDuplexSessionChannel InnerChannel{ get; private set; }
        public MessageInspector MessageInspector{ get; private set; }
        public UnreliableNetworkSimulateChannel(IDuplexSessionChannel innerChannel,
            int dropRate)
        {
            this.InnerChannel = innerChannel;
            this.MessageInspector = new MessageInspector(dropRate);
        }
        public IAsyncResult BeginReceive(TimeSpan timeout, AsyncCallback callback,
            object state)
        {
            return this.InnerChannel.BeginReceive(timeout, callback, state);
        }
        public void Send(Message message, TimeSpan timeout)
        {
            this.MessageInspector.ProcessMessage(ref message);
            if (null != message)
            {
                this.InnerChannel.Send(message, timeout);
            }
        }
        public void Send(Message message)
        {
            this.MessageInspector.ProcessMessage(ref message);
            if (null != message)
            {
                this.InnerChannel.Send(message);
            }
        }
        //其他成员:直接调用 InnerChannel 的相应的方法或者返回同名属性
    }
}

```

通过上面的代码可以看到, 在 `Send` 方法中消息对象会先传入 `MessageInspector` 的 `ProcessMessage` 方法中, 如果返回值不为空, 将其递交给 `InnerChannel`, 反之意味着消息在信道层中丢失。接下来为该自定义信道创建信道管理器, 由于该信道只在客户端使用, 因此只需要为之创建信道工厂即可 (`Channel Factory`)。 `UnreliableNetworkSimulateChannel` 对应的信道工厂 `UnreliableNetworkSimulateChannelFactory<TChannel>` 定义如下。

```

using System;
using System.ServiceModel.Channels;
using System.ServiceModel;
namespace Artech.ImageTransfer.Extensions
{
    public class UnreliableNetworkSimulateChannelFactory<TChannel> :
        ChannelFactoryBase<IDuplexSessionChannel>
    {
        public int DropRate{get; private set;}
        public IChannelFactory<TChannel> InnerChannelFactory{ get; private set; }
        public UnreliableNetworkSimulateChannelFactory(BindingContext context,

```

```

        int dropRate):base(context.Binding)
    {
        this.InnerChannelFactory = context.BuildInnerChannelFactory<TChannel>();
        this.DropRate = dropRate;
    }
    protected override IDuplexSessionChannel OnCreateChannel(EndpointAddress
        address, Uri via)
    {
        var innerChannel =
            (IDuplexSessionChannel)this.InnerChannelFactory.CreateChannel(
                address, via);
        return new UnreliableNetworkSimulateChannel(innerChannel, this.DropRate);
    }
    protected override IAsyncResult OnBeginOpen(TimeSpan timeout, AsyncCallback
        callback, object state)
    {
        return this.InnerChannelFactory.BeginOpen(timeout, callback, state);
    }
    protected override void OnEndOpen(IAsyncResult result)
    {
        this.InnerChannelFactory.EndOpen(result);
    }
    protected override void OnOpen(TimeSpan timeout)
    {
        this.InnerChannelFactory.Open(timeout);
    }
}
}

```

我们为信道工厂创建了一个具有如下定义的 `UnreliableNetworkSimulateBindingElement` 绑定元素。

```

using System.ServiceModel.Channels;
namespace Artech.ImageTransfer.Extensions
{
    public class UnreliableNetworkSimulateBindingElement : BindingElement
    {
        public int DropRate { get; set; }
        public UnreliableNetworkSimulateBindingElement(int dropRate)
        {
            this.DropRate = dropRate;
        }
        public override BindingElement Clone()
        {
            return new UnreliableNetworkSimulateBindingElement(this.DropRate);
        }
        public override T GetProperty<T>(BindingContext context)
        {
            return context.GetInnerProperty<T>();
        }
        public override IChannelFactory<TChannel>
            BuildChannelFactory<TChannel>(BindingContext context)
        {
            return (IChannelFactory<TChannel>)new UnreliableNetworkSimulate-
                ChannelFactory<TChannel>(
                    context, this.DropRate);
        }
    }
}

```

为了使上面的绑定元素具有可配制性，我们通过继承 `BindingElementExtensionElement` 为绑定元素定义一个具有如下定义的 `UnreliableNetworkSimulateExtensionElement` 配置元素类。将丢包率定义成配置属性，该属性默认值为 20（20%丢包率）。

```
using System;
using System.Configuration;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
namespace Artech.ImageTransfer.Extensions
{
    public class UnreliableNetworkSimulateExtensionElement :
        BindingElementExtensionElement
    {
        [ConfigurationProperty("dropRate", IsRequired = false, DefaultValue = 20)]
        public int DropRate
        {
            get{return (int)this["dropRate"];}
            set{this["dropRate"] = value;}
        }
        public override Type BindingElementType
        {
            get { return typeof(UnreliableNetworkSimulateBindingElement); }
        }
        protected override BindingElement CreateBindingElement()
        {
            return new UnreliableNetworkSimulateBindingElement(this.DropRate);
        }
    }
}
```

步骤二：创建图片传输服务

服务契约 `IImageTransfer` 具有两个单向（One-Way）服务操作。`Transfer` 方法用于对图片切片（以字节数组的形式）的传输，而 `Erase` 则用于通知接收端将之前接收的图片删除。

```
using System.ServiceModel;
namespace Artech.ImageTransfer.Service.Interface
{
    [ServiceContract(Namespace="http://www.artech.com/")]
    public interface IImageTransfer
    {
        [OperationContract(IsOneWay = true)]
        void Transfer(byte[] imageSlice);
        [OperationContract(IsOneWay = true)]
        void Erase();
    }
}
```

服务端需要将接收到的图片切片组装成一个完整的图片，这里通过如下一个叫做 `ImageAssembler` 的静态类来提供图片组装的功能。对应于服务契约定义的两个服务操作，`ImageAssembler` 中定义两个静态事件 `ImageSliceReceived` 和 `ImageErasing`。这两个事件分别通过静态方法 `ReceiveImageSlice` 和 `Erase` 触发。事件 `ImageSliceReceived` 的事件参数类型为 `ImageReceivedEventArgs`，它和 `ImageAssembler` 定义如下。

```

using System;
namespace Artech.ImageTransfer.Service
{
    public static class ImageAssembler
    {
        public static void ReceiveImageSlice(byte[] imageSlice)
        {
            if (null != ImageSliceReceived)
            {
                ImageSliceReceived(null, new ImageReceivedEventArgs(imageSlice));
            }
        }

        public static void Erase()
        {
            if (null != ImageErasing)
            {
                ImageErasing(null, EventArgs.Empty);
            }
        }

        public static event EventHandler<ImageReceivedEventArgs>
            ImageSliceReceived;
        public static event EventHandler ImageErasing;
    }

    public class ImageReceivedEventArgs : EventArgs
    {
        public byte[] ImageSlice { get; private set; }

        public ImageReceivedEventArgs(byte[] imageSlice)
        {
            this.ImageSlice = imageSlice;
        }
    }
}

```

接下来是图片传输服务的实现，该服务定义在如下的 `ImageTransferService` 类中。对于两个服务操作，分别调用 `ImageAssembler` 的两个对应的静态方法提供实现。

```

using System.ServiceModel;
using Artech.ImageTransfer.Service.Interface;
namespace Artech.ImageTransfer.Service
{
    [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
    public class ImageTransferService : IImageTransfer
    {
        public void Transfer(byte[] imageSlice)
        {
            ImageAssembler.ReceiveImageSlice(imageSlice);
        }

        public void Erase()
        {
            ImageAssembler.Erase();
        }
    }
}

```

步骤三：服务寄宿和图片接收程序实现

图片传输服务 ImageTransferService 最终被寄宿于一个 Windows Forms 应用中，该应用同时作为图片接收程序。下面是服务寄宿端的配置。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="nonReliableSession">
          <binaryMessageEncoding>
            <readerQuotas maxArrayLength="2147483647" />
          </binaryMessageEncoding>
          <tcpTransport maxBufferSize="2147483647"
            maxReceivedMessageSize="2147483647" />
        </binding>

        <binding name="reliableSession">
          <reliableSession ordered="false" />
          <binaryMessageEncoding>
            <readerQuotas maxArrayLength="2147483647" />
          </binaryMessageEncoding>
          <tcpTransport maxBufferSize="2147483647"
            maxReceivedMessageSize="2147483647" />
        </binding>

        <binding name="orderedDelivery">
          <reliableSession ordered="true" />
          <binaryMessageEncoding>
            <readerQuotas maxArrayLength="2147483647" />
          </binaryMessageEncoding>
          <tcpTransport maxBufferSize="2147483647"
            maxReceivedMessageSize="2147483647" />
        </binding>
      </customBinding>
    </bindings>
    <services>
      <service name="Artech.ImageTransfer.Service.ImageTransferService">
        <endpoint address="net.tcp://127.0.0.1:7777/imagetransferservice"
          binding="customBinding"
          bindingConfiguration="nonReliableSession"
          contract="Artech.ImageTransfer.Service.Interface.IImageTransfer" />
        <endpoint address="net.tcp://127.0.0.1:8888/imagetransferservice"
          binding="customBinding"
          bindingConfiguration="reliableSession"
          contract="Artech.ImageTransfer.Service.Interface.IImageTransfer" />
        <endpoint address="net.tcp://127.0.0.1:9999/imagetransferservice"
          binding="customBinding"
          bindingConfiguration="orderedDelivery"
          contract="Artech.ImageTransfer.Service.Interface.IImageTransfer" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

我们为 ImageTransferService 配置了基于自定义绑定的三个终结点，并且采用 TCP 传输方式和二进制消息编码。考虑到对较大尺寸图片的支持，将 BinaryMessageEncodingElement

的 `MaxArrayLength` 属性及 `TcpTransportElement` 的 `MaxBufferSize` 和 `MaxReceivedMessageSize` 都设置成最大。

对于这三个终结点的绑定配置，具有如下不一致的地方。`reliableSession` 和 `orderedDelivery` 终结点对应的绑定比 `nonReliableSession` 多了一个 `ReliableSessionElement` 绑定元素。相信你已经猜到了，`ReliableSessionElement` 是为了实现可靠会话而存在的。`reliableSession` 和 `orderedDelivery` 终结点绑定的 `ReliableSessionElement` 的 `Ordered` 属性分别为 `False` 和 `True`。也就是意味着 `orderedDelivery` 终结点能够实现对消息的有序交付，而 `reliableSession` 终结点则不能。

图片的接收窗口如图 5-3 所示，其中每一个方格是一个 `PictureBox`，用于显示接收到的图片切片。对于这些 `PictureBox` 的 ID，从上到下，从左到右依次是 `pictureBox11`、`pictureBox12`、...、`pictureBox15`、...、`pictureBox55`。

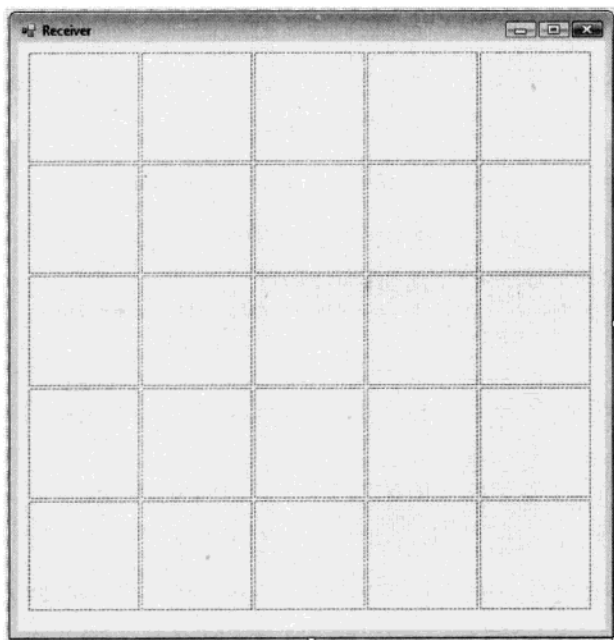


图 5-3 图片接收窗口

整个服务寄宿和图片接收实现在如下的代码中。`ImageAssembler_ImageClliceReceived` 方法将接收到的字节数组转换成位图，依次显示到上述的 25 个 `PictureBox` 上。在方法上面应用了一个 `MethodImplAttribute` 特性并指定 `MethodImplOptions.Synchronized` 作为参数，所以该方法是同步执行的。也就是说该方法处理的消息次序就是消息被交付的次序。

```
using System;
using System.Drawing;
using System.IO;
using System.Runtime.CompilerServices;
```

```

using System.ServiceModel;
using System.Threading;
using System.Windows.Forms;
namespace Artech.ImageTransfer.Service
{
    public partial class Recevier : Form
    {
        private PictureBox[] _pictureBoxes;
        private SynchronizationContext _synchronizationContext = null;
        private int _index = 0;
        private ServiceHost _serviceHost = null;
        public Recevier()
        {
            InitializeComponent();
            _pictureBoxes = new PictureBox[] {

this.pictureBox11, this.pictureBox12, this.pictureBox13, this.pictureBox14,
this.pictureBox15,
this.pictureBox21, this.pictureBox22, this.pictureBox23, this.pictureBox24,
this.pictureBox25,
this.pictureBox31, this.pictureBox32, this.pictureBox33, this.pictureBox34,
this.pictureBox35,
this.pictureBox41, this.pictureBox42, this.pictureBox43, this.pictureBox44,
this.pictureBox45,
this.pictureBox51, this.pictureBox52, this.pictureBox53, this.pictureBox54,
this.pictureBox55};
            ImageAssembler.ImageSliceReceived += ImageAssembler_Image
                CliceReceived;
            ImageAssembler.ImageErasing += ImageAssembler_ImageErasing;
        }
        [MethodImpl(MethodImplOptions.Synchronized)]
        private void ImageAssembler_ImageCliceReceived(object sender,
            ImageReceivedEventArgs args)
        {
            Bitmap bitmap = null;
            using (MemoryStream stream = new MemoryStream(args.ImageSlice))
            {
                bitmap = new Bitmap(stream);
                _synchronizationContext.Send(state => _pictureBoxes
                    [_index++].Image = bitmap, null);
            }
        }
        private void ImageAssembler_ImageErasing(object sender, EventArgs args)
        {
            _index = 0;
            foreach (var pictureBox in _pictureBoxes)
            {
                pictureBox.Image = null;
            }
        }
        private void Recevier_Load(object sender, EventArgs e)
        {
            _synchronizationContext = SynchronizationContext.Current;
            _serviceHost = new ServiceHost(typeof(ImageTransferService));
            _serviceHost.Open();
        }
    }
}

```


步骤四：创建图片发送程序

最后编写我们的图片发送端程序，即对图片进行切片，并通过调用图片传输服务对切片进行发送。下面照例先来看看 WCF 在客户端的配置。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="nonReliableSession">
          <binaryMessageEncoding>
            <readerQuotas maxArrayLength="2147483647" />
          </binaryMessageEncoding>
          <unreliableNetworkSimulate dropRate="10"/>
          <tcpTransport maxBufferSize="2147483647"
            maxReceivedMessageSize="2147483647" />
        </binding>

        <binding name="reliableSession">
          <reliableSession ordered="false" />
          <binaryMessageEncoding>
            <readerQuotas maxArrayLength="2147483647" />
          </binaryMessageEncoding>
          <unreliableNetworkSimulate dropRate="10"/>
          <tcpTransport maxBufferSize="2147483647"
            maxReceivedMessageSize="2147483647" />
        </binding>

        <binding name="orderedDelivery">
          <reliableSession ordered="true" />
          <binaryMessageEncoding>
            <readerQuotas maxArrayLength="2147483647" />
          </binaryMessageEncoding>
          <unreliableNetworkSimulate dropRate="10"/>
          <tcpTransport maxBufferSize="2147483647"
            maxReceivedMessageSize="2147483647" />
        </binding>
      </customBinding>
    </bindings>
    <client>
      <endpoint name="nonReliableSession"
        address="net.tcp://127.0.0.1:7777/
        imagetransferservice"
        binding="customBinding"
        bindingConfiguration="nonReliableSession"
        contract="Artech.ImageTransfer.Service.Interface.IImageTransfer"/>
      <endpoint name="reliableSession"
        address="net.tcp://127.0.0.1:8888/imagetransferservice"
        binding="customBinding"
        bindingConfiguration="reliableSession"
        contract="Artech.ImageTransfer.Service.Interface.IImageTransfer"/>
      <endpoint name="orderedDelivery"
        address="net.tcp://127.0.0.1:9999/imagetransferservice"
        binding="customBinding"
        bindingConfiguration="orderedDelivery"
        contract="Artech.ImageTransfer.Service.Interface.IImageTransfer"/>
    </client>
  </extensions>
</configuration>
```

```

        <bindingElementExtensions>
            <add name="unreliableNetworkSimulate"
                type="Artech.ImageTransfer.Extensions.UnreliableNetworkSimulate
                ExtensionElement, Artech.ImageTransfer.Extensions, Version=1.0.0.0,
                Culture=neutral, PublicKeyToken=null" />
            </bindingElementExtensions>
        </extensions>
    </system.serviceModel>
</configuration>

```

与服务寄宿端的配置一样，客户端也配置了三种自定义绑定。所不同的是，它们均多了一个额外的绑定元素 `UnreliableNetworkSimulateBindingElement`（丢包率被设置为 10%），即我们之前创建的用于模拟不稳定网络环境的绑定元素。

图 5-4 是图片发送窗口，上边部分是一个 `PictureBox`，会显示通过单击 `Browse` 按钮选择的图片。当成功选择某一张用于发送的图片后，单击 `Send` 按钮将其发送。`Reliable Session` 和 `Ordered Delivery` 两个 `CheckBox` 供用户决定是否采用可靠会话和有序交付机制进行图片的发送。默认情况下，并不采用可靠会话机制进行图片发送。图片的选择、切片和发送通过下面的代码实现。

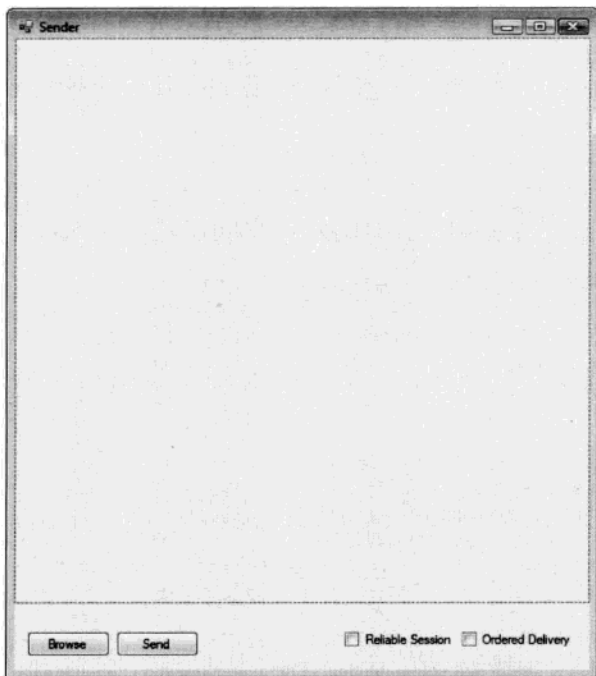


图 5-4 图片发送端窗口

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;

```

```

using System.ServiceModel;
using System.Windows.Forms;
using Artech.ImageTransfer.Service.Interface;
namespace Artech.ImageTransfer.Client
{
    public partial class Sender : Form
    {
        private string _imageSource = string.Empty;
        private IImageTransfer _nonReliableSessionProxy = null;
        private IImageTransfer _reliableSessionProxy = null;
        private IImageTransfer _orderedDeliveryProxy = null;
        ChannelFactory<IImageTransfer> _nonReliableSessionFactory =
            new ChannelFactory<IImageTransfer>("nonReliableSession");
        ChannelFactory<IImageTransfer> _reliableSessionFactory =
            new ChannelFactory<IImageTransfer>("reliableSession");
        ChannelFactory<IImageTransfer> _orderedDeliveryFactory =
            new ChannelFactory<IImageTransfer>("orderedDelivery");

        public Sender()
        {
            InitializeComponent();
        }

        private IImageTransfer GetProxy()
        {
            if (null != _nonReliableSessionProxy)
            {
                (_nonReliableSessionProxy as ICommunicationObject).Close();
            }
            if (null != _reliableSessionProxy)
            {
                (_reliableSessionProxy as ICommunicationObject).Close();
            }
            if (null != _orderedDeliveryProxy)
            {
                (_orderedDeliveryProxy as ICommunicationObject).Close();
            }
            if (!this.checkBoxReliableSession.Checked)
            {
                _nonReliableSessionProxy =
                    _nonReliableSessionFactory.CreateChannel();
                (_nonReliableSessionProxy as ICommunicationObject).Open();
                return _nonReliableSessionProxy;
            }
            else if (!this.checkBoxOrdered.Checked)
            {
                _reliableSessionProxy = _reliableSessionFactory.CreateChannel();
                (_reliableSessionProxy as ICommunicationObject).Open();
                return _reliableSessionProxy;
            }
            else
            {
                _orderedDeliveryProxy = _orderedDeliveryFactory.CreateChannel();
                (_orderedDeliveryProxy as ICommunicationObject).Open();
                return _orderedDeliveryProxy;
            }
        }
    }
}

```

```

    }
    private void buttonOpen_Click(object sender, EventArgs e)
    {
        OpenFileDialog openFileDialog = new OpenFileDialog();
        if (openFileDialog.ShowDialog() == DialogResult.OK)
        {
            _imageSource = openFileDialog.FileName;
            this.pictureBox1.Load(_imageSource);
        }
        this.buttonSend.Enabled = true;
    }
    private byte[] BitmapToBytes(Bitmap bitmap)
    {
        using (MemoryStream ms = new MemoryStream())
        {
            bitmap.Save(ms, ImageFormat.Bmp);
            byte[] data = new byte[ms.Length];
            ms.Seek(0, SeekOrigin.Begin);
            ms.Read(data, 0, Convert.ToInt32(ms.Length));
            return data;
        }
    }

    private void buttonSend_Click(object sender, EventArgs e)
    {
        this.buttonSend.Enabled = false;
        IList<byte[]> imageSlices = new List<byte[]>();
        Bitmap bmp = new Bitmap(this._imageSource);
        double width = (double)bmp.Width / 5;
        double height = (double)bmp.Height / 5;
        for (int y = 0; y < 5; y++)
        {
            for (int x = 0; x < 5; x++)
            {
                Rectangle rect = new Rectangle(Convert.ToInt32(x * width),
                    Convert.ToInt32(y * height), Convert.ToInt32(width),
                    Convert.ToInt32(height));
                byte[] data = BitmapToBytes(bmp.Clone(rect, PixelFormat.
                    DontCare));
                imageSlices.Add(data);
            }
        }
        IImageTransfer proxy = GetProxy();
        proxy.Erase();
        for (int i = 0; i < imageSlices.Count; i++)
        {
            proxy.Transfer(imageSlices[i]);
        }
        this.buttonSend.Enabled = true;
    }
}
}

```

我们通过 GetProxy()方法根据是否选择了“可靠会话”和“有序递交”复选框针对相应终结点对应的 ChannelFactory<IImageTransfer>对象。在 buttonSend_Click 方法中，被选择的

图片被均分成 25 个切片，并按照从上到下、从左至右的顺序通过调用 `BitmapToBytes` 方法转换成字节数据，最终利用创建的服务代理发送出去。在发送之前调用服务操作通知接收端擦除已经接收到的切片。

所有的编程工作完成后，运行我们的程序。图 5-5 表示的是没有采用可靠会话时的图片传输情况。从中可以看到两接收方组装后的图片不完整，有 4 个切片缺失。此外接收方组装后的切片完全是错位的。



图 5-5 没有采用可靠会话的图片传输情况

图 5-6 表示的是选择了可靠会话选项，但是没有选择有序交付选项时图片传输的情况。可以看出，这一次解决了切片丢失的问题，但是错位的情况依然存在。

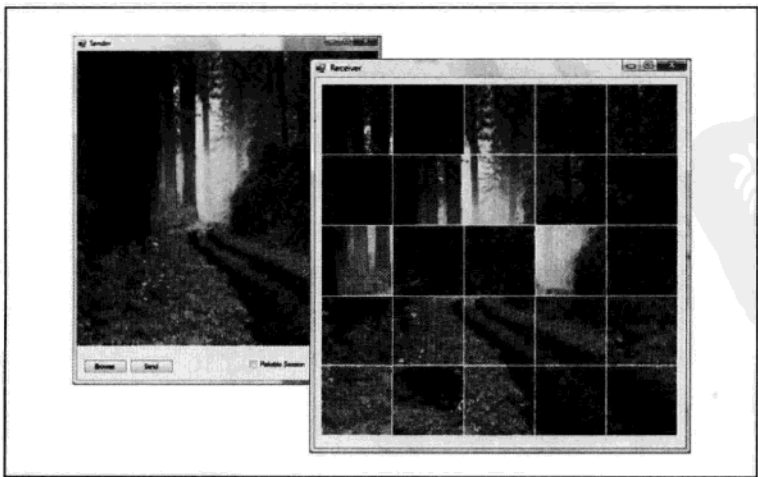


图 5-6 选择可靠会话但不选择有序交付时图片传输情况

最后我们同时选择有序交付选项, 在接收端将会得到一张完完整整的图片, 既不会有切片丢失, 也不会出现切片错位的情况。最终的结果如图 5-7 所示, 这才是我们希望的。



图 5-7 同时选择可靠会话和有序交付时的图片传输情况

5.2.2 可靠会话绑定

在上面给出的例子中, 实际上是通过对接点的绑定进行相应的配置让整个消息的交换过程在一个可靠会话中进行, 进而实现可靠消息传输的目的。由于整个可靠会话的机制是完全在信道层实现的, 所以可靠会话编程是围绕着绑定进行的。从结构组成的角度讲, 绑定本质上就是一组绑定元素的有序集合, 实现可靠会话的就是 `ReliableSessionBindingElement` 绑定元素。

1. 从 `ReliableSessionBindingElement` 谈起

WCF 的可靠会话是建立在客户端和服务端之间, 确保消息可靠传输的上下文, 相当于 WS-RM 中的序列。在消息发送端和接收端具有一个消息缓冲区 (或者称为消息窗口) 对消息进行缓存, 前者缓存已经发送但是尚未接收到确认的未决消息, 后者缓存尚未向上交付的消息。

消息在发送之前, 会被赋予一个特殊的 SOAP 报头, 其中包含表示消息在整个可靠会话生命周期内被发送的序号。消息的副本会保存到消息缓冲区中。消息被接收端成功接收之后, 会向发送端发送一个确认消息表示具有某个序号的消息已经成功接收。如果需要保障有序交付, 接收端在交付之前需要确定先于该消息发送的所有消息是否成功交付。如果是则实施交付, 否则将其放入消息缓冲区, 等待之前消息的抵达。当之前的所有消息被成功接收之后, 接收端按照消息序号从小到大的顺序对消息实施交付。缓存的消息被成功交付后, 会从缓冲区移除。

而消息发送端在接收到消息确认之后,会根据消息序号将对应的消息从缓冲区中冲移除。如果在限定的超时时限内没有接收到已发消息的确认,会认为该消息发送失败,该消息会从缓冲区中提取出来并重新发送。

WCF 中整个可靠会话的实现完全是通过 `ReliableSessionBindingElement` 这个绑定元素创建的信道实现的。下面的代码列出了 `ReliableSessionBindingElement` 主要的属性成员定义,而可靠会话实现的行为是受这些属性控制的。

```
public sealed class ReliableSessionBindingElement : BindingElement,
IPolicyExportExtension
{
    //其他成员
    public TimeSpan AcknowledgementInterval { get; set; }
    public bool FlowControlEnabled { get; set; }
    public TimeSpan InactivityTimeout { get; set; }
    public int MaxPendingChannels { get; set; }
    public int MaxRetryCount { get; set; }
    public int MaxTransferWindowSize { get; set; }
    public bool Ordered { get; set; }
    public ReliableMessagingVersion ReliableMessagingVersion { get; set; }
}
```

现在了解了 WCF 可靠会话大体机制的情况,下面来了解一下 `ReliableSessionBindingElement` 各个属性的含义。

- **AcknowledgementInterval**: 如果消息的发送方和接收方通过双工通道连接,则接收方能够随时向接收方发送确认。为了降低网络流量,WCF 采用批量确认的机制。当接收端成功接收到某个消息的时候,并不会立即针对该消息发送确认,而是等待一定时间后,对所有接收到的消息进行批量确认。`AcknowledgementInterval` 表示消息接收端发送确认之前等待的时间间隔,默认为 0.2 秒(200 毫秒)。该设置仅针对于 `NetTcpBinding` 和 `WSDualHttpBinding` 等支持双工通信的绑定有效,而像 `BasicHttpBinding` 和 `WSHttpBinding` 这样的绑定则无此设置。
- **FlowControlEnabled**: 该属性指示可靠会话是否已启用流控制(Flow Control)。流控制的目的是确保发送方所发送的消息数不超过接收方可处理的消息数。接收方拥有消息缓冲区,用于容纳突增的消息和无序的消息。接收方在每次确认时都会告知发送方此缓冲区中的剩余空间量。利用此信息,发送方就可以在接收方缓冲区中没有剩余空间时,停止发送新的消息。`FlowControlEnabled` 默认值为 `True`。
- **InactivityTimeout**: 在 WS-RM 中,被创建的 RM 序列具有一个 `Expires` 属性表示序列的生命周期,但是我们不能设置可靠会话的整个生命周期的时限。WCF 并不能完全依赖于可靠会话被显式地终止,而需要设定一个超时时限,在该时间范围内如果没有活动的消息交换,WCF 会将可靠会话关闭。`InactivityTimeout` 就是这么一个表示可靠会话在关闭之前保持活动状态的时间间隔,默认值为 10 分钟。

- **MaxPendingChannels**: 信道在等待被接收时处于挂起状态, 该值表示最大运行挂起的信道数量。一旦达到该限制, 就不会创建任何通道, 直到此数值降低。这是对每个侦听器的限制, 当达到此阈值时, 如果远程应用程序尝试建立新的可靠会话, 则会拒绝请求且打开操作将提示此错误。该属性的默认值为 4。
- **MaxRetryCount**: 如果在某个可接受时间范围内, 消息的发送端没有接收到某个已发消息的确认, 会对该消息进行重传。MaxRetryCount 表示重传的次数, 最小值为 1, 最大值为 0x7FFFFFFF (Int32.MaxValue), 默认值为 8。
- **MaxTransferWindowSize**: 该属性表示消息缓冲区的大小, 或者说缓冲区能够容纳消息的数量。MaxTransferWindowSize 最小值为 1, 最大值为 4096, 默认值为 32。MaxTransferWindowSize 属性的值可在发送方和接收方进行设置。如果达到发送方的这个限制, 则会阻止其他发送调用。如果达到接收方的这个限制, 则不会接受到达的新消息。
- **Ordered**: 该属性表示是否启用“有序传输”机制确保消息的接收端完全按照消息被发送的顺序进行交付。该属性的默认值为 False。
- **ReliableMessagingVersion**: 该属性表示可靠会话支持的 WS-RM 的版本。在 WCF 中, 该 WS-RM 通过 ReliableMessagingVersion 类型表示 (不是枚举)。ReliableMessagingVersion 定义如下, 可以通过两个静态只读属性 WSReliableMessagingFebruary2005 和 WSReliableMessaging11 得到两个表示 WS-RM 1.0 和 WS-RM 1.1 的 ReliableMessagingVersion 对象。静态只读属性 Default 表示默认的 WS-RM 版本, 其值目前和 WSReliableMessagingFebruary2005 属性一致。

```
public abstract class ReliableMessagingVersion
{
    public static ReliableMessagingVersion Default { get; }
    public static ReliableMessagingVersion WSReliableMessagingFebruary2005
    { get; }
    public static ReliableMessagingVersion WSReliableMessaging11 { get; }
}
```

2. 为系统绑定的可靠会话进行设置

在众多系统绑定中, 有很大一部分都为可靠会话提供支持 (比如 WSHttpBinding、WS2007HttpBinding、NetTcpBinding、WSFederationHttpBinding、WS2007FederationHttpBinding 和 WSDualHttpBinding)。这些绑定类型中均有一个名称为 ReliableSession 的属性, 属性类型为 ReliableSession 或其子类 OptionalReliableSession。下面的代码演示了 WSHttpBindingBase (WSHttpBinding 和 WS2007HttpBinding 的基类) 和 WSDualHttpBinding 中的 ReliableSession 属性的定义。

```
public abstract class WSHttpBindingBase : Binding, IBindingRuntimePreferences
{
```

```

    //其他成员
    public OptionalReliableSession ReliableSession { get; }
}
public class WSDualHttpBinding : Binding, IBindingRuntimePreferences
{
    //其他成员
    public ReliableSession ReliableSession { get; }
}

```

在讨论不同类型的系统绑定对可靠会话的支持之前，不妨先来看看 `ReliableSession` 和 `OptionalReliableSession` 的定义。从下面的定义可以看出 `ReliableSession` 仅具有两个可读写的 `InactivityTimeout` 和 `Ordered` 属性，它们和 `ReliableSessionBindingElement` 的同名属性相匹配（实际上整个 `ReliableSession` 对象是根据 `ReliableSessionBindingElement` 对象创建的）。`ReliableSession` 的子类 `OptionalReliableSession` 多了一个额外可读写的属性 `Enabled`，这是一个让绑定启用可靠会话的开关。

```

public class ReliableSession
{
    public ReliableSession(
        ReliableSessionBindingElement reliableSessionBindingElement);
    public TimeSpan InactivityTimeout { get; set; }
    public bool Ordered { get; set; }
}
public class OptionalReliableSession : ReliableSession
{
    public OptionalReliableSession(
        ReliableSessionBindingElement reliableSessionBindingElement);
    public bool Enabled { get; set; }
}

```

虽然在 `ReliableSessionBindingElement` 绑定元素中定义了众多控制可靠会话行为的属性，但是 `ReliableSession` 中仅为我们公布了其中两个（`InactivityTimeout` 和 `Ordered`）。潜在的信息告诉我们，对于这些支持可靠会话的系统绑定来说，只能设置可靠会话在关闭之前保持非活动状态的时间间隔和开启或关闭有序交付特性。其他选项，比如支持的 WS-RM 版本及消息缓冲区大小，都是系统为我们定制的，不能修改。

对于前面提到的若干支持可靠会话的系统绑定，除了 `WSDualHttpBinding` 的 `ReliableSession` 属性类型为 `ReliableSession` 外，其余的均为 `OptionalReliableSession`。也就是我们不能关闭 `WSDualHttpBinding` 的可靠会话特性，它总是按照可靠会话的机制进行消息的交换。WCF 之所以如此设计，是源于 `WSDualHttpBinding` 支持双工通信的特殊机制决定的。基于请求-回复模式的 HTTP 传输不能够独立提供对双工通信的支持，WCF 采用的是双通道的方式，即 `WSDualHttpBinding` 创建的所谓的双工通道是由两个方向相反的 HTTP 连接组成的，而 WCF 需要采用可靠会话机制提供对这两个连接的匹配。

除了 `InactivityTimeout` 和 `Ordered` 两个属性可以进行设置之外，定义在 `ReliableSessionBindingElement` 绑定元素中的各个属性大多采用默认值。但是有一个例外，即表示支持 WS-RM 版本的 `ReliableMessagingVersion` 属性。对于 `WSHttpBinding`、`WSDualHttpBinding`

和 WSFederationHttpBinding 支持的版本是 WS-RM 1.0, 而 WS2007HttpBinding 和 WS2007FederationHttpBinding 则支持的是 WS-RM 1.1。

可以通过编程的方式开启或者关闭终结点使用的除 WSDualHttpBinding 之外的其他系统绑定 (限于支持可靠会话系统绑定) 的可靠会话开关, 设置 InactivityTimeout 和 Ordered 属性。不过最好还是采用配置的方式对可靠会话进行设置。可靠会话相关配置定义在具体绑定配置中的 reliableSession 节点中。下面的配置中, 在客户端对终结点使用的 WS2007HttpBinding 的可靠会话进行了设置。

```
<system.serviceModel>
  <bindings>
    <ws2007HttpBinding>
      <binding name="reliableSession2007Binding">
        <reliableSession enabled="True"
          inactivityTimeout="00:20:00"
          ordered="True"/>
        ...
      </binding>
    </ws2007HttpBinding>
  </bindings>
  ...
</system.serviceModel>
```

3. 为自定义绑定的可靠会话进行设置

对于可靠会话, 如果采用系统绑定, 定制的范围其实很窄 (仅限于 InactivityTimeout 和 Ordered 属性)。但是如果采用自定义绑定, 由于操作的对象就是 ReliableSessionBindingElement 绑定元素, 因此可以对所有的选项进行自由配置。

为了能够让读者了解某个特性的配置, 我个人觉得最好的办法就是直接让读者看看相关配置节的定义。WCF 将 ReliableSessionBindingElement 的配置定义在具有如下定义的类型 System.ServiceModel.Configuration.ReliableSessionElement 中。通过 ReliableSessionElement, 不但可以了解可靠会话相关的配置属性, 还可以了解到它们的最大值、最小值和默认值等。

```
public sealed class ReliableSessionElement : BindingElementExtensionElement
{
    //其他成员
    [ConfigurationProperty("acknowledgementInterval", DefaultValue =
        "00:00:00.2")]
    public TimeSpan AcknowledgementInterval { get; set; }
    [ConfigurationProperty("flowControlEnabled", DefaultValue = true)]
    public bool FlowControlEnabled { get; set; }
    [ConfigurationProperty("inactivityTimeout", DefaultValue = "00:10:00")]
    public TimeSpan InactivityTimeout { get; set; }
    [ConfigurationProperty("maxPendingChannels", DefaultValue = 4)]
    public int MaxPendingChannels { get; set; }
    [ConfigurationProperty("maxRetryCount", DefaultValue = 8)]
    public int MaxRetryCount { get; set; }
    [ConfigurationProperty("maxTransferWindowSize", DefaultValue = 8)]
    public int MaxTransferWindowSize { get; set; }
```

```

[ConfigurationProperty("ordered", DefaultValue = true)]
public bool Ordered { get; set; }
[ConfigurationProperty("reliableMessagingVersion",
DefaultValue = "WSReliableMessagingFebruary2005")]
public ReliableMessagingVersion ReliableMessagingVersion { get; set; }
}

```

在对自定义绑定进行配置的时候，只需要在绑定元素集合中添加 **ReliableSessionElement** 配置元素，并对相应的配置属性进行设置即可。下面的配置中定义了一个自定义绑定。该绑定由三个绑定元素组成（**TextMessageEncodingElement**、**HttpTransportBindingElement** 和 **ReliableSessionBindingElement**）。在 `<reliableSession>` 配置节点中，我对所有的属性进行了显式设置。

```

<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="reliableSessionBinding">
          <textMessageEncoding />
          <reliableSession acknowledgementInterval="00:00:02"
            flowControlEnabled="false"
            inactivityTimeout="00:20:00"
            maxPendingChannels="8" maxRetryCount="4"
            maxTransferWindowSize="16"
            reliableMessagingVersion="WSReliable
            Messaging11"/>
          <httpTransport />
        </binding>
      </customBinding>
    </bindings>
    ...
  </system.serviceModel>
</configuration>

```

一个绑定可以由一系列绑定元素构成，那么 **ReliableSessionBindingElement** 应该置于何处呢？要搞清楚这个问题，需要对 WCF 的绑定模型有一个大致的了解。绑定旨在创建一个用于处理和传输消息的信道栈，信道在信道栈的顺序决定于对应的绑定元素的排列顺序。可靠会话将客户端和服务端通过 **ReliableSessionBindingElement** 创建的可靠信道作为分界线，并在它们之间提供消息可靠传输保障。

也就是说，信道栈中位于可靠信道之下（靠近传输信道）部分存在于可靠会话的范围之中，而上面部分则脱离可靠会话的管辖范围。这也是在上面给出的实例中，我们将用于模拟不可靠网络的绑定元素配置到 **ReliableSessionBindingElement** 之下的原因所在。

由于在实际的应用中，主要通过可靠会话为网络传输的不稳定性提供可靠传输的保障，因此一般将 **ReliableSessionBindingElement** 配置到传输绑定元素之上。至于消息编码绑定元素，置于 **ReliableSessionBindingElement** 之上或者之下均没有关系。如果你认真阅读过上册的第 3 章“绑定（Binding）”，会知道消息编码绑定元素并不参与信道的创建，而是将编码的方式传入绑定上下文。传输信道据此采用相应的编码方式进行消息的编码或者解码。

4. 通过 DeliveryRequirementsAttribute 对可靠会话进行强约束

如下面的代码所示, `System.ServiceModel.DeliveryRequirementsAttribute` 特性实际上是一个契约行为。我们可以将其应用到服务契约类型或者服务类型上, 强制要求相应终结点绑定必须满足设定的关于消息传输方面的要求。

```
[AttributeUsage(AttributeTargets.Interface | AttributeTargets.Class,
    AllowMultiple=true)]
public sealed class DeliveryRequirementsAttribute : Attribute,
    IContractBehavior, IContractBehaviorAttribute
{
    //其他成员
    public QueuedDeliveryRequirementsMode QueuedDeliveryRequirements { get;
    set; }
    public bool RequireOrderedDelivery { get; set; }
    public Type TargetContract { get; set; }
}
```

`DeliveryRequirementsAttribute` 定义了 `QueuedDeliveryRequirements` 和 `RequireOrderedDelivery` 两个属性, 分别代表相应的终结点绑定必须满足的两个要求, 即队列传递和有序交付。队列传递即采用消息队列 (即 MSMQ) 的机制进行消息传递, 我们将在本书第6章“队列服务 (Queued Service)”中对队列服务进行单独介绍。`TargetContract` 属性是对 `IContractBehaviorAttribute` 的实现, 当我们将 `DeliveryRequirementsAttribute` 特性应用到某个实现了多个服务契约的服务上时, 可以指定设置的消息传递要求是针对某个服务契约。

如果寄宿服务的终结点绑定无法满足通过 `DeliveryRequirementsAttribute` 特性 (应用在服务契约和服务类型) 定义的有序交付的要求时, 会抛出一个 `InvalidOperationException` 异常。如果将 `DeliveryRequirementsAttribute` 特性应用到服务契约上, 客户端同样会验证用于服务调用的终结点绑定是否满足相应的要求。如果无法满足, 同样会抛出一个 `InvalidOperationException` 异常。

举个例子, 假设我们定义如下一个 `IOrderService` 服务契约用于处理订单。在 `IOrderService` 接口上, 应用了 `DeliveryRequirementsAttribute` 特性并将 `RequireOrderedDelivery` 设置成 `True`, 要求终结点绑定强制开启可靠消息的有序交付特性。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
[DeliveryRequirements(RequireOrderedDelivery = true)]
public interface IOrderService
{
    [OperationContract]
    void Process(Order order);
}
```

现在采用如下的配置对实现该服务契约的服务进行寄宿。从中可以看到, 我们采用 `WS2007HttpBinding` 作为终结点的绑定。通过绑定配置, 开启了可靠会话, 但是将 `ordered` 属性配置成 `False`。

```

<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="reliableSessionHttpBinding">
          <reliableSession          ordered="false"
                                   enabled="true" />
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <services>
      <service name="Artech.OrderService">
        <endpoint address="http://127.0.0.1:3721/OrderService"
                  binding="ws2007HttpBinding"
                  bindingConfiguration="reliableSessionHttpBinding"
                  contract="Artech.IOrderService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

当进行服务寄宿的时候,会得到如图 5-8 所示的 `InvalidOperationException` 异常。当进一步看清具体的异常消息的时候,你的第一感觉可能就是作者把图片弄错了。因为终结点绑定不满足 `DeliveryRequirementsAttribute` 设定的关于有序交付的要求,和队列传递根本就不相关。但是图 5-8 就是真实运行后的截图,这是 WCF 自身的一个 Bug。

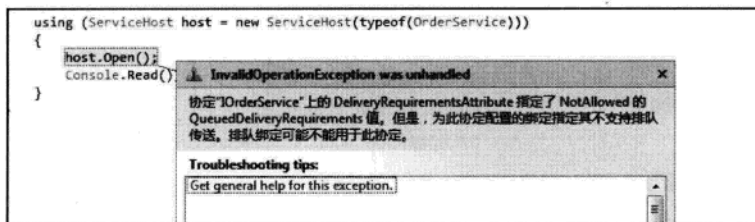


图 5-8 终结点绑定不能满足 `DeliveryRequirementsAttribute` 设定的要求导致的异常

5.3 可靠会话的实现原理

从上一节针对可靠会话编程的介绍中,不难看出可靠会话的编程仅围绕着一个对象,那就是绑定。对可靠会话的实现,是完全在信道层实现的。

5.3.1 从信道层看可靠会话的实现

绑定是一系列绑定元素的有序组合,而实现可靠会话的是一个叫做 `ReliableSessionBindingElement` 的绑定元素。绑定元素的主要任务是用于创建信道管理器(信道工厂和信道监听器)。由所有信道工厂和信道监听器创建的信道按照其创建者的顺序构建起一个消息处理的信道栈。

WCF 最终通过 `ReliableSessionBindingElement` 创建可靠会话信道 (Reliable Session Channel, 以下简称 RS 信道), 提供对可靠消息传输的实现。在 WS-RM 定义的可靠消息传输模型中, 可靠消息传输是在 RM 源和 RM 目的地之间进行的, 在这里可以将客户端和服务端的 RS 信道看成 RM 源和 RM 目的地。

1. 可靠会话信道层模型

图 5-9 反映的是可靠会话在信道层的实现模型, 从中可以看出可靠会话建立在客户端和服务端的 RS 信道之间。作为客户端或者服务端信道栈中的一员, RS 信道在信道栈中位置由 `ReliableSessionBindingElement` 在绑定元素集合中的位置决定。WCF 中的可靠消息传输保障仅存在于客户端和服务端的 RS 信道之间。这其中不仅包括存在于客户端和服务端之间的传输网络, 也包括存在于可靠会话信道之下的所有信道。

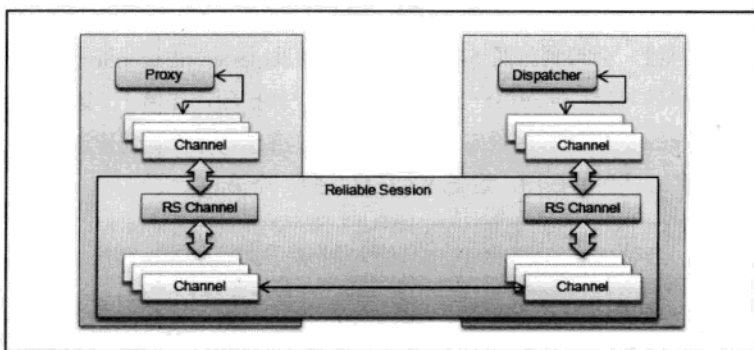


图 5-9 可靠会话信道层模型

WS-RM 为可靠消息传输的体现定义了一个可扩展的消息交换模型, 而 WCF 的可靠会话是对该模型具体的实现。接下来看看 WCF 的可靠会话是如何实现定义在 WS-RM 中的每一个消息交换步骤的。WCF 目前支持 WS-RM 1.0 和 1.1 两个版本, 在这里我们基于的是 WS-RM 1.1。

2. CreateSequence 和 CreateSequenceResponse

基于 WS-RM 的可靠消息传输是在一个 RM 序列中进行的, RM 序列为实现可靠消息传输提供了一个上下文环境。WCF 中与 RM 序列对应的概念就是可靠会话。基于 WS-RM 可靠消息传输从 RM 序列的创建开始, 对于 WCF 来说实现可靠消息传输需要先创建可靠会话。WCF 的可靠会话创建于可靠信道开启之时。

WS-RM 中序列创建过程是从 RM 源向 RM 目的地发送 `CreateSequence` 请求开始, 到接收到对方返回 `CreateSequenceResponse` 回复作为结束。对于可靠会话来说, 客户端和服务端信道栈中的 RS 信道充当 RM 源和 RM 目的地的角色。当客户端 RS 信道开启的时候, 它会

创建 CreateSequence 消息，并沿着信道栈路径发送到服务端。该 CreateSequence 消息被服务端信道栈接收并最终递交给 RS 信道后，RS 负责创建 RM 序列。

序列创建成功后，可靠会话上下文在服务端部分被成功创建，被创建的 RM 序列被封装到 CreateSequenceResponse 消息中返回到客户端。当客户端 RS 信道接收到 CreateSequenceResponse 消息后，在客户端部分创建可靠会话上下文，至此有两个 RS 信道的客户端会话被创建出来。

WS-RM 中某个 RM 序列只能保证单向的消息传输的可靠性。要想解决双向可靠消息传输，需要借助于两个 RM 序列。所以对于请求-回复模式和双工模式下的可靠消息传输需要双 RM 序列的支持。WS-RM 通过序列提供机制 (Sequence Offering) 对此提供支持。接下来讨论 WCF 的可靠会话对 WS-RM 序列提供机制的实现。

在客户端 RS 信道开启时，RS 信道会先检测当前终结点服务契约中所有服务操作采用的消息交换模式。如果所有操作均采用单向消息交换模式，RS 将不会采用序列提供机制。表现在消息交换上面，就意味着 CreateSequence 消息不会包含 Offer 元素。如果服务契约包含任何一个非单向操作，RS 信道会在客户端创建入栈序列 (Inbound Sequence)，并将其作为提供序列封装在 CreateSequence 消息的 <Offer> 元素中。下面的 XML 片段就是这样一个包含提供序列的 CreateSequence 消息。需要注意的是，在 RS 信道生成的 CreateSequence 消息中，Offer/Endpoint 和 AcksTo 的终结点引用是相同的。

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">
      http://docs.oasis-open.org/ws-rx/wsrn/200702/CreateSequence
    </a:Action>
    <a:MessageID>urn:uuid:f41d4443-fa5c-4f9d-95ff-96159c96ebec</a:MessageID>
    <a:To s:mustUnderstand="1">http://www.artech.com/calculatorservice</a:To>
  </s:Header>
  <s:Body>
    <CreateSequence xmlns="http://docs.oasis-open.org/ws-rx/wsrn/200702">
      <AcksTo>
        <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
      </AcksTo>
      <Offer>
        <Identifier>urn:uuid:25b6383f-b5a1-4839-8249-b0f273a7f502</Identifier>
        <Endpoint>
          <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
        </Endpoint>
        <IncompleteSequenceBehavior>
          DiscardFollowingFirstGap
        </IncompleteSequenceBehavior>
      </Offer>
    </CreateSequence>
  </s:Body>
</s:Envelope>
```

当包含 <Offer> 元素的 CreateSequence 消息被服务端 RS 信道成功接收到之后，会分析该

元素封装的由客户端提供的 RM 序列是否满足要求。如果满足则选择“接受”该序列。服务端接受客户端提供的序列，体现在服务端 RS 信道会在 CreateSequenceResponse 消息中添加如下一个<Accept>元素。

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">
      http://docs.oasis-open.org/ws-rx/wsrn/200702/CreateSequenceResponse
    </a:Action>
    <a:RelatesTo>urn:uuid:f41d4443-fa5c-4f9d-95ff-96159c96ebec</a:RelatesTo>
  </s:Header>
  <s:Body>
    <CreateSequenceResponse xmlns="http://docs.oasis-open.org/ws-rx/wsrn/200702">
      <Identifier>urn:uuid:97182c7f-ca5a-4c5f-8e17-6509387fa8bf</Identifier>
      <IncompleteSequenceBehavior>
        DiscardFollowingFirstGap
      </IncompleteSequenceBehavior>
      <Accept>
        <AcksTo>
          <a:Address>http://www.artech.com/calculatorservice</a:Address>
        </AcksTo>
      </Accept>
    </CreateSequenceResponse>
  </s:Body>
</s:Envelope>
```

3. Sequence 和 SequenceAcknowledgement

当 RM 会话成功建立起来之后，相当于在客户端和服务端的 RS 信道之间建立起了一个（对于所有操作均是单向的情况）或者两个（操作列表中具有至少一个非单向的操作）RM 序列。此后当应用级别的消息通过传入发送端（可能是客户端，也可能是服务端）信道栈抵达 RS 信道的时候，RS 信道会为之添加一个基于 WS-RM 的<Sequence>报头。

同 ASP.NET 的会话一样，WCF 中的可靠会话实际上也可以看成是一种状态保持机制，它将客户端的服务调用请求关联到 RM 序列这样一个上下文中。可靠会话不但保持着创建的 RM 序列的标识，还保持一个计数器保存在会话生命周期内发送出的消息数量，该消息数量也就是消息的序号。

在发送端 RS 信道为传入消息添加的<Sequence>报头中，按照 WS-RM 的规定包含 RM 序列的标识和消息的序号。下面的 XML 片段展示的就是我们熟悉的计算服务调用请求经过客户端 RS 信道后整个请求消息的结构。

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:r="http://docs.oasis-open.org/ws-rx/wsrn/200702"
  xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <r:Sequence s:mustUnderstand="1">
      <r:Identifier>urn:uuid:2052a548-1505-4635-8995-a7c7e1e47379</r:Identifier>
      <r:MessageNumber>1</r:MessageNumber>
    </r:Sequence>
  </s:Header>
```



```

</r:Sequence>
<a:Action s:mustUnderstand="1">
  http://www.artech.com/ICalculator/Add
</a:Action>
<a:MessageID>urn:uuid:eacdc55d-eabf-4066-a958-5cc6753f1fe0</a:MessageID>
<a:ReplyTo>
  <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
</a:ReplyTo>
<a:To s:mustUnderstand="1">http://www.artech.com/calculatorservice</a:To>
</s:Header>
<s:Body>
  <Add xmlns="http://www.artech.com/">
    <x>1</x>
    <y>2</y>
  </Add>
</s:Body>
</s:Envelope>

```

当包含 Sequence 报头的消息被接收端信道栈接收并将其递交给 RS 信道时，RS 信道负责对接收到的消息进行确认。WCF 可靠会话采用两种不同的确认机制，我个人将这两种确认机制命名为“单独确认”和“背负（Piggy-Back）确认”。

对于前者，接收端 RS 信道会创建一个空的消息，并添加相应的确认报头。而对于后者，添加的确认报头直接将其放置到另一个消息中，这个消息可以是应用相关的，也可以是应用无关的（比如关闭、终止序列的消息），甚至可以是错误消息。无论采用怎样的确认机制，接收端 RS 信道都会在确认消息中添加<SequenceAcknowledgement>报头，并指定 RM 序列标识和确认消息序号范围。

一般来说，对于单向服务操作调用请求或者回调采用单独确认机制，而对于基于请求-回复模式的服务调用或者回调，会采用背负确认机制。单独确认机制很简单，在这里我们主要介绍背负确认。以请求-回复为例，假设在可靠会话情况下客户端通过如下的代码进行两次服务调用。

```

using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorservice"))
{
    ICalculator proxy = channelFactory.CreateChannel();
    (proxy as ICommunicationObject).Open();
    proxy.Add(1, 2);
    proxy.Add(1, 2);
    (proxy as ICommunicationObject).Close();
}

```

当第一次调用 Add 方法的时候，服务端会接收到如上面的 XML 所示的包含有 Sequence 报头的请求消息。当服务端 RS 接收到该请求时，并不会立即对其进行确认，而是利用回复消息进行确认。具体地说，当请求消息被分发给服务模型层并成功执行后，执行后的结果被封装成回复消息。当回复消息传入信道层后会被 RS 信道接收，此时它会将<SequenceAcknowledgement>报头添加到回复消息中。下面的 XML 片段展示了客户端最终接收到的回复消息。

```

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:r="http://docs.oasis-open.org/ws-rx/wsrn/200702"
            xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <r:Sequence s:mustUnderstand="1">
      <r:Identifier>urn:uuid:aeb0849b-cca9-4fc5-bdf4-06b6d4f2109b</r:Identifier>
      <r:MessageNumber>1</r:MessageNumber>
    </r:Sequence>
    <r:SequenceAcknowledgement>
      <r:Identifier>urn:uuid:2052a548-1505-4635-8995-a7c7ele47379</r:Identifier>
      <r:AcknowledgementRange Lower="1" Upper="1"/>
      <netrm:BufferRemaining
        xmlns:netrm="http://schemas.microsoft.com/ws/2006/05/rm">8
      </netrm:BufferRemaining>
    </r:SequenceAcknowledgement>
    <a:Action s:mustUnderstand="1">
      http://www.artech.com/ICalculator/AddResponse
    </a:Action>
    <a:RelatesTo>urn:uuid:eacdc55d-eabf-4066-a958-5cc6753f1fe0</a:RelatesTo>
  </s:Header>
  <s:Body>
    <AddResponse xmlns="http://www.artech.com/">
      <AddResult>3</AddResult>
    </AddResponse>
  </s:Body>
</s:Envelope>

```

从上面的 XML 可以看到, 回复消息的<SequenceAcknowledgement>报头正是对请求消息的确认, 消息序号范围是从 1 到 1 (Lower="1" Upper="1")。除了<SequenceAcknowledgement>报头之外, 回复消息同样具有一个<Sequence>报头。因为可靠会话不仅保障请求消息的可靠传输, 同样需要为回复消息的可靠传输提供保障。前面已经讨论过了, 可靠会话通过对 WS-RM 序列提供机制的实现, 帮助实现消息传输的双向保障。回复消息的<Sequence>报头包含客户端提供的 RM 序列标识和消息在该序列中的序号。那么回复消息又是如何被确认的呢?

第一次请求的回复是通过第二次服务调用的请求消息进行确认的。当通过相同的服务代理第二次调用 Add 方法的时候, 客户端 RS 信道会在请求消息上面添加如下一个<SequenceAcknowledgement>报头作为对上一次回复消息的接收确认。

```

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:r="http://docs.oasis-open.org/ws-rx/wsrn/200702"
            xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <r:SequenceAcknowledgement>
      <r:Identifier>urn:uuid:dc416e9a-2646-44ef-a289-0c008a2a3ba0</r:Identifier>
      <r:AcknowledgementRange Lower="1" Upper="1"/>
    </r:SequenceAcknowledgement>
    <r:Sequence s:mustUnderstand="1">
      <r:Identifier>urn:uuid:fda625fa-9db4-46c4-9688-76ae3bc288ea</r:Identifier>
      <r:MessageNumber>2</r:MessageNumber>
    </r:Sequence>
    <a:Action s:mustUnderstand="1">
      http://www.artech.com/ICalculator/Add
    </a:Action>
  </s:Header>
  <s:Body>
    <Add>
      <AddResult>3</AddResult>
    </Add>
  </s:Body>
</s:Envelope>

```

```

<a:MessageID>urn:uuid:e79bcca9-ccc8-4e63-b893-22fc5adeb6d3</a:MessageID>
<a:ReplyTo>
  <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
</a:ReplyTo>
<a:To s:mustUnderstand="1">http://www.artech.com/calculatorservice</a:To>
</s:Header>
<s:Body>
  <Add xmlns="http://www.artech.com/">
    <x>1</x>
    <y>2</y>
  </Add>
</s:Body>
</s:Envelope>

```

和第一次服务调用的请求消息进行对比，第二次服务调用请求消息多了一个 `<AcknowledgementRange>` 报头实现对上一次回复消息的确认。你也应该想到第二次服务调用请求会在本次回复消息中被确认。但是最后一次服务调用的回复消息如何被确认呢？

在前面给出的服务调用代码中，在进行第二次服务调用之后服务代理就被关闭了。第二次服务调用的回复消息貌似没有被确认的机会了。实则不然，当关闭服务代理的时候，客户端 RS 信道会向服务端发送一个 `CloseSequence` 消息请求关闭当前的 RM 序列。而该 `CloseSequence` 消息包含 `<SequenceAcknowledgement>` 报头实现对客户端接收到的所有回复消息的确认。整个消息发送和确认的过程如图 5-10 所示。

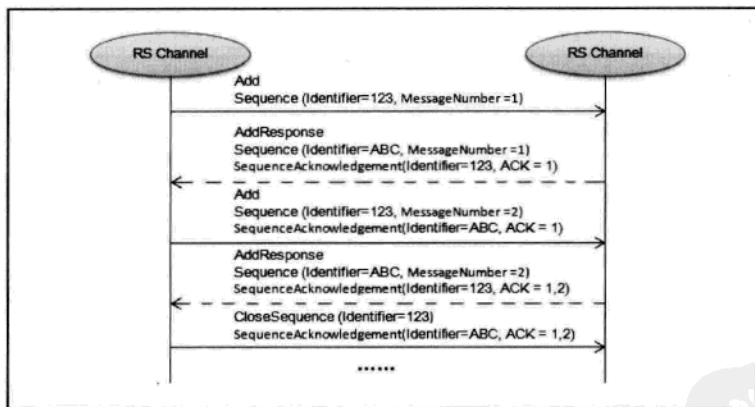


图 5-10 请求-回复模式下的消息确认实现

4. CloseSequence 和 CloseSequenceResponse

当基于某个服务代理的所有服务调用结束后，客户端程序应该关闭该代理。在开启可靠会话的情况下，服务代理的关闭同时意味着对可靠会话的终止 (Termination)。反映在 WS-RM 上就是对 RM 序列的终止。在 RM 序列终止之前，还有一个额外的过程，即 RM 序列的关闭。

当客户端 RS 信道被关闭时，它负责关闭可靠会话。它会按照 WS-RM 规范创建 `CloseSequence` 消息。不过在发送 `CloseSequence` 消息之前，RS 信道会等待所有已发消息的

确认均已成功接收。如果在可靠会话生命周期内有消息发送, CloseSequence 消息中还会包含最后一个消息的序号。

如果此时还有回复消息或者回调消息等待确认, CloseSequence 消息还包含对回复消息或者回调消息进行确认的<SequenceAcknowledgement>报头。在该<SequenceAcknowledgement>报头中具有一个<Final>元素表明这是最后一个确认。下面的 XML 片段展示了客户端 RS 信道生成的 CloseSequence 消息。

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:r="http://docs.oasis-open.org/ws-rx/wsrn/200702"
  xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <r:SequenceAcknowledgement>
      <r:Identifier>urn:uuid:d3e23ddf-7b6d-474f-9171-78bb76f4f977</r:Identifier>
      <r:AcknowledgementRange Lower="1" Upper="2"/>
      <r:Final/>
    </r:SequenceAcknowledgement>
    <a:Action s:mustUnderstand="1">
      http://docs.oasis-open.org/ws-rx/wsrn/200702/CloseSequence
    </a:Action>
    <a:MessageID>urn:uuid:fa99c2cf-4910-4598-a8ac-0e5b73787cc8</a:MessageID>
    <a:To s:mustUnderstand="1">http://www.artech.com/calculatorservice</a:To>
  </s:Header>
  <s:Body>
    <r:CloseSequence>
      <r:Identifier>urn:uuid:04127209-14ce-4648-90a5-a8eca6006fd2</r:Identifier>
      <r:LastMsgNumber>2</r:LastMsgNumber>
    </r:CloseSequence>
  </s:Body>
</s:Envelope>
```

有一点需要提醒读者的是, WS-RM 规定 RM 源和 RM 目的地具有可以在某个时刻向对方发送对现有 RM 序列关闭或者终止的请求。但是 WCF 仅对基于 RM 源序列终止或者关闭请求提供支持, 也就是只有客户端的 RS 信道才能主动请求终止目前的可靠会话。

CloseSequence 请求被服务端的 RS 信道成功接收之后, 它同样按照 WS-RM 规范生成 CloseSequenceReponse 消息回复给客户端。CloseSequenceReponse 消息同样包含一个<SequenceAcknowledgement>报头提供对所有接收到的消息的确认。和 CloseSequence 消息一样, 该<SequenceAcknowledgement>报头包含一个<Final>元素表示这是最后一个确认。下面的 XML 片段展示了服务端 RS 生成的 CloseSequenceReponse 消息。

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:r="http://docs.oasis-open.org/ws-rx/wsrn/200702"
  xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <r:SequenceAcknowledgement>
      <r:Identifier>urn:uuid:04127209-14ce-4648-90a5-a8eca6006fd2</r:Identifier>
      <r:AcknowledgementRange Lower="1" Upper="2"/>
      <r:Final/>
      <netrm:BufferRemaining
        xmlns:netrm="http://schemas.microsoft.com/ws/2006/05/rm">8
      </netrm:BufferRemaining>
    </r:SequenceAcknowledgement>
  </s:Header>
  <s:Body>
    <r:CloseSequenceReponse>
      <r:Identifier>urn:uuid:04127209-14ce-4648-90a5-a8eca6006fd2</r:Identifier>
      <r:LastMsgNumber>2</r:LastMsgNumber>
    </r:CloseSequenceReponse>
  </s:Body>
</s:Envelope>
```

```

</r:SequenceAcknowledgement>
<a:Action s:mustUnderstand="1">
  http://docs.oasis-open.org/ws-rx/wsrn/200702/CloseSequenceResponse
</a:Action>
<a:RelatesTo>urn:uuid:fa99c2cf-4910-4598-a8ac-0e5b73787cc8</a:RelatesTo>
</s:Header>
<s:Body>
  <r:CloseSequenceResponse>
    <r:Identifier>urn:uuid:04127209-14ce-4648-90a5-a8eca6006fd2</r:Identifier>
  </r:CloseSequenceResponse>
</s:Body>
</s:Envelope>

```

5. TerminateSequence 和 TerminateSequenceResponse

当客户端和服务端的 RS 信道完成了 CloseSequence/CloseSequenceResponse 握手之后, 开始为可靠会话的终止展开新一轮的对话。可靠会话的终止从客户端 RS 信道向对方发送 RM 序列终止请求开始。客户端 RS 信道按照 WS-RM 规范生成一个 TerminateSequence 请求发送给服务端。

一般来说, TerminateSequence 消息也会包含与 CloseSequence 消息一致的<SequenceAcknowledgement>报头。如果在可靠会话生命周期内有过消息发送, TerminateSequence 消息中还应该包含最后一个发送消息的序号, 并且该序号与 CloseSequence 消息中的一致。下面是一个由 WCF 客户端 RS 信道生成的 TerminateSequence 消息。

```

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:r="http://docs.oasis-open.org/ws-rx/wsrn/200702"
  xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <r:SequenceAcknowledgement>
      <r:Identifier>urn:uuid:d3e23ddf-7b6d-474f-9171-78bb76f4f977</r:Identifier>
      <r:AcknowledgementRange Lower="1" Upper="2"/>
      <r:Final/>
    </r:SequenceAcknowledgement>
    <a:Action s:mustUnderstand="1">
      http://docs.oasis-open.org/ws-rx/wsrn/200702/TerminateSequence
    </a:Action>
    <a:MessageID>urn:uuid:651a09f7-795a-4331-9faf-1856f0975b47</a:MessageID>
    <a:To s:mustUnderstand="1">http://www.artech.com/calculator-service</a:To>
  </s:Header>
  <s:Body>
    <r:TerminateSequence>
      <r:Identifier>urn:uuid:04127209-14ce-4648-90a5-a8eca6006fd2</r:Identifier>
      <r:LastMsgNumber>2</r:LastMsgNumber>
    </r:TerminateSequence>
  </s:Body>
</s:Envelope>

```

当服务端 RS 信道接收到 TerminateSequence 消息后, 会按照 WS-RM 规范生成 TerminateSequenceResponse 回复并返回给客户端。一般来说, TerminateSequenceResponse 消息具有与 CloseSequenceResponse 消息一样的<SequenceAcknowledgement>报头。下面就是一个通过服务端 RS 信道生成的 TerminateSequenceResponse 消息。

```

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:r="http://docs.oasis-open.org/ws-rx/wsrn/200702"
  xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <r:SequenceAcknowledgement>
      <r:Identifier>urn:uuid:04127209-14ce-4648-90a5-a8eca6006fd2</r:Identifier>
      <r:AcknowledgementRange Lower="1" Upper="2"/>
      <r:Final/>
      <netrm:BufferRemaining
        xmlns:netrm="http://schemas.microsoft.com/ws/2006/05/rm">8
      </netrm:BufferRemaining>
    </r:SequenceAcknowledgement>
    <a:Action s:mustUnderstand="1">
      http://docs.oasis-open.org/ws-rx/wsrn/200702/TerminateSequenceResponse
    </a:Action>
    <a:RelatesTo>urn:uuid:651a09f7-795a-4331-9faf-1856f0975b47</a:RelatesTo>
  </s:Header>
  <s:Body>
    <r:TerminateSequenceResponse>
      <r:Identifier>urn:uuid:04127209-14ce-4648-90a5-a8eca6006fd2</r:Identifier>
    </r:TerminateSequenceResponse>
  </s:Body>
</s:Envelope>

```

6. 可靠会话如何实现流控制 (Flow Control)

接下来谈谈在 WS-RM 可靠消息传输模型中没有提及的一个主题——流控制。流控制是 WCF 基于 WS-RM 规范的一个扩展，它的实现从一个方面反映所有 WS-* 规范的可扩展性。

在前面已经说过，消息缓冲是 WS-RM 实现可靠消息传输的主要机制。消息缓冲机制反映在 WCF 的可靠会话上，就是客户端和服务端的 RS 信道各自拥有消息缓冲区，它们的大小即容纳消息的数量可以独立地进行配置。对于消息发送端来说，如果消息缓冲区已满，RS 信道就不能处理从上层信道传来的消息，直到接收到某个已发消息的确认后对应的消息从缓冲区中移除。对于消息接收端来讲，如果缓冲区已满，RS 信道则不能处理来自下层信道消息，直到缓存的消息被成功交付后从缓冲区移除。

由于客户端和服务端 RS 信道维持的消息缓冲区是相互独立的，因此如果发送端的消息缓冲区远远大于接收端消息缓冲区的大小，就会导致消息在接收端出现阻塞的现象。如果我们将消息的传输比喻成一条河流，上面的场景就意味着河流的源头水量很大，但是下流河道很窄，不能充分容纳从源头流流入的水量，这就必然导致河水泛滥。为了解决这个问题，WCF 的可靠会话采用了流控制的机制。

实际上流控制机制从实现上非常简单，我们可以将其称为“接收端接收容量通知机制”。表现在消息交换上面就是：消息的接收端在对接收消息进行确认的时候，会顺带将本地消息缓冲区还能容纳的消息数量一并放在确认消息中。消息发送端在接收到消息确认后，提取该值并确定是否继续发送消息。

在前面给出的 SequenceAcknowledgement 消息中，细心的读者可能留意到了在 <SequenceAcknowledgement> 报头中具有一个在 WS-RM 规范中没有提及的元素 <Buffer

Remaining>。该元素携带的数字就是接收端消息缓冲区中还能继续接纳的消息数量。

```
<r:SequenceAcknowledgement>
  <r:Identifier>urn:uuid:04127209-14ce-4648-90a5-a8eca6006fd2</r:Identifier>
  <r:AcknowledgementRange Lower="1" Upper="2"/>
  <r:Final/>
  <netrm:BufferRemaining:netrm="http://schemas.microsoft.com/ws/2006/05/rm">8
</netrm:BufferRemaining>
</r:SequenceAcknowledgement>
```

5.3.2 从传输协议的局限性和消息交换模式看可靠会话的实现

前面的章节中我们站在信道层的角度剖析了 WCF 为了实现可靠会话在信道层进行的一系列消息交换，或者说客户端和服务端的 RS 信道为了实现可靠消息传输所进行的一轮又一轮的握手。这一切都是基于这样一个假设：两个 RS 信道均可以在适当的时机向对方发送消息，或者两个 RS 信道之间是一个双工的通道。

如果站在传输层看待这个问题，该假设对于 TCP 传输是成立的，但是对于 HTTP 来说就有点问题了。HTTP 本身就是一个基于请求-回复消息交换模式的应用层网络协议，并不能对双工通信提供支持。

而 WCF 通过 WSDualHttpBinding 实现的双工通信机制和 NetTcpBinding 支持的双工通信具有本质的区别。NetTcpBinding 创建的传输通道就是一个双工的 TCP 连接，而 WSDualHttpBinding 创建的所谓的双工通道实际上是两个方向相反的 HTTP 连接。接下来主要讨论当采用基于 HTTP 的 WSHttpBinding（或者是 WS2007HttpBinding）和 WSDualHttpBinding 时，实现可靠会话所进行的通信方式。

1. WSHttpBinding V.S. WSDualHttpBinding

如果采用 WSHttpBinding，最终创建的是一条从客户端到服务端的 HTTP 通道。在这种情况下，客户端 RS 信道和服务 RS 信道之间的多轮握手（CreateSequence/CreateSequenceResponse、Sequence/SequenceAcknowledgement、CloseSequence/CloseSequence 和 TerminateSequence/TerminateSequenceResponse）均是采用这样的消息交换方式：客户端将相应的消息通过 HTTP 请求的形式发送到服务端，相应的回复或者确认通过 HTTP 回复返回。图 5-11 揭示了上述的几次握手在传输层上的实现，其中实线部分代表 HTTP 请求，虚线部分代表 HTTP 回复。

从图 5-11 中可以清晰地看到，CreateSequence/CreateSequenceResponse、CloseSequence/CloseSequence 和 TerminateSequence/TerminateSequenceResponse 完全是按照 HTTP 请求-HTTP 回复的形式实现的。在进行服务调用的时候，即使采用的是单向消息交换模式，发送的消息请求依然会接收到一个包含 SOAP 消息的 HTTP 回复。服务端将确认消息置于每一个 HTTP 回复之中。

之所以采用如上的方式，根本目的在于，WSHttpBinding 创建的传输层通道是从客户端

到服务端的一条 HTTP 连接。HTTP 连接是一条单工通道，客户端和服务端总是扮演着请求者和回复者的角色。由于服务端不能主动联系客户端，因此无论是对 RM 序列创建、关闭和终止的回复，还是消息确认，只能放在 HTTP 回复中。

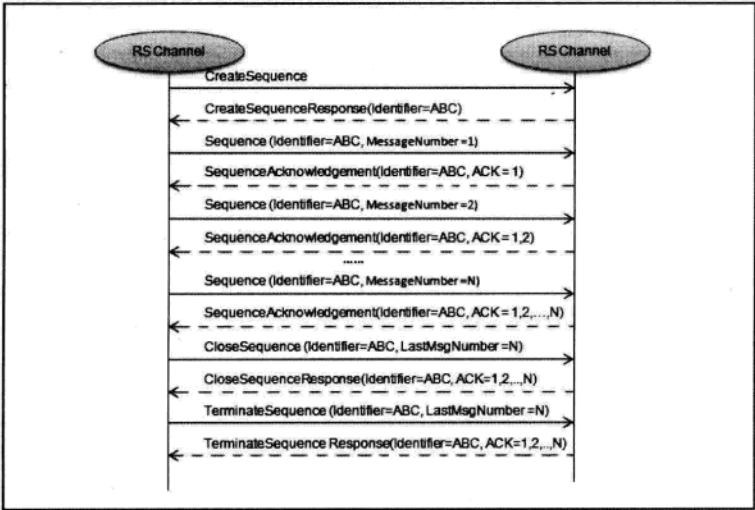


图 5-11 可靠会话基于通过 WSHttpBinding 创建的单通道的消息交换

如果我们采用 WSDualHttpBinding 作为终结点绑定，情况就大不一样了。由于 WSDualHttpBinding 会创建两条 HTTP 连接构成一个所谓的双工通道，服务端可以随时联系到客户端，不需要将相应的回馈通过 HTTP 回复捎回去。

借助于 WSDualHttpBinding 创建的双工通道，可靠会话的上述握手采用如下的消息交换方式：客户端通过 HTTP 请求将 RM 序列创建、终止请求及携带<Sequence>报头的应用消息发送给服务端，并得到一个状态为 202 的空 HTTP 回复。而真正的回复和消息确认都是通过另一个 HTTP 连接的 HTTP 请求返回给客户端的，这些 HTTP 请求会得到一个状态为 202 的空 HTTP 回复。

图 5-12 是可靠会话消息交换在传输层的反映。可能你会觉得这和前面介绍的 WS-RM 消息交换模式不一致。不但没有了 CloseSequence/CloseSequence 握手，对于 TerminateSequence 请求也没有相应的 TerminateSequenceResponse 回复。这是因为 WSDualHttpBinding 支持的 WS-RM 版本是 1.0，而不是我们前面介绍的 1.1。

除了上述的两点，还有一个不一样的地方。客户端在发送 RM 序列终止请求之前会发送一个携带<Sequence>报头的空消息，而包含在该空消息中的<Sequence>报头，除了包含消息序号之外，还具有一个额外的<LastMessage>元素表明这是 RM 序列终止前的最后一个消息。关于 WS-RM 1.0，限于篇幅，在本书中不可能再进行深入的介绍，有兴趣的读者可以参阅 OASIS 官方文档。

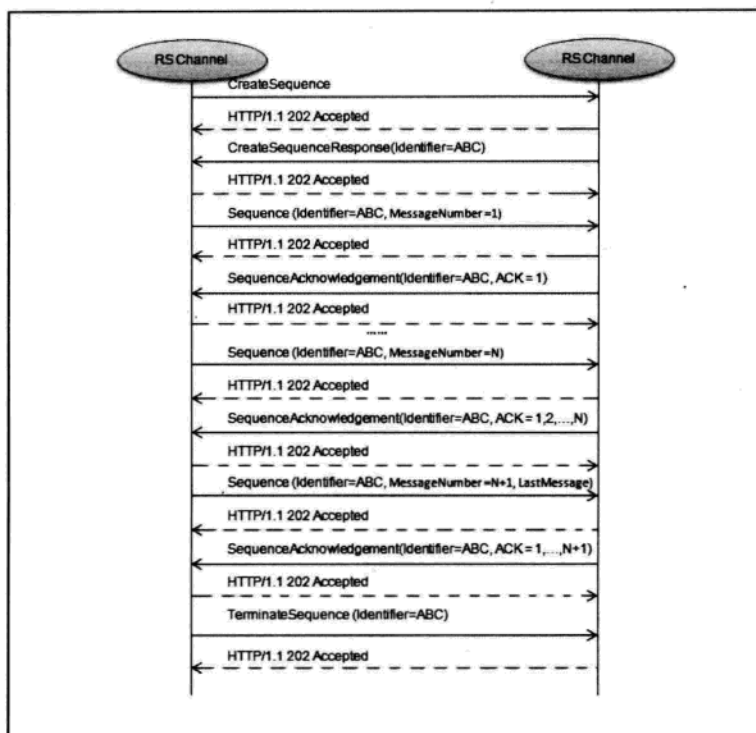


图 5-12 可靠会话基于通过 WSDualHttpBinding 创建的单通道的消息交换

也可以从另外一种视角来看 WSHttpBinding 和 WSDualHttpBinding 对可靠会话的不同实现方式。对于 WSHttpBinding 创建的单向信道来说，客户端对于服务端是一个不可寻址（Non-Addressable）的终结点。客户端不能主动向服务端发起请求，只能在客户端对自己发起请求时，被动地将相应的信息通过 HTTP 回复的形式返回到客户端。

但是对于 WSDualHttpBinding 创建的双工信道，情况就不一样了。双工通道使客户端和服务端成为对等终结点，无论是服务端还是客户端，对于对方来说都是可寻址的。服务端可以在任何时候向客户端发起请求，将相应的信息通过 HTTP 请求的方式发送给客户端。

双工通道成就了可靠会话的“批量回复”机制。为了尽可能地降低网络流量，接收端 RS 信道接收到消息之后，并不会立即为该消息进行单独确认，而是会等待一定的时间（通过 ReliableSessionBindingElement 的 AcknowledgementInterval 属性设置），对之前接收到的消息进行批量确认。由于接收端 RS 信道接收到消息和发送确认有一定的延迟，因此也称这种机制为“延迟确认”。

2. 单向模式（One-Way）V.S.请求-回复（Request-Reply）和双工（Duplex）模式

决定实现 WCF 可靠会话真正采用的消息交换还具有另外一个因素，那就是操作采用的消息交换模式。单向模式和请求-回复及双工模式下，可靠会话采用的消息交换方式具有很

大的不同。

如果终结点服务契约中的所有操作均是单向的 (通过 `OperationContractAttribute` 特性的 `IsOneway` 属性设置), 对于可靠会话来说仅存在一个从客户端到服务端的 RM 序列。反映在序列的创建上就意味着在客户端 RS 生成的 `CreateSequence` 消息中并不存在 `<Offer>` 节点。

从应用层次讲, 单向操作意味着客户端向服务端发送消息而不会接收到任何回复。由于服务端不会有任何的应用消息从服务端返回到客户端, 因此服务端的 RS 信道只能创建一个空的包含 `<SequenceAcknowledgement>` 报头的消息对接收的消息进行确认。

如果终结点服务契约中的所有操作中具有一个以上的非单向操作, WCF 可靠会话不仅需要保障消息从客户端到服务端的可靠性, 也需要对服务端到客户端的消息传输提供保障, 所以 WCF 可靠会话需要建立两个方向相反的 RM 序列。可靠会话采用序列提供机制创建了双向的 RM 序列。在客户端 RS 信道生成 `CreateSequence` 请求之前先在本地创建一个 RM 序列, 然后将该序列封装到 `CreateSequence` 消息的 `<Offer>` 元素中“提供”给服务端。服务端 RS 信道接收到 `CreateSequence` 消息之后, 除了创建客户端请求的 RM 序列之外, 还会接受 (或者拒绝) 提供的序列。

不同于单向模式下采用单独的独立消息进行消息确认, 在请求-回复模式下, 为了尽量降低网络流量, 可靠消息采用“背负 (piggy-back)”机制实现消息确认。客户端 RS 信道将 `<SequenceAcknowledgement>` 报头放到请求消息中, 实现对接收到的回复消息的确认。服务端 RS 信道则将 `<SequenceAcknowledgement>` 报头放到回复消息中, 实现对已经接收到的请求消息的确认。

而双工 (Duplex) 是由两个简单消息交换模式 (单向或者请求-回复模式) 组合而成的, 具体消息交换方式应该可以从上面这两种基本消息交换模式推导出来, 在这里就不再赘述了。

5.3.3 可靠会话最佳实践

经过前面的介绍, 无论是对于 WCF 可靠会话的编程模型还是实现原理, 都有了一个大致的了解。在此基础上, 介绍关于可靠会话在具体应用开发中的一些最佳实践。以下介绍的这些内容全部来源于 MSDN。

1. 设置 `MaxTransferWindowSize`

WCF 中的可靠会话使用传输窗口保存客户端和服务上的消息。可配置属性 `MaxTransferWindowSize` 指示传输窗口可以保存多少条消息。

在发送方, 指示在等待确认消息时传输窗口可以保存多少条消息, 在接收方, 则指示为服务缓冲多少条消息。

选择合适的大小可影响使用网络的效率及运行服务的最佳容量。下面将详细介绍选择此属性的值时要考虑的事宜及值的影响。

默认传输窗口大小是 8 条消息。

2. 有效使用网络

此处的“网络”一词对应于在客户端（发送方）和服务（接收方）之间用做通信基础的任何事物。包括传输连接及中间的任何中介或者网桥，包括 SOAP 路由器或 HTTP 代理/防火墙。

有效使用网络可确保充分利用网络容量。每秒通过网络传输的数据量（称为“数据速率”）及从发送方到接收方传输数据所用的时间（称为“延迟”）都会影响利用网络的有效性。

在发送方，属性 `MaxTransferWindowSize` 指示在等待确认消息时，其传输窗口可以保存多少条消息。因此，如果网络延迟时间很长，为确保及时响应发送者和对网络的有效利用，应增加传输窗口大小。

例如，即使发送方满足数据速率，如果发送方和接收方之间存在多个中介或者中介或网络存在损失，则延迟时间会很长。因此，在发送方接收要在网络上发送的新消息之前，必须在其传输窗口中等待消息的确认信息。具有高延迟的缓冲区越小，网络利用率就越低。另一方面，传输窗口大小过高可能会影响服务，原因是服务可能需要满足客户端的高发送速率。

3. 满负荷运行服务

为最大程度地有效使用网络，理想情况是服务也按最佳容量运行。接收方的传输窗口大小属性指示接收方可以缓冲多少条消息。此消息缓冲不仅帮助网络进行流控制，还可让服务满负荷运行。例如，如果缓冲区是 1，而且消息到达的速度超过了服务可以处理的速度，则网络可能会丢弃一些消息，并可能浪费或闲置网络容量。

使用缓冲区可提高服务的可用性，因为服务可以在处理以前接收到的消息的同时并发接收和缓冲消息。

建议在发送方和接收方使用相同的 `MaxTransferWindowSize`。

4. 启用流控制

流控制是确保发送方和接收方保持步调一致的机制，也就是说，使用和处理消息的速度与产生消息的速度一样快。客户端和服务端的传输窗口大小可确保发送方和接收方在一个合理的同步窗口中。

当在 WCF 客户端和 WCF 服务之间使用可靠会话时，强烈建议将 `FlowControlEnabled` 属性设置 `True`。

5. 设置 MaxPendingChannels

当编写一个允许从不同的客户端启用可靠会话通信的服务时,可能会有许多客户端同时建立与该服务的可靠会话。在这些情况下,服务的响应取决于 MaxPendingChannels 属性。

当发送方创建到接收方的可靠会话通道时,发送方和接收方之间的握手将建立可靠会话。建立可靠会话之后,该通道会放入到挂起的通道队列中以供服务端接收。此 MaxPendingChannels 属性指示有多少个通道可以处于此状态。

服务端有可能会处于一种无法接收更多通道的状态。如果队列已满,则会拒绝建立可靠会话的尝试,客户端必须重试。

队列中挂起的通道也可能在队列中保持很长时间。此外,可能会出现可靠会话的非活动超时,从而导致通道转换到错误状态。

因此,在编写同时服务于多个客户端的服务时,应设置一个适合需要的值。为 MaxPendingChannels 属性设置过高的值会影响工作集。

MaxPendingChannels 的默认值为 4。

6. 可靠会话和宿主

在为使用可靠会话服务提供 Web 宿主时,应该记住下面的重要注意事项。

- 可靠会话是有状态的,而状态在 AppDomain 中进行维护。这意味着属于可靠会话的一部分的所有消息必须在同一个 AppDomain 中进行处理。其大小超过 1 的网络场和网络园无法保证满足此约束。
- 使用双工 HTTP 通道(例如使用 WsDualHttpBinding)的可靠会话会要求多于默认的每个客户端 2 个 HTTP 连接的连接数。这意味着双工可靠会话会在每个方向上要求 2 个连接,因为并发应用程序和协议消息可能会在任意给定时间在每个方向上进行传输。这意味着,在某些特定的条件下,根据消息交换服务模式的不同,使用双工 HTTP 和可靠会话的 Web 承载服务可能会出现死锁。若要增加每个客户端允许的 HTTP 连接数,请将下列代码添加到相关配置文件中(例如,相关服务的 web.config)。其中的“XX”是需要的连接数。

```
<configuration>
  <system.net>
    <connectionManagement>
      <add name = "*" maxconnection = "XX" />
    </connectionManagement>
  </system.net>
</configuration>
```

第 6 章 队列服务

(Queued Service)

WCF 是 Windows 平台下传统分布式技术之集大成者,其中较有代表性的一项分布式技术就是消息队列 (Message Queuing),一般简称为 MSMQ。我们将采用 MSMQ 作为通信方式的服务称为队列服务 (Queued Services)。队列服务不仅能够很好地支持离线通信 (Offline Communication),很好地解决峰谷平衡,还通过 MSMQ 本身的机制提供了另一种形式的可靠消息传输 (RM, Reliable Messaging)。

虽然说队列服务就是终结点采用 `NetMsmqBinding` 作为绑定的服务，但是由于其通信机制采用特殊的消息交换方式，决定了队列服务在服务契约的定义、事务支持、会话和实例管理，以及异常处理方面都和传统的服务有所不同，本章将从这些方面全面深入地剖析队列服务。

6.1 MSMQ 简介

考虑到很多人可能对 MSMQ 没有太多的了解，所以在正式介绍 WCF 的队列服务之前，先对 MSMQ 做简单的介绍。如果你是这方面的专家，完全可以忽略这一节，直接进入下一节。下面先来讨论 MSMQ 具体能给我们带来什么。

6.1.1 MSMQ 能解决什么问题

从参与者之间的关联程度，可以将分布式应用中的消息通信分成两类。一类需要参与者在正式进行消息交换之前必须建立连接，称为面向连接的通信 (Connected Communication)。另一类则不需要预先建立连接，消息的发送者和接收者相对独立。发送端的任务只是向目标地址以单向 (One-Way) 的方式发送消息，而接收方则只需要绑定到某个地址进行消息的监听和接收就可以了。这是一种非连接的通信方式 (Disconnected Communication)。如果我们用两种通信协议与这两种通信方式做类比，那么它们分别相当于 TCP 和 UDP 协议。

MSMQ 提供更高程度的非连接通信支持，因为它支持离线通信 (Offline Communication) 方式。消息的发送者可以在离线方式下进行工作。离线通信方式让进行消息交换的参与者变成完全独立的两个应用，它们之间唯一的纽带就是某个消息队列 (Message Queue)。

基于 MSMQ 的离线通信就相当于我们传统的书信邮递。除了必要的收件人地址、邮编和姓名之外，寄信人不需要知道对方更多的信息。他只需要将这些基本信息写在信封上，贴上邮票放入邮筒即可。而我们的邮政系统会将此信投递到对方的邮箱里。作为收信人也无须考虑太多，只需要在规定的期限内去查看自家的邮箱是否装有邮件即可。

消息队列表现出来的作用就相当于寄信人的邮筒和收信人的邮箱，而消息队列服务 (Message Queuing Service) 就相当于隐藏在背后实现信件投递的系统。消息的发送者在发送的消息中写上目标队列的地址，然后通过 MSMQ 进行发送。这个发送的消息可以暂存在本地的消息队列中，就好比是寄信人家附近的邮筒。如果目标消息队列可达，则将消息发送过去，否则会长时间地在本地保存该消息，以等待能够成功连接到目标队列的那一刻重发。而绑定到目标队列的接收者会通过 MSMQ 去监听和接收该消息。

我们可以通过 MSMQ 创建一个面向业务员的用于处理订单的应用。这个应用的客户端是安装在业务员笔记本上的桌面程序，而业务员所处的环境并不能保证一直具有网络连接。因为 MSMQ 支持离线通信方式，所以业务员在无网络连接的情况下依然可以“正常地”提交订单。用于封装订单的消息会先暂存在本地的消息队列中，等到他晚上回到家中并连接到 Internet，订单会自动地发送到服务端。

提供离线通信除了让参与消息交换的发送者和接收者变成两个完全独立的应用之外，还能解决“峰谷平衡”的问题。由于业务往往都具有周期性，业务处理量并不是每一个时刻都是一样的。很有可能在某个时刻出现一个超出处理能力的峰值，并在另一个时刻出现让大多数服务器都闲置的谷值（该值甚至可能为零）。如果采用基于连接的通信方式，必须采购足够的服务器或者升级硬件以应对峰值出现时的高负载。但是这会带来资源的极大浪费，对这些添置的资源来说，只是“养兵千日，用在一时”。

如果借助于 MSMQ 提供的通信方式，无论出现多高的负载，只要抵达的消息累积量不超过设定的限额，MSMQ 就完全可以按照自己的节奏来进行处理。将高负载时累积的请求延后到低负载的时候进行处理，很好地解决了“峰谷平衡”的问题。

在本书的第 5 章“可靠会话 (Reliable Sessions)”中，我们谈到 WCF 可以通过可靠会话的方式实现消息的可靠、有序的递交。而 MSMQ 本身通过确认和重传机制，也可以实现消息的可靠传输。

前面我们一直在谈离线通信的好处，但是具体采用怎样的通信方式取决于业务的具体需求。基本上只有对实时性要求不高的业务处理才会选择这种通信方式。此外 MSMQ 采用的消息发送永远是单向的，所以不会立即接收到回复，也不会自动捕捉到接收应用抛出的异常。

6.1.2 MSMQ 的安装

MSMQ 不是一个完全独立的通信框架，而是作为 Windows 操作系统的一部分发布出来的。所以需要按照操作系统组件或者功能 (Feature) 的方式来安装 MSMQ，并且只能安装与操作系统一致的 MSMQ 版本。下面列出了不同的 Windows 操作系统所支持的 MSMQ 的版本。

- Windows 7/Windows Server 2008 R2: MSMQ 5.0;
- Windows Vista/Windows Server 2008: MSMQ 4.0;
- Windows XP/Windows Server 2003: MSMQ 3.0;
- Windows 2000: MSMQ 2.0。

假设我们所采用的是 Windows 7 操作系统，可以通过选择“控制面板”|“程序和功能 (Programs and Features)”|“打开或关闭 Windows 功能 (Turn windows features on or off)”

来开启或者关闭 MSMQ 相关的功能。如图 6-1 所示, 整个 MSMQ 的功能集具有两个主要的组成部分: MSMQ 服务器核心 (MSMQ Server Core) 和 MSMQ DCOM 代理 (MSMQ DCOM Proxy)。



图 6-1 MSMQ 功能安装

下面列出了具体的功能元素的作用。

- **MSMQ Active Directory 服务集成 (MSMQ Active Directory Domain Services Integration):** 采用 AD 域服务集成, 消息队列名称可以注册到 AD 域中, 使我们在无须指定机器名称的情况下就可以定位访问某个消息队列。
- **MSMQ HTTP 支持 (MSMQ HTTP Support):** 该功能使我们可以通过 HTTP 协议发送和接收消息。
- **MSMQ 触发器 (MSMQ Triggers):** 触发器使我们可以在某个消息抵达目标队列的时候自动激活某个应用来处理它。
- **多播支持 (Multicasting Support):** 该功能使我们可以通过多播的形式将某个消息发送给一组服务器。
- **MSMQ DCOM 代理 (MSMQ DCOM Proxy):** 采用 DCOM 代理, 系统可以借助于 DCOM 应用程序接口 (API) 连接远程服务器。

MSMQ 的上述功能最终承载在相应的 Windows 服务中。当启动 MSMQ 服务的时候, 相当于启动了一个名称为 Message Queuing 的 Windows 服务。而 MSMQ 触发器对应的 Windows 服务名称则为 Message Queuing Triggers。图 6-2 在服务对话框中的 Windows 服务列表中标出了这两个 MSMQ 相关的 Windows 服务。

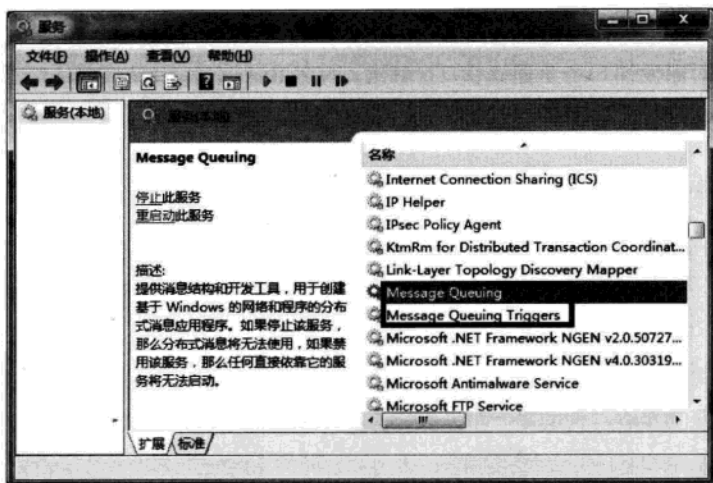


图 6-2 MSMQ 相关的两个 Windows 服务

6.1.3 消息队列

作为消息存储容器的消息队列，存在于系统目录%Windir%\System32\msmq\storage 下。按照队列自身属性，以及在消息通信场景中扮演的角色不同，我们将消息队列分成相应的类型。

1. 普通队列

所谓普通队列，就是为具体的应用创建的，用于存储基于业务消息的队列。普通队列分为公有队列和私有队列。公有队列的名称被注册到 AD 域中，所以我们无须指定队列所在的机器名称就可以存取队列。当我们将某个公有队列从一个机器转移到另一台机器时，访问该队列的应用可以保持不变。公有队列还可以提供基于域账号的 Windows 认证机制，所以对于正式发布的应用来说，通常采用公有队列。

因为公有队列需要注册到 AD 域中，所以它只能被使用在域（Domain）模式下。在工作组（Work Group）模式下，只能使用私有队列。访问私有队列需要指定包含队列所在机器名称的路径。

2. 管理队列（Administration Queue）

MSMQ 也是通过消息确认和重传的机制实现对消息的可靠传输的，确认消息被存储于消息发送端的管理队列中。被发送的消息在发送之前被设置一个管理队列，当消息成功抵达目标队列或者被成功接收后，接收方会从消息中获取管理队列的路径，并创建一个确认消息（ACK, Acknowledgment）并发送到该管理队列中。

除了这种正面的、表示消息成功抵达和接收的确认之外,在消息接收失败的情况下,也可以向发送方的管理对象发送一个负面确认 (NACK, Negative Acknowledgment)。从队列属性来讲,管理队列和普通队列没有本质的区别,需要自行创建。

3. 回复队列 (Response Queue)

MSMQ 完全采用单向的消息交换模式,消息被发送之后是没有回复消息返回给发送端的。但是在一些情况下,除了简单的确认 (ACK 或者 NACK) 之外,要求接收端应用真正地回复以相应的内容,这就涉及了回复队列的使用。

和消息的确认机制类似,需要被回复的消息在发送之前会指定一个回复队列。消息在被接收端应用成功接收并处理后,会创建一个回复消息发送到回复队列中。

4. 日志队列 (Journal Queue)

当消息被成功发送或者接收后,MSMQ 可以将消息的拷贝作为发送或者接收日志存储起来。这个消息拷贝称为日志消息,存放消息的队列称为日志队列。按照日志是基于发送还是接收操作,可以将日志分为源日志 (Source Journaling) 和目标日志 (Target Journaling) 两种类型。

源日志消息被存储于发送端的日志队列中,是否需要进行日志记录由队列本身的属性决定。目标日志消息被存储于接收端的日志队列中,是否需要进行日志记录取决于消息本身的属性。

当我们创建一个公有或者私有队列的时候,基于该队列的日志队列被自动创建。除了基于某个队列的日志队列,还具有一个系统级别的日志队列。

5. 事务性队列 (Transactional Queue)

MSMQ 和 SQL Server 一样,属于事务管理器 (RM, Resource Manager),可以登记到一个分布式事务中,但这仅仅局限于事务性队列。事务性队列可以确保 MSMQ 以事务的方式处理消息的发送、传输和接收。

由于 MSMQ 消息交换模式的特殊性,消息的发送、传输和接收之间可以跨越很长一段时间。而事务本身并不具有长时间的控制力,所以一个事务不能跨越从消息被发送到被接收这一段完整的过程。一个事务只能分别控制消息的发送 (发送到本地队列)、网络传输 (源队列到目标队列) 和接收 (从目标队列中读取)。

可靠传输还具有一个额外的传输保障,那就是“有序传输”,即取保消息以被发送的顺序向接收端应用递交。消息的有序递交机制也仅限于事务性队列。

6. 死信队列 (Dead-Letter Queue)

现实中的死信 (Dead-Letter) 代表无法投递的信件。在 MSMQ 中也具有相似的概念,

即代表在限定的时间范围内（在消息发送时指定的超时时限）无法递交的消息。导致递交失败的因素有很多，比如网络不通、认证失败、目标队列已满等，这些死信消息被存放在死信队列中。MSMQ 具有两个系统级别的死信队列，它们分别用于非事务性队列和事务性队列。

7. 报表队列（Report Queue）

报表队列是公有队列，用于存储基于路由追踪（Route Tracking）的报表消息。如果发送端主机开启了路由追踪，不论消息离开还是抵达某个主机，携带着路由信息的报表消息将会自动发送到报表队列中。

报表队列是基于主机的，存在于 AD 域中的某台主机具体采用哪个公有队列作为本机的报表队列是由消息队列管理员指定的。

8. 子队列（SubQueue）

队列是消息的容器，针对消息队列的一项常用的操作是将消息从一个队列转移到另一个队列。但是前面介绍的所有队列都对应于某个物理文件，要实现跨物理队列之间的消息转移，在性能上是比较差的。在这种情况下子队列是一种不错的选择。

子队列并不对应一个单独的物理文件，而是依附于某个物理队列的逻辑队列，所以消息在物理队列和它的子队列之间进行转移具有非常好的性能优势。子队列无须手工创建，它在消息抵达的时候会自动创建，而当消息被清空之后则自动删除。子队列具有如下两个典型的应用。

- 有序递交：如果物理队列接收的消息的序列和发送的次序不一致，可以将子队列作为缓存区。接收到的消息先存储在该缓存区域中并进行排序，等到具有最小序列号的消息抵达之后才进行有序的递交。
- 毒性队列：对于在处理过程中频繁出错的消息，可以将其暂存于相应的子队列中以使其他的消息能够得到及时处理。我们将这样的消息称为毒性消息（Poison Message），将这样的子队列称为毒性队列（Poison Queue）。

除了这两个典型的应用，WCF 还使用子队列实现了消息接收的重试机制，具体的实现机制会在本章的后续部分进行详细的介绍。子队列在 Windows Vista 开始被引入 MSMQ，所以 Windows XP 和 Windows 2003 均不支持子队列。

6.1.4 MSMQ 编程

接下来从编程的角度继续认识 MSMQ。基于 MSMQ 的应用编程接口（API）基本上定义在 System.Messaging.dll 程序集中，程序集的名称对应着命名空间的名称。所以对于下面出现的类型，如果没有具体指明其命名空间，意味着它定义在命名空间 System.Messaging 下。

1. 消息队列的创建与删除

首先来看看如何通过编程的方式创建一个新的消息队列。在 MSMQ 的应用编程接口 (API) 中, 一个消息队列通过类型 `MessageQueue` 来表示。`MessageQueue` 定义了两个重载的 `Create` 的静态方法供我们创建一个新的消息队列。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public static MessageQueue Create(string path);
    public static MessageQueue Create(string path, bool transactional);
}
```

如上面的代码片段所示, 两个 `Create` 方法均具有一个字符串类型的参数 `path`, 代表被创建消息队列的路径。而其中一个 `Create` 方法具有一个布尔类型的 `transactional`, 代表是否要创建一个事务性队列。默认情况下 (没有显式指定 `transactional` 参数) 创建的队列是非事务性的。两个方法返回代表被创建的消息队列的 `MessageQueue` 对象。

消息队列的路径由队列所在的机器名称、队列的类型 (公有或者私有) 和队列的名称三个部分组成。下面的列表分别列出了公有队列和私有队列的路径格式。而对于本地的消息队列, 机器名称可以表示为 `localhost` 或者 `."`。需要注意的是, 不能用 IP 地址代替机器名称。在工作组模式下, 只能创建私有队列, 并且只能创建本地私有队列。

- 公有队列: `{MachineName}\{QueueName}`
- 私有队列: `{MachineName}\{Private$}\{QueueName}`

一般来说, 我们在通过指定队列路径进行消息队列的创建之前, 会检验基于指定的路径是否已经存在一个消息队列。消息队列存在性的检验可以通过调用 `MessageQueue` 的另一个方法静态 `Exists` 来实现。如下面的代码片段所示, 该方法将代表队列路径的字符串作为输入, 如果对应的消息队列存在, 方法返回 `True`, 否则返回 `False`。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public static bool Exists(string path);
}
```

对于一个存在的消息队列, 可以调用 `MessageQueue` 的静态方法 `Delete` 来将其删除。从下面的代码片段来看, `Delete` 方法接受一个代表删除队列路径的字符串。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public static void Delete(string path);
}
```

实际上, 当我们删除一个消息队列的时候, 不仅可以指定删除队列的路径, 还可以指定删除的对象的标签 (`Label`) 和格式名称 (`FormatName`)。如下面的代码所示, 一个

MessageQueue 具有一个可读写的代表队列标签的 Label 属性（并非消息队列的唯一标识）和一个只读的 FormatName 属性。可以在 Delete 方法中指定队列的标签或者格式名称删除相应的消息队列。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public string Label { get; set; }
    public string FormatName { get; }
}
```

不论是队列路径，还是队列的标签和格式名称，都是一个字符串，MSMQ 通过预定义的前缀来区分具体指定的字符串是标签（格式为“Label: {Label}”）还是格式名称（格式为“FormatName: {FormatName}”）。在工作组模式下基于标签的队列删除是不被支持的。关于消息队列的格式名称，将在后续部分进行详细介绍。

2. 消息队列的查询

如果需要对创建的消息队列进行统一管理，就需要根据相应的查询条件去搜索相应的消息队列。MessageQueue 通过静态方法的方式提供了一系列的方法供我们进行自定义查询条件的队列搜索。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public static MessageQueue[] GetPrivateQueuesByMachine(string machineName);
    public static MessageQueue[] GetPublicQueuesByMachine(string machineName);
    public static MessageQueue[] GetPublicQueuesByCategory(Guid category);
    public static MessageQueue[] GetPublicQueuesByLabel(string label);
    public static MessageQueue[] GetPublicQueues();
    public static MessageQueue[] GetPublicQueues(MessageQueueCriteria criteria);
}
```

从上面的代码片段可以看出，可以通过调用相应的 GetXxx 方法得到基于相应查询条件的消息队列数组。比如通过指定机器名称得到某台机器上创建的所有私有队列。而针对公有队列，则提供多种查询方式，包括基于机器名称、类别和标签。如果调用没有参数的 GetPublicQueues 方法，会得到当前网络中的所有消息队列。

GetPublicQueuesByCategory 方法中类型为 GUID 的输入参数 category，是一个代表队列类别的 GUID，对应着 MessageQueue 的可读写属性 Category。可以通过指定消息队列的 Category 属性实现对其进行分类或者分组的目的。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public Guid Category { get; set; }
}
```

对于公共队列，除了调用相应的 GetPublicQueuesByXxx 方法进行条件查询之外，还可

以调用 `GetPublicQueues` 方法通过指定 `MessageQueueCriteria` 对象实现多条件组合查询。`MessageQueueCriteria` 具有如下的属性定义，可以通过它来组合基于多个队列属性（机器名称、类别、标签、创建时间和最后修改时间）的复杂查询条件。需要提醒的是，所有基于公有队列的查询在工作组模式下都是不被支持的。

```
public class MessageQueueCriteria
{
    //其他成员
    public string MachineName { get; set; }
    public Guid Category { get; set; }
    public string Label { get; set; }

    public DateTime CreatedAfter { get; set; }
    public DateTime CreatedBefore { get; set; }
    public DateTime ModifiedAfter { get; set; }
    public DateTime ModifiedBefore { get; set; }
}
```

除了上面这些直接以 `MessageQueue` 对象数组的方式查询的方法之外，`MessageQueue` 还提供了如下两个重载的返回值类型为 `MessageQueueEnumerator` 的 `GetMessageQueueEnumerator` 方法。前者用于获取当前网络下的所有公有队列，所以该方法在工作组模式下是不被支持的。后者则通过传入的 `MessageQueueCriteria` 指定的查询条件进行队列的搜索。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public static MessageQueueEnumerator GetMessageQueueEnumerator();
    public static MessageQueueEnumerator
        GetMessageQueueEnumerator(MessageQueueCriteria criteria);
}
```

`MessageQueueEnumerator` 是一个实现了 `IEnumerator` 接口的迭代器，可以使用它对返回的 `MessageQueue` 集合进行迭代。如下面的代码片段所示，`MessageQueueEnumerator` 显式实现了只读的 `Current` 属性，并且定义了属于自己的类型为 `MessageQueue` 的 `Current` 属性。

```
public class MessageQueueEnumerator : MarshalByRefObject, IEnumerator,
IDisposable
{
    //其他成员
    public MessageQueue Current { get; }
    object IEnumerator.Current { get; }
}
```

3. 创建一个 `MessageQueue` 对象

一个消息队列通过一个 `MessageQueue` 对象来表示。但是这里所说的创建一个 `MessageQueue` 对象并不是建立一个新的消息队列，而是调用相应的构造函数创建一个针对现有消息队列的 `MessageQueue` 对象。现在看看 `MessageQueue` 定义了哪些构造函数重载。

```

public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public MessageQueue(string path);
    public MessageQueue(string path, bool sharedModeDenyReceive);
    public MessageQueue(string path, QueueAccessMode accessMode);
    public MessageQueue(string path, bool sharedModeDenyReceive, bool
        enableCache);
    public MessageQueue(string path, bool sharedModeDenyReceive, bool enableCache,
        QueueAccessMode accessMode);
}

```

一般通过指定队列的路径来创建对应的 `MessageQueue` 对象，这个路径通过构造函数的 `path` 参数指定。对于普通的消息队列来说，它们的路径由机器名称、队列类型（公有或者私有）和队列名称这三部分组成，而对于公有队列，只保留机器名称和队列名称两部分即可。对于一些系统级队列和伴随着普通对象被创建出来的日志队列，它们的路径又如何表示呢？下面列出了这些特殊队列的路径格式。

- 日志队列（公有）：`{MachineName}\{QueueName}\Journal$`
- 日志队列（私有）：`{MachineName}\Private$\{QueueName}\Journal$`
- 系统日志队列：`{MachineName}\Journal$`
- 系统死信队列：`{MachineName}\DeadLetter$`
- 系统事务死信队列：`{MachineName}\XactDeadLetter$`

作为 `MessageQueue` 构造函数的 `path` 参数，除了接受队列的路径和格式名称作为输入外，还可以接收一个代表队列标签的字符串（以“Label:”作为前缀）。不过这仅限于域模式，在工作组模式下是不被支持的。下面的代表片段演示了分别针对路径、格式名称和标签的 `MessageQueue` 的创建方式。

```

string path      = @".\Private$\MyQueue";
string formatName = @"FormatName:DIRECT=OS:Jinnan-PC\Private$\MyQueue";
string lable     = "Label:MyLabel";

MessageQueue queue1 = new MessageQueue(path);
MessageQueue queue2 = new MessageQueue(formatName);
MessageQueue queue3 = new MessageQueue(lable);

```

在一系列的构造函数中，除了 `path` 参数还具有另外三个参数，即 `sharedModeDenyReceive`、`enableCache` 和 `queueAccessMode`。如将 `sharedModeDenyReceive` 参数指定为 `true`，意味着赋予了当前应用独占地读取目标队列的权限。`enableCache` 则代表是否创建连接缓存（`Connection Cache`），而 `queueAccessMode` 参数类型为 `QueueAccessMode`，代表创建的 `MessageQueue` 对象将会用于何种操作（发送、接收和查看），该枚举定义如下。

```

public enum QueueAccessMode
{
    Peek,
    PeekAndAdmin,
    Receive,

```

```

ReceiveAndAdmin,
Send,
SendAndReceive
}

```

除了指定路径和标签作为 `path` 参数的值之外, 该参数还接受队列的格式名称作为输入。如果我们的应用要提供离线通信的支持, 在创建 `MessageQueue` 的时候只能采用格式名称。

4. 消息队列的格式名称 (Format Name)

一个消息队列的格式名称包含执行相应操作 (比如消息的发送与接收, 队列属性的获取与设置等) 所需的必要信息。严格来说, 格式名称并不能算做某个消息队列的属性, 仅仅是队列的一个唯一标识。格式名称具有很多的类型, 不同的类型具有不同的语法, 接下来介绍三种典型的格式名称。

(1) 公有格式名称 (Public Format Name)

顾名思义, 公有格式名称专门针对于公有消息队列。该格式名称具有一个 “Public=” 字符前缀, 后跟一个能够唯一标识一个公有消息队列号 (Queue Number) 的 GUID。而基于公有消息队列的日志队列则在此基础上添加一个 “;JOURNAL” 后缀, 具体语法如下。

- 公有队列: `PUBLIC={QueueNumber}`
- 日志队列: `PUBLIC={QueueNumber};JOURNAL`

当采用一个基于公有格式名称的 `MessageQueue` 对象进行消息发送的时候, 队列管理器会通过 AD 域服务根据队列号查询该消息队列所在的机器名称, 采用的访问协议, 以及进行消息路由所需的其他信息。由于需要借助于 AD, 在工作组模式下该类型的格式名称是不被支持的。

(2) 私有格式名称 (Private Format Name)

和公有格式名称相对, 私有格式名称用于唯一地标识一个私有消息队列, 其前缀为 “PRIVATE=”。后面的两个私有格式名称分别针对私有队列和它的日志队列, 其中 `{MachineID}` 代表机器在 AD 中的注册号 (GUID)。至于 `{QueueNumber}`, 则与公有格式名称 (队列在 AD 中的注册号) 不同, 是在创建时直接生成并写入队列文件中, 能够在机器范围内作为唯一标识的队列号。

- 私有队列: `PRIVATE={MachineID}\{QueueNumber}`
- 日志队列: `PRIVATE={MachineID}\{QueueNumber};JOURNAL`

队列管理器在解析私有格式名称的时候, 需要借助于 AD 根据机器号得到具体的机器名称。正因为名称的解析需要 AD 的参与, 所以在工作组模式下私有格式名称是不被支持的。

(3) Direct 格式名称 (Direct Format Name)

不论是公有格式名称还是私有格式名称, 都需要 AD 辅助完成针对队列号/机器号的解

析，而 Direct 格式名称则完全不需要 AD 的参与。正如其名称所展示的一样，所谓 Direct 格式名称，就是说直接根据其名称本身就可以定位到对应的消息队列。换言之，Direct 格式名称包括辅助实现消息路由的各项信息，包括访问的协议、消息队列所在机器的名称/IP 地址/DNS 名称及队列名称等。Direct 格式名称以“DIRECT=”为前缀，下面 4 个 Direct 格式名称分别针对公有队列、公有日志队列、私有队列和私有日志队列。

- 公有队列：DIRECT={Address Specification}\{Queue Name}
- 公有日志队列：DIRECT={Address Specification}\{Queue Name}\{...};JOURNAL
- 私有队列：DIRECT={Address Specification}\Private\$\{Queue Name}
- 私有日志队列：DIRECT={Address Specification}\Private\$\{Queue Name}\{...};JOURNAL

其中{Address Specification}部分代表基于某种协议的地址，而具体的访问协议又包括如 TCP、HTTP (S)、IPX 和 OS 等。其中 TCP 和 HTTP (S) 协议支持 IP 地址和 DNS 域名服务名称。IPX 代表建立在协议 SPX (Sequenced Packet Exchange) 上的面向连接的 Internetwork Packet Exchange 协议，基于该协议的地址由网络号和主机号组成，格式为“{Network Number}:{Host Number}”。而 OS 采用 Windows 原生的机器名转换方式，可以直接使用机器名称作为其地址，对于 Windows NT 4.0 之后的版本，可以采用 DNS 域名服务名称。下面列出了公有队列和私有队列针对不同的协议的 Direct 格式名称。

- 公有队列 (TCP)：DIRECT=TCP:192.168.80.83\MyQueue
- 私有队列 (TCP)：DIRECT=TCP:192.168.80.83\Private\$\MyQueue
- 公有队列 (HTTP)：DIRECT=HTTP://Jinnan-PC/msmq/MyQueue
- 私有队列 (HTTP)：DIRECT=HTTP://Jinnan-PC/msmq/Private\$/MyQueue
- 公有队列 (HTTPS)：DIRECT=HTTPS://www.artech.com/msmq/MyQueue
- 私有队列 (HTTPS)：DIRECT=HTTPS://www.artech.com/msmq/Private\$/MyQueue
- 公有队列 (IPX)：DIRECT=IPX:00000012:00a0234f7500\MyQueue
- 私有队列 (IPX)：DIRECT=IPX:00000012:00a0234f7500\Private\$\MyQueue
- 公有队列 (OS)：DIRECT=OS:Jinnan-PC\MyQueue
- 私有队列 (OS)：DIRECT=OS:Jinnan-PC\MyQueue

对于几个重要的系统级别的日志队列、死信队列和事务性死信队列，如果采用 OS 协议，可以按照下面的格式来表示 (Jinnan-PC 为机器名称)。

- 系统日志队列：DIRECT=OS:Jinnan-PC\SYSTEM\$;JOURNAL
- 系统死信队列：DIRECT=OS:Jinnan-PC\SYSTEM\$;DEADLETTER
- 系统事务性死信队列：DIRECT=OS:Jinnan-PC\SYSTEM\$;DEADXACT

5. 消息的发送

针对消息队列的操作，不外乎三种基本的类型，即往目标队列中发送 (Send) 消息，从某个消息队列中接收 (Receive) 消息和查看 (Peek) 消息。MessageQueue 提供一系列实例方法帮助我们完成这些操作，所以在这之前我们通过上面介绍的方式创建一个针对某个具体消息队列的 MessageQueue 对象。

如下面的代码片段所示，MessageQueue 定义了一系列的 Send 方法重载实现向目标队列发送消息。可以直接将用于封装消息内容的对象传入该 Send 方法 (obj 参数)，也可以给消息加上一个标签 (label 参数)。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public void Send(object obj);
    public void Send(object obj, string label);

    public void Send(object obj, MessageQueueTransaction transaction);
    public void Send(object obj, MessageQueueTransactionType transactionType);
    public void Send(object obj, string label, MessageQueueTransaction
        transaction);
    public void Send(object obj, string label, MessageQueueTransactionType
        transactionType);
}
```

(1) MSMQ 消息的序列化

传入 Send 方法的对象最终需要转换成 MSMQ 消息，这必然需要一个序列化的过程。被应用程序接收的消息则需要被反序列化成相应的对象以便进一步处理。MSMQ 消息的序列化和反序列化工作通过一个叫做消息格式化器的组件来实现，而所有的消息格式化器都实现了具有如下定义的 IMessageFormatter 接口。针对消息（这里的消息是 MSMQ 消息，对应的类型为 System.Messaging.Message，不要和 WCF 消息混淆）的序列化和反序列化分别通过 Write 和 Read 实现。

```
public interface IMessageFormatter : ICloneable
{
    bool CanRead(Message message);
    object Read(Message message);
    void Write(Message message, object obj);
}
```

微软为我们定义了 BinaryMessageFormatter、XmlMessageFormatter 和 ActiveXMessageFormatter 三个消息格式化器。前两个分别采用二进制和 XML（文本）的序列化方式，而 ActiveXMessageFormatter 则采用与 ActiveX 组件兼容的方式进行序列化。在调用 Send 方法时具体采用何种消息格式化器，取决于创建的 MessageQueue 对象。

MessageQueue 具有一个如下定义的 Formatter 可读写属性，该属性返回的消息格式化器用于实现消息的序列化和反序列化。在默认的情况下采用 XmlMessageFormatter 格式化器，

如果需要采用另外的序列化方式，可以通过设置该属性来实现。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public IMessageFormatter Formatter { get; set; }
}
```

(2) 事务控制

除了代表发送消息或者消息主体内容的 `obj` 参数和代表消息标签的 `label` 参数，`Send` 方法还具有两个与事务相关的 `transaction` 和 `transactionType` 参数。其中 `transaction` 的类型为 `MessageQueueTransaction`，代表一个基于 MSMQ 的内部事务。`MessageQueueTransaction` 的定义如下。方法 `Begin`、`Commit` 和 `Abort` 分别用于开启、提交和终止事务。

```
public class MessageQueueTransaction : IDisposable
{
    //其他成员
    public MessageQueueTransaction();
    public void Abort();
    public void Begin();
    public void Commit();
    public void Dispose();
    public MessageQueueTransactionStatus Status { get; }
}
```

`MessageQueueTransaction` 的只读属性 `Status` 表示当前事务的状态，其类型为具有如下定义的 `MessageQueueTransactionStatus` 枚举。4 个枚举值对应着事务的 4 种状态（被终止、被提交、被初始化和未决状态）。

```
public enum MessageQueueTransactionStatus
{
    Aborted,
    Committed,
    Initialized,
    Pending
}
```

如果需要将针对某个目标队列的消息发送操作纳入某个事务中，可以直接将开启的（通过调用 `Begin` 方法）`MessageQueueTransaction` 对象作为 `Send` 方法的参数。具体的编程模式可以参考如下的代码片段，在这段代码中将针对相同队列的多次消息发送操作纳入一个事务之中。

```
string path = @".\Private$\transactionalQ";
MessageQueue queue = new MessageQueue(path);
MessageQueueTransaction transaction = new MessageQueueTransaction();
try
{
    transaction.Begin();
    queue.Send("XXX", transaction);
    queue.Send("YYY", transaction);
    queue.Send("ZZZ", transaction);
    transaction.Commit();
}
```

```

}
catch
{
    transaction.Abort();
    //异常处理
}

```

至于 Send 方法的另一个参数 transactionType, 则是一个 MessageQueueTransactionType 枚举, 代表消息发送操作被纳入事务的方式。MessageQueueTransactionType 枚举具有三个选项, None (默认值) 意味着发送操作并不会被加入任何事务之中, Automatic 则会自动检测到当前的环境事务并自动加入其中, 而 Single 则意味着专门创建一个 MSMQ 内部事务 (相当于 MessageQueueTransaction) 来控制消息的发送。

```

public enum MessageQueueTransactionType
{
    None = 0,
    Automatic = 1,
    Single = 3
}

```

如果采用 TransactionScope 来控制消息的发送, 在 Send 方法中必须显式地传入 MessageQueueTransactionType.Automatic 作为 transactionType 参数的值。下面一段代码与之前采用的 MessageQueueTransaction 具有相同的效果。值得一提的是, 只有针对事务性队列, 这样的事务编程才是有效的。

```

string path = @".\Private$\transactionalQ";
MessageQueue queue = new MessageQueue(path);
using (TransactionScope transactionScope = new TransactionScope())
{
    queue.Send("XXX", MessageQueueTransactionType.Automatic);
    queue.Send("YYY", MessageQueueTransactionType.Automatic);
    queue.Send("ZZZ", MessageQueueTransactionType.Automatic);
    transactionScope.Complete();
}

```

6. MSMQ 消息

虽然可以将一个封装 MSMQ 消息主体内容的对象传入 MessageQueue 的 Send 方法中进行发送, 但是如果我们需要对消息的发送具有更多的控制, 就需要创建一个 MSMQ 消息进行发送。MSMQ 消息通过类型 Message 表示, 下面的代码片段列出了它的构造函数和少数几个属性成员。

```

public class Message : Component
{
    //其他成员
    public Message(object body);
    public Message(object body, IMessageFormatter formatter);

    public object Body { get; set; }
    public Stream BodyStream { get; set; }
}

```

```

public MessagePriority Priority { get; set; }

public Acknowledgment Acknowledgment { get; }
public MessageQueue AdministrationQueue { get; set; }
public MessageQueue ResponseQueue { get; set; }

public bool UseDeadLetterQueue { get; set; }
public bool UseJournalQueue { get; set; }
}

```

我们通过一个可序列化对象和一个消息格式化器（可选，默认采用 `MessageQueue` 的格式化器）创建一个 `Message` 对象。创建时指定的对象作为消息的主体通过 `Body` 属性返回，而 `BodyStream` 是对消息主体进行存取的流。`Message` 的 `Priority` 代表接收方处理消息的优先级，级别高的消息会被优先处理。而 `Acknowledgment` 和 `AdministrationQueue` 则用于指定消息确认（比如消息抵达确认、消息接收确认等）类型和接收确认消息的管理队列。`ResponseQueue` 则表示消息接收方应用发送回复消息的目标队列。两个布尔类型的属性 `UseDeadLetterQueue` 和 `UseJournalQueue` 分别表示是否要使用死信队列和日志队列。

其实 `Message` 类型还有很多额外的属性，但是由于篇幅所限，在这里不能对其进行全面的介绍，对此有兴趣的读者可以参阅 MSDN 中关于 MSMQ 部分的文档。

7. 消息的接收和查看

对于发送到目标队列中的消息，具有接收（Receive）和查看（Peek）两种基本的操作。前者将消息从队列中移出后对其进行相应的处理，后者直接获取消息但消息依然存储在队列中。两种操作分别通过 `MessageQueue` 的 `Receive` 和 `Peek` 方法完成。

```

public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public Message Receive(MessageQueueTransaction transaction);
    public Message Receive(MessageQueueTransactionType transactionType);
    public Message Receive(TimeSpan timeout);
    public Message Receive(TimeSpan timeout, MessageQueueTransaction transaction);
    public Message Receive(TimeSpan timeout, MessageQueueTransactionType transactionType);

    public Message Peek();
    public Message Peek(TimeSpan timeout);
}

```

如上面的代码片段所示，`MessageQueue` 定义了若干 `Receive` 和 `Peek` 方法的重载。不论是调用 `Receive` 方法还是 `Peek` 方法，如果没有显式指定超时时限（`timeout` 参数），方法调用将一直等待，直到从队列中获取一个消息。也就是说，如果队列中根本不存在任何消息，并且在未来一段时间内也不会接收任何消息，这个方法将持久地等待下去。我们可以在调用的时候传入一个 `TimeSpan` 对象控制操作的时间，如果在规定的时间内没有从队列中获取到消息，就会抛出异常。

对于 Receive 方法, 还可以采用与 Send 完全一样的编程方式实现对事务的控制。如果 Receive 方法在一个事务内执行, 只有事务被提交的情况下消息才会从队列中移出来, 在提交前出现的异常不会导致消息的丢失。

除了以同步的方式实现对消息的接收与查看, MessageQueue 还定义了相应的异步方法。如下面的代码片段所示, MessageQueue 具有一系列的 BeginPeek/BeginReceive 方法, 它们以异步的方式完成对消息的接收和查看。当接收和查看操作结束的时候, 会触发 ReceiveCompleted/PeekCompleted 事件。

```
public class MessageQueue : Component, IEnumerable
{
    //其他成员
    public event PeekCompletedEventHandler PeekCompleted;
    public event ReceiveCompletedEventHandler ReceiveCompleted;

    public IAsyncResult BeginPeek();
    public IAsyncResult BeginPeek(TimeSpan timeout);
    public IAsyncResult BeginPeek(TimeSpan timeout, object stateObject);
    public IAsyncResult BeginPeek(TimeSpan timeout, object stateObject,
        AsyncCallback callback);

    public IAsyncResult BeginReceive();
    public IAsyncResult BeginReceive(TimeSpan timeout);
    public IAsyncResult BeginReceive(TimeSpan timeout, object stateObject);
    public IAsyncResult BeginReceive(TimeSpan timeout, object stateObject,
        AsyncCallback callback);
}
```

实际上 MessageQueue 还具有额外的 Receive/Peek 和 BeginReceive/BeginPeek 方法重载, 由于篇幅所限, 不能对其做全面的介绍。对 MSMQ 的介绍就到此为止, 读者如果有兴趣对其进行系统的了解, 可以参阅 MSDN 和相应的技术书籍。

6.2 从队列服务的终结点谈起

我们所说的队列服务基本上可以看成是终结点采用 NetMsmqBinding 作为绑定的服务 (实际上 WCF 中还具有另一个基于 MSMQ 的绑定 MsmqIntegrationBinding, 用于进行与其他的 MSMQ 应用进行集成。不过本章不会涉及对 MsmqIntegrationBinding 的介绍)。NetMsmqBinding 绑定利用 MSMQ 实现消息的传输, 并且提供可靠递交的保障。由于 MSMQ 采用特殊的消息交换方式, 因此作为队列服务的终结点有其特殊之处。

6.2.1 地址

下面看看基于 NetMsmqBinding 的终结点地址具有怎样的格式。MSMQ 在 WCF 中的地址以 “net.msmq://” 为前缀, 其余部分与消息队列的路径差不多, 唯一一点不同是 “\$” 字

符是不需要的。下面列出了公有队列和私有队列的地址格式。有一点需要注意的是，对于本地队列{Host Name}部分，可以用“LocalHost”或者“.”表示，但是不能使用“127.0.0.1”。实际上不论是本地队列还是远程队列，都不能用 IP 地址代替{Host Name}部分。

- 公有地址: net.msmq: //{Host Name}/{Queue Name}
- 私有地址: net.msmq: //{Host Name}/Private/{Queue Name}

除了可以通过 net.msmq 地址标识物理队列的地址之外，子队列也可以通过 net.msmq 地址来表示。子队列的 net.msmq 地址只需要在其主队列的地址上加上“;{SubQueueName}”后缀即可。

net.msmq 地址相当于消息队列的路径。通过 6.1 节的介绍，我们知道除了通过具体的路径去定位某个消息队列之外，一个消息队列还可以通过它的格式名称唯一标识。WCF 提供了 net.msmq 地址和队列的格式名称之间的匹配关系。对于公有格式名称和私有格式名称，需要借助于 AD 的力量来解析机器号和队列号，而 NetMsmqBinding 通过 UseActiveDirectory 表示是否需要使用 AD。

```
public class NetMsmqBinding : MsmqBindingBase
{
    //其他成员
    public bool UseActiveDirectory { get; set; }
}
```

如果采用 Direct 格式名称，需要指定具体采用的协议类型（HTTP/HTTPS、TCP、OS 和 IPX），那么 WCF 在进行 net.msmq 地址和 Direct 格式名称之间的匹配时如何决定采用何种协议呢？这涉及 NetMsmqBinding 如下的属性：QueueTransferProtocol。

```
public class NetMsmqBinding : MsmqBindingBase
{
    //其他成员
    public QueueTransferProtocol QueueTransferProtocol { get; set; }
}
public enum QueueTransferProtocol
{
    Native,
    Srmp,
    SrmpSecure
}
```

NetMsmqBinding 的 QueueTransferProtocol 属性类型为 System.ServiceModel.QueueTransferProtocol 枚举。顾名思义，QueueTransferProtocol 代表访问消息队列所采用的传输协议。有三种协议可供选择（Native、Ermp 和 SrmpSecure），分别代表原生 MSMQ 协议、SRMP（Soap Reliable Messaging Protocol）和 SRMPS（Soap Reliable Messaging Protocol Secure）。SRMP 和 SRMPS 都是基于 SOAP 的协议，它们分别采用 HTTP 和 HTTPS 作为低层传输协议。

在 UseActiveDirectory 属性为 True 的情况下, 只能选择 Native, 基于公有队列和私有队列的 net.msmq 地址将会转换成公有格式名称和私有格式名称。当 UseActiveDirectory 属性被设置为 False 时, Native、Ermp 和 SrmqSecure 将分别转换成基于 OS、HTTP 和 HTTPS 的 Direct 格式名称。

举个例子, 在使用 AD (UseActiveDirectory = True) 的情况下, 两个分别代表公有队列地址和私有队列地址的 net.msmq 地址在 QueueTransferProtocol 为 Native (唯一选择) 的情况下被转换成如下所示的格式名称。

```
net.msmq://Jinnan-PC/MyQueue
    Native: PUBLIC=6847CC75-1B2B-47B3-971D-75F88E3C7FFE
net.msmq://Jinnan-PC/private/MyQueue
    Native: PRIVATE=51DF4CFC-DF17-44A2-AA6F-DDE79049E3DE/MyQueue
```

在不使用 AD (UseActiveDirectory = False) 的情况下, 相同的 net.tcp 地址在三种不同的 QueueTransferProtocol 类型下被转换成如下所示的格式名称。

```
net.msmq://Jinnan-PC/MyQueue
    Native: DIRECT=OS:Jinnan-PC\MyQueue
    Ermp: DIRECT=HTTP://Jinnan-PC/msmq/MyQueue
    SrmqSecure: DIRECT=HTTPS://Jinnan-PC/msmq/MyQueue
net.msmq://Jinnan-PC/private/MyQueue
    Native: DIRECT=OS:Jinnan-PC\private$\MyQueue
    Ermp: DIRECT=HTTP://Jinnan-PC/msmq/private$/MyQueue
    SrmqSecure: DIRECT=HTTPS://Jinnan-PC/msmq/private$/MyQueue
```

6.2.2 绑定

绑定 (NetMsmqBinding) 是队列服务的核心, 后续部分的内容基本上都是围绕这个绑定展开介绍的。在这里仅仅讨论在默认的情况下, NetMsmqBinding 具有怎样的属性, 它们给编程带来怎样的影响。

在工作组模式下, 通过下面一段简单的代码对队列服务进行寄宿。我们为寄宿的服务添加了一个基于 NetMsmqBinding 的终结点, 该终结点使用一个本地的私有队列。

```
using (ServiceHost host = new ServiceHost(typeof(GreetingService)))
{
    host.AddServiceEndpoint("IGreetingService", new NetMsmqBinding(),
        "net.msmq://localhost/private/queue4demo");
    host.Open();
    //...
}
```

但是当 ServiceHost 开启的时候会抛出如图 6-3 所示的 InvalidOperationException 异常, 提示“绑定验证失败, 因为绑定的 MsmqAuthenticationMode 属性被设置为 WindowsDomain 但在安装 MSMQ 时 Active Directory 集成被禁用了。无法打开通道工厂或服务主机”。这是因为默认情况下的 NetMsmqBinding 采用 Transport 安全模式, 并且采用基于 AD 的认证模式。



图 6-3 默认 NetMsmqBinding 在工作组模式下抛出的异常

为了解决这个问题，最直接的方式就是将 NetMsmqBinding 的安全模式设置为 None 或者 Message。相应的改动如下面的代码片段所示。

```
using (ServiceHost host = new ServiceHost(typeof(GreetingService)))
{
    host.AddServiceEndpoint("IGreetingService",
        new NetMsmqBinding(NetMsmqSecurityMode.None),
        "net.msmq://localhost/private/queue4demo");
    //或者
    host.AddServiceEndpoint("IGreetingService",
        new NetMsmqBinding(NetMsmqSecurityMode.Message),
        "net.msmq://localhost/private/queue4demo");
    host.Open();
    //...
}
```

针对上面的这段修正过的服务寄宿代码，如果我们使用的是非事务性队列，那么在寄宿的时候，会抛出如图 6-4 所示的 InvalidOperationException 异常，提示“绑定验证失败，因为绑定的 ExactlyOnce 属性被设置为 true 而目标队列是非事务性的。无法打开服务主机。通过将 ExactlyOnce 属性设置为 false 或者为此绑定创建事务性队列可以解决此冲突”。

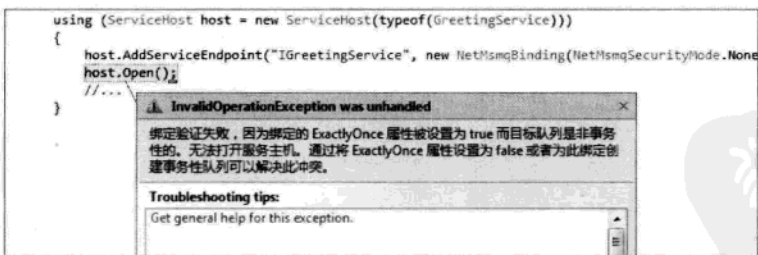


图 6-4 默认 NetMsmqBinding 在使用非事务性队列的情况下抛出的异常

导致这个异常的原因是 NetMsmqBinding 具有一个布尔类型的 ExactlyOnce 属性，表示是否要避免重复的消息。但是只有事务性队列才能提供消息有序递交的机制，而 ExactlyOnce 的默认值为 True。通过编程或者配置的方式将此属性设置为 False 就能解决这个问题。

同一个消息队列只能被某个寄宿的服务单独使用。换句话说，多个服务的终结点地址不能指向同一个消息队列。道理很简单，队列服务针对消息队列进行的是接收（Receive）操

作，而不是查看 (Peek) 操作，被接收的消息最终消息会从队列中移除。如果针对两个不同服务 (Service1 和 Service2) 的终结点共享同一个消息队列，发送给 Service1 的消息可能被 Service2 的终结点接收并丢弃 (因为通过消息筛选机制找不到匹配的终结点，基于消息筛选的终结点选择机制可以参阅本书的第9章“扩展 (Extension)”)。但是一个消息队列可以被同一个服务的多个终结点共享，因为相应的消息可以通过消息筛选机制分发给对应的终结点。

6.2.3 契约

对于一个完整的队列服务来说，其服务和客户端完全是两个独立的应用。客户端无须考虑服务的存在，它的任务是将服务调用转换成 MSMQ 消息并发送往终结点地址所指的消息队列。服务也无须处理实时抵达的客户请求，只需要检测相应消息队列中抵达的 MSMQ 消息并进行处理。

队列服务总是采用单向 (One-Way) 的消息交换模式，这要求队列服务实现的所有服务契约只能定义单向操作。具体来说，就是要求在定义服务契约的时候，必须将应用在操作方法上的 `OperationContractAttribute` 特性的 `IsOneWay` 显式设置为 `True`。当然，单向的操作方法的返回类型只能是 `void`。比如在如下定义的契约接口 `IGreetingService` 中，两个操作 `SayHello` 和 `SayGoodbye` 都是单向操作。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface IGreetingService
{
    [OperationContract(IsOneWay = true)]
    void SayHello(string name);
    [OperationContract(IsOneWay = true)]
    void SayGoodbye(string name);
}
```

在对队列服务进行寄宿的时候，如果发现服务类型实现的契约接口中，有一个或多个操作没有将 `IsOneWay` 设置为 `True`，就会抛出如图 6-5 所示的 `InvalidOperationException` 异常，并提示“协定需要双向(请求-答复或双工)，但是绑定‘NetMsmqBinding’不支持它或者因配置不正确而无法支持它”。

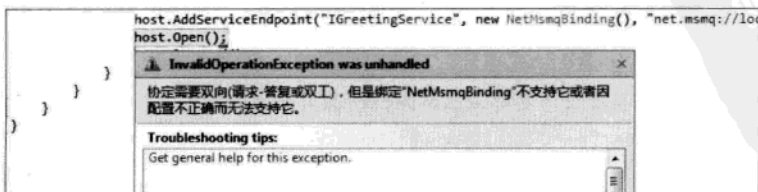


图 6-5 队列服务契约包含非单向操作抛出的异常

由于单向消息交换不会有回复消息返回给客户端，因此客户端也不会捕捉服务端抛出的异常。也正是因为无法进行异常处理，定义在服务操作上的错误契约 (Fault Contract) 也是

没有意义的。不仅如此，在对队列服务进行寄宿的时候，如果服务类型实现的契约接口中的任何一个操作上应用了 `FaultContractAttribute` 特性，也会抛出异常。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface IGreetingService
{
    [OperationContract(IsOneWay = true)]
    [FaultContract(typeof(string))]
    void SayHello(string name);
    [OperationContract(IsOneWay = true)]
    void SayGoodbye(string name);
}
```

比如针对上面定义的契约接口 `IGreetingService`，我在操作方法 `SayHello` 上应用了 `FaultContractAttribute` 特性定义了针对字符串类型的错误契约。在对实现了该接口的服务进行寄宿的时候会抛出如图 6-6 所示的 `InvalidOperationException` 异常，提示“单向方法不能声明 `FaultContractAttribute`”。

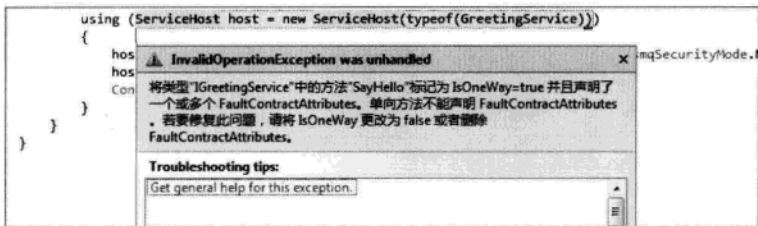


图 6-6 队列契约操作包含错误契约时抛出的异常

6.3 事务控制

MSMQ 旨在解决离线场景的通信问题，这决定了它采用与众不同的事务支持方式。在离线的场景下，应用对消息的发送和消息在客户端和服务端之间的传输并不能保证是连续进行的。如果网络问题导致目标队列不可达，应用程序旨在向目标队列发送的消息只是暂存在本地的出栈消息队列中。等到网络连接正常后才真正地将消息递交到目标队列。

另一方面，对于抵达目标队列中的消息，我们也不能够确定它会在什么时候被目标应用接收处理。总的来说，消息的发送、网络传播和消息的接收这三个基本操作的执行时间是不确定的。而事务旨在确保针对一个短暂业务流程的数据一致性，所以我们不可能通过一个单一的事务涵盖消息从被发送到被接收处理的整个流程。那么 MSMQ 采用怎样的机制实现对事务的管理呢？

6.3.1 MSMQ 事务模型

从消息的整个生命周期来看，消息具有发送、网络传输和接收三个基本的操作。我们不

可能创建一个事务囊括消息的整个生命周期,但是可以利用事务的机制来控制这三个单一的环节。实际上对于事务性队列,上述的这三个基本的操作都是在一个单一的事务中进行的。为了叙述方便,我将分别控制这三个方面的事务称为发送事务 (Sending Transaction)、(网络) 递交事务 (Delivery Transaction) 和接收事务 (Receiving Transaction)。

如图 6-7 所示,发送事务控制消息被发送到本地队列。这意味着只有在事务被提交之后,消息才会被真正存储于本地的队列中。递交事务控制消息在本地队列到目标队列之间的网路传输,如果事务失败,消息依然应该存储于本地队列中而不至于丢失。缓存于本地队列中的消息只有在事务被成功提交的情况下抵达目标队列才会被删除。而接收事务则控制应用程序从目标队列中对消息的接收操作,消息只有在事务成功提交的情况下才会从目标队列中删除,否则它依然会存在于队列之中。

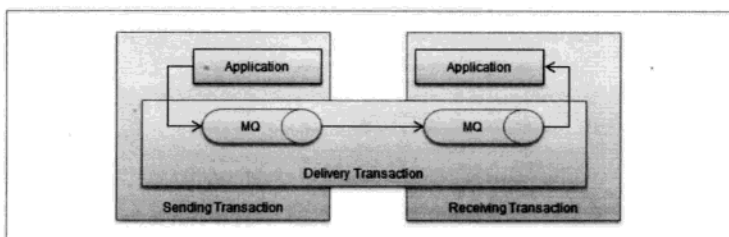


图 6-7 MSMQ 事务模型

由于事务性队列属于资源管理器 (RM, Resource Manager), 在执行发送和接收操作的时候具有环境事务的探测和登记能力, 因此可以利用 TransactionScope 将消息的发送/接收和相关的业务操作组成一个单一的事务。关于事务相关的内容, 可以参阅本书的第 3 章“事务 (Transaction)”。

6.3.2 客户端事务

队列服务的一个重要的特征就是服务和其客户端之间的独立性。它们面对的其实不是彼此, 而是用于进行消息传输和存储的队列。客户端将所谓的服务调用转换成 MSMQ 消息发送到目标队列之中, 服务端只需要监听目标队列并接收和处理其中的消息。

正因如此, 在本书第 3 章“事务 (Transaction)”中介绍的事务流转 (Transaction Flowing) 在这里不适用。服务流转旨在将服务操作的执行纳入从客户端开始的事务。而对于队列服务, 不但服务操作不可能加入客户端事务, 该事务也不能控制消息在客户端队列和服务端队列之间的传输。

正因为队列服务不支持事务的流转, 所以在定义服务契约的时候不能在操作方法上应用 TransactionFlowAttribute 特性并将参数设置为 TransactionFlowOption.Mandatory。之所以不能通过如此的特性定义强制进行事务流转, 其根本原因在于 NetMsmqBinding 不具有事务传播的能力 (不包含 TransactionFlowBindingElement 元素)。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface IHelloWorld
{
    [TransactionFlow(TransactionFlowOption.Mandatory)]
    [OperationContract(IsOneWay = true)]
    void SayHello(string name);
}
```

如上面的代码片段所示，我们在契约接口 `IHelloWorld` 的操作方法 `SayHello` 上应用了一个参数为 `TransactionFlowOption.Mandatory` 的 `TransactionFlowAttribute` 特性，要求强制进行事务的传播。在进行寄宿的时候，会抛出如图 6-8 所示的 `InvalidOperationException` 异常，提示“‘HelloWorldService’ 协定上至少有一个操作配置为将 `TransactionFlowAttribute` 特性设置为‘强制’，但是通道的绑定‘`NetMsmqBinding`’未使用 `TransactionFlowBindingElement` 进行配置。没有 `TransactionFlowBindingElement`，无法使用设置为‘强制’的 `TransactionFlowAttribute` 特性”。

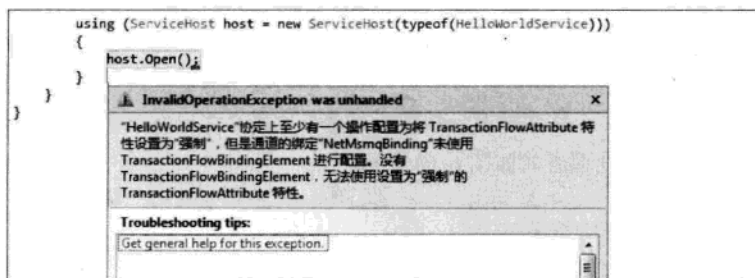


图 6-8 服务契约强制事务流转导致的异常

对于事务性队列，消息的发送总是在一个事务中进行的。如果服务调用并没有在事务中执行，那么上述的发送事务会在消息发送的那一刻开始，并在发送成功的时候提交。这就带来了一个问题：如果客户端操作在进行服务调用之后抛出异常，消息一样会发送到本地的消息队列并最终向目标队列发送，最终也会导致服务操作的执行。

为了避免这样的问题发送，应该通过 `TransactionScope` 让包括服务调用的所有操作纳入同一个事务中执行。MSMQ 在进行消息的发送时会探测到环境事务的存在，并且会将消息的发送操作自动纳入这个环境事务之中执行。当所有的操作正常结束之后，MSMQ 才会真正将消息发送到本地的消息队列之中。这两种不同的事务控制方式如图 6-9 所示。

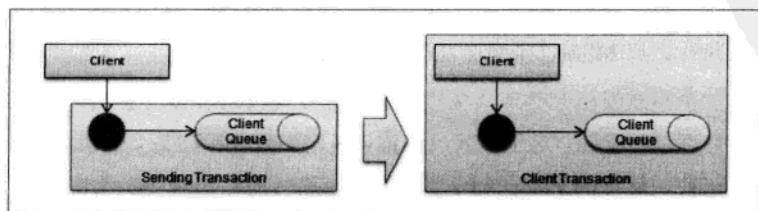


图 6-9 非事务客户端与事务客户端

除了提供基本的事务提交和回滚之外,发送事务还提供了“重发”机制,它和接收事务的“重收”机制一起实现对消息的可靠传输。如果 MSMQ 在试图向本地队列发送消息的过程中发生异常,它会自动地进行重发。对于可解决性错误的发送失败,这样的重发机制能够确保消息最终能够被成功发送出去,否则在重试次数超出指定的限额时会终止尝试。

6.3.3 服务端事务

对于事务性队列,应用程序从队列中接收消息总是在接收事务中进行的,一旦消息成功接收,便会从队列中移除。对于队列服务来说,在执行服务操作之前消息已经从队列中成功获取出来了,如果在执行服务操作过程中出现异常,消息也回不到目标队列之中。相同的情况同样出现在单独为服务操作开启事务的场景中,因为服务操作的事务和 MSMQ 接收的接收事务是两个独立的事务。

实际上这样的问题是不能接受的,我们乐于接受的是这样的事务处理方式:服务操作的执行也被纳入 MSMQ 的接收事务中来。换句话说,就是让消息的接收和操作的执行共享同一个事务。如果事务中出现任何的异常都将导致整个事务的回滚,而最终确保消息还是保存在队列之中。这里介绍的三种处理方式如图 6-10 所示。

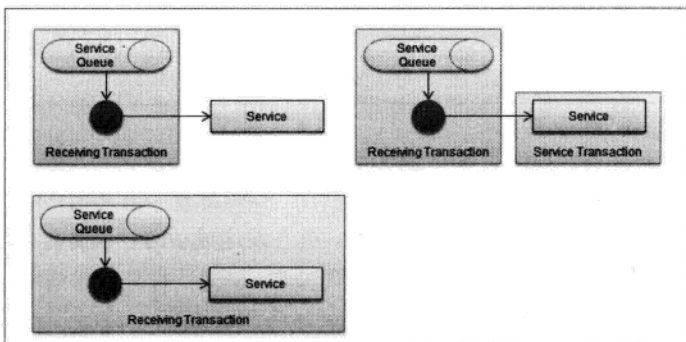


图 6-10 服务端异常的三种处理方式

要实现消息接收和操作执行的事务共享,只需要在定义在服务类型上的相应操作上应用 `OperationBehaviorAttribute` 特性将 `TransactionScopeRequired` 属性设置为 `True` 即可。由于该特性的 `TransactionAutoComplete` 属性默认值为 `True`,意味着操作执行结束时事务会自动提交,也就是说当操作完全执行之后 MSMQ 消息会从队列中删除。

与基于发送事务的消息重发机制类似,在接收事务接收失败之后 MSMQ 自动实施重收机制,而重收的次数是可以配置的。对于可解决性错误(比如数据库在某个高负载情况下发生死锁)导致的接收失败,这样的重收机制可以确保消息最终能够被成功接收。

发送事务和接收事务分别在消息的发送端和接收端实现了失败后的重试机制,提供了不同于通过可靠会话提供的消息可靠传输机制。除此之外,基于 `NetMsmqBinding` 的队列服务

还能够避免相同消息的重复接收，意味着服务端从网络中可能接收多个相同的消息，但是却能保证向上递交唯一的消息。消息接收的唯一性通过 `NetMsmqBinding` 继承的父类 `MsmqBindingBase` 的属性 `ExactlyOnce` 表示。

```
public abstract class MsmqBindingBase: Binding, IBindingRuntimePreferences
{
    //其他成员
    [DefaultValue(true)]
    public bool ExactlyOnce { get; set; }
    [DefaultValue(true)]
    public bool Durable { get; set; }
}
```

从上面的代码片段也可以看出，`ExactlyOnce` 属性在默认情况下的值为 `True`，而非事务性消息队列不具有确保接收唯一性的能力，所以在默认情况下才会出现在前面提到过的 `InvalidOperationException` 异常。而对于事务性消息队列来说，接收唯一性的保障是其与生俱来的能力，所以我们又不能将此 `ExactlyOnce` 属性设置为 `False`，否则在服务寄宿时也会抛出 `InvalidOperationException` 异常，提示绑定采用的消息队列（事务性消息队列）与 `ExactlyOnce` 属性值不相匹配（`ExactlyOnce = False`）。

除了 `ExactlyOnce`，`MsmqBindingBase` 还具有另一个布尔类型的属性 `Durable`，表示是否会对 MSMQ 消息进行持久化保存。对消息的持久化保存可以确保在进程中止甚至是主机重启的情况下消息不至于丢失。如上面的代码所示，`Durable` 属性的默认值为 `True`。

6.3.4 事务性批量接收

在默认的情况下，服务端用于消息接收的事务仅针对一条单一的 MSMQ 消息。为了提高性能，增强服务的吞吐量，WCF 提供了批量消息接收机制。所谓事务性批量接收，就是将针对多个 MSMQ 消息的接收纳入同一个事务之中进行批量处理。批量消息接收通过终结点行为 `TransactedBatchingBehavior` 实现。

1. TransactedBatchingBehavior 终结点行为

如果需要采用针对多 MSMQ 消息的批量处理，可以在服务相应的终结点上应用行为 `System.ServiceModel.Description.TransactedBatchingBehavior` 来实现。如下面的代码所示，除了实现终结点行为接口 `IEndpointBehavior` 的 4 个方法外，`TransactedBatchingBehavior` 具有一个整型的 `MaxBatchSize` 属性，表示进行批量处理的最大消息数量。默认值为 0，即在默认的情况下根本不是“批量的”。

```
public class TransactedBatchingBehavior : IEndpointBehavior
{
    //其他成员
    public int MaxBatchSize { get; set; }
}
```

可以通过配置的方式将 `TransactedBatchingBehavior` 行为应用到服务终结点相应的终结点上。相应的配置可以参考如下的 XML 片段。在这段配置中，定义了一个包含有 `TransactedBatchingBehavior` 的终结点行为，批量处理最大允许的消息数量被设置为 20。

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <endpointBehaviors>
        <behavior name="batchingMessages">
          <transactedBatching maxBatchSize="20" />
        </behavior>
      </endpointBehaviors>
    </behaviors>
    <services>
      <service name="...">
        <endpoint behaviorConfiguration="batchingMessages" .../>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

并不是任何终结点都可以应用 `TransactedBatchingBehavior` 行为，实际上该行为对终结点本身具有一定的要求。`TransactedBatchingBehavior` 首先要求目标终结点的绑定具有一个实现了具有如下定义的 `System.ServiceModel.Channels.ITransactedBindingElement` 接口的绑定元素，并且要该绑定元素的 `TransactedReceiveEnabled` 属性为 `True`。

```
public interface ITransactedBindingElement
{
    bool TransactedReceiveEnabled { get; }
}
```

到目前为止，只有 `MsmqTransportBindingElement` 和 `MsmqIntegrationBindingElement` 这两个基于 MSMQ 的绑定元素的基类 `MsmqBindingElementBase` 实现了该接口。至于其属性 `TransactedReceiveEnabled`，则与 `ExactlyOnce` 具有相同的取值。原因很简单，只有针对事务性队列的 `NetMsmqBinding` 的 `ExactlyOnce` 属性为 `True`，而消息的批量接收处理显然需要采用事务性队列。如果我们将 `TransactedBatchingBehavior` 行为应用到非 MSMQ 绑定的终结点，或者是 MSMQ 绑定的 `ExactlyOnce` 属性被设置成 `False`，都是不合法的。

```
public abstract class MsmqBindingElementBase : TransportBindingElement,
ITransactedBindingElement, ...
{
    //其他成员
    public bool ExactlyOnce { get; set; }
    public bool TransactedReceiveEnabled
    {
        get { return this.ExactlyOnce; }
    }
}
```

`TransactedBatchingBehavior` 终结点行为还对终结点的契约有相应的要求，服务契约的会话模式不能是 `SessionMode.Required`。此外还要求分发运行时的 `ReleaseServiceInstance`

OnTransactionComplete 属性不能为 True。下面定义的服务契约和服务类型的终结点都不能应用 TransactedBatchingBehavior 行为，否则会在寄宿的时候会抛出 InvalidOperationException。

服务契约：

```
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IGreeting
{
    [OperationContract(IsOneWay = true)]
    void SayHello(string name);
    [OperationContract(IsOneWay = true)]
    void SayGoodbye(string name);
}
```

服务类型：

```
[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete = true)]
public class GreetingService : IGreeting
{
    //省略成员
}
```

最后需要说明一点，消息的事务性批量接收处理模式仅限于应用了 OperationBehavior Attribute 特性并将 TransactionScopeRequired 和 TransactionAutoComplete（默认为 True）属性设置为 True 的操作。

2. 实例演示：如何进行消息的批量接收处理（S601）

接下来通过一个简单的例子来演示如何应用 TransactedBatchingBehavior 终结点行为实现对消息的事务性批量接收处理。这个实例还将用于证明针对多个消息的处理确实是在同一个事务中进行的。我们首先定义了如下一个简单的类似于 HelloWorld 的契约接口。

```
using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface IGreeting
    {
        [OperationContract(IsOneWay = true)]
        void SayHello(string name);
        [OperationContract(IsOneWay = true)]
        void SayGoodbye(string name);
    }
}
```

下面是实现了这个契约接口的服务类型。值得注意的是，我们在服务类型上应用了 ServiceBehaviorAttribute 特性将 ReleaseServiceInstanceOnTransactionComplete 属性显式设置为 False。在两个操作方法中以控制台打印出了当前事务的分布式 ID（DistributedIdentifier）。我们就是通过这个值来确定操作执行所在的事务的。两个操作方法均应用了 Operation

BehaviorAttribute 特性并将 TransactionScopeRequired 属性设置为 True。

```
using System;
using System.ServiceModel;
using System.Transactions;
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Service
{
    [ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete = false)]
    public class GreetingService : IGreeting
    {
        [OperationBehavior(TransactionScopeRequired = true)]
        public void SayHello(string name)
        {
            Console.WriteLine("[{1}]Hello, {0}", name,
                Transaction.Current.TransactionInformation.Distributed
                Identifier);
        }
        [OperationBehavior(TransactionScopeRequired = true)]
        public void SayGoodbye(string name)
        {
            Console.WriteLine("[{1}]Goodbye, {0}", name,
                Transaction.Current.TransactionInformation.Distributed
                Identifier);
        }
    }
}
```

我们采用控制台应用对上面定义的 GreetingService 进行寄宿, 下面给出了配置和服务寄宿程序。从这段配置中可以看到, 寄宿服务的唯一终结点应用了一个名称为 batchingMessages 的行为。该终结点行为中包含一个 MaxBatchSize 属性为 2 的 TransactedBatchingBehavior 终结点行为。

配置:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding name="bindingWithNoneSecurityMode">
          <security mode="None"/>
        </binding>
      </netMsmqBinding>
    </bindings>
    <behaviors>
      <endpointBehaviors>
        <behavior name="batchingMessages">
          <transactedBatching maxBatchSize="2"/>
        </behavior>
      </endpointBehaviors>
    </behaviors>
    <services>
      <service name="Artech.WcfServices.Service.GreetingService">
        <endpoint
          address="net.msmq://./private/XactQueue4Demo"
          binding="netMsmqBinding"
          bindingConfiguration="bindingWithNoneSecurityMode">
```

```

        behaviorConfiguration="batchingMessages"
        contract="Artech.WcfServices.Service.Interface.IGreeting"/>
    </service>
</services>
</system.serviceModel>
</configuration>

```

寄宿程序:

```

string path = @".\private$\XactQueue4Demo";
if (!MessageQueue.Exists(path))
{
    MessageQueue.Create(path, true);
}

using (ServiceHost host = new ServiceHost(typeof(GreetingService)))
{
    host.Open();
    Console.Read();
}

```

客户端分别调用 4 个服务代理针对服务的两个操作 SayHello 和 SayGoodbye 进行调用。下面是相应的配置和服务调用程序。

配置:

```

<configuration>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding name="bindingWithNoneSecurityMode">
          <security mode="None"/>
        </binding>
      </netMsmqBinding>
    </bindings>
    <client>
      <endpoint name="greetingService"
        address="net.msmq://localhost/private/XactQueue4Demo"
        binding="netMsmqBinding"
        bindingConfiguration="bindingWithNoneSecurityMode"
        contract="Artech.WcfServices.Service.Interface.IGreeting"/>
    </client>
  </system.serviceModel>
</configuration>

```

服务调用程序:

```

IGreeting proxy;
using (ChannelFactory<IGreeting> channelFactoryHello = new
ChannelFactory<IGreeting>("greetingService"))
{
    proxy = channelFactoryHello.CreateChannel();
    proxy.SayHello("Foo");

    proxy = channelFactoryHello.CreateChannel();
    proxy.SayGoodbye("Bar");

    proxy = channelFactoryHello.CreateChannel();
}

```

```

proxy.SayHello("Foo");

proxy = channelFactoryHello.CreateChannel();
proxy.SayGoodbye("Bar");
}

```

现在先运行客户端程序，它会发送 4 个 MSMQ 消息到目标队列。然后启动服务寄宿程序，会得到如下所示的输出结果。输出结果可以证明前两个操作和后两个操作共享不同的事务。这是因为我们将应用到终结点上的 `TransactedBatchingBehavior` 行为的 `MaxBatchSize` 设置为 2，意味着它在一个以事务代表的批次中最多能够处理两条 MSMQ 消息。

```

[04335678-c172-4341-8f94-7d3b8a2845e8]Hello, Foo
[04335678-c172-4341-8f94-7d3b8a2845e8]Goodbye, Bar

[073285a1-75be-4b9b-b2c0-ac808baf2345]Hello, Foo
[073285a1-75be-4b9b-b2c0-ac808baf2345]Goodbye, Bar

```

6.4 会话

会话通过建立客户端与服务之间的上下文来保持来源于相同客户端的多次服务调用的状态，它使得为单一服务调用而进行的消息交换不再是客户端与服务之间独立的对话 (Dialog)，而是将基于相同客户端的所有消息交换构成一个具有语境的会话。由于通信方式的不同，使得会话在队列服务中的行为表现得与众不同。

6.4.1 客户端会话

在定义服务契约的时候，可以通过应用在契约接口上的 `ServiceContractAttribute` 特性的 `SessionMode` 属性定义三种不同的会话模式 (`Allowed`、`Required` 和 `NotAllowed`)。对队列服务来说，只有会话模式为 `Required` 的时候才会真正采用会话的消息交换方式，而当选择 `Allowed` 和 `NotAllowed` 时具有相同的效果。换句话说，即使你选择 `Allowed` 作为会话模式，也得不到会话的支持。

非会话的队列服务并没有太多特殊之处，WCF 总是将单一服务调用转换成一个 MSMQ 消息发送到目标服务。所以本节我们主要介绍客户端针对于会话服务应该采用怎样的编程方式，以及背后的消息交换是如何完成的。

1. 会话与事务

队列服务的会话是通过事务来实现的，所有针对会话服务的调用必须在一个事务中进行。比如我们定义了如下一个简单的契约接口 `IGreeting`，会话模式被设置为 `SessionMode.Required`。

```

using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface

```

```

{
    [ServiceContract(Namespace = "http://www.artech.com/",
        SessionMode = SessionMode.Required)]
    public interface IGreeting
    {
        [OperationContract(IsOneWay=true)]
        void SayHello(string name);
        [OperationContract(IsOneWay = true)]
        void SayGoodbye(string name);
    }
}

```

客户端在进行服务调用的时候，必须将调用代码置于如下一个 `TransactionScope` 中，即让隶属于同一个会话的多次服务调用被纳入同一个事务。

```

using (ChannelFactory<IGreeting> channelFactoryHello = new
ChannelFactory<IGreeting>("greetingService"))
{
    using (TransactionScope scope = new TransactionScope())
    {
        IGreeting proxy = channelFactoryHello.CreateChannel();
        proxy.SayHello("Foo");
        proxy.SayGoodbye("Bar");
        (proxy as ICommunicationObject).Close();
        scope.Complete();
    }
}

```

针对于会话队列服务的调用，WCF 会检查当前环境事务 (`Transaction.Current`) 是否存在，如果不存在就会抛出异常。所以如果将上面的代码片段中事务控制的相关代码移除，在调用 `SayHello` 操作的时候就会抛出如图 6-11 所示的 `InvalidOperationException` 异常，提示“在 `Transaction.Current` 中找不到事务，但此操作需要事务。无法打开通道。确保在事务范围内调用此操作”。

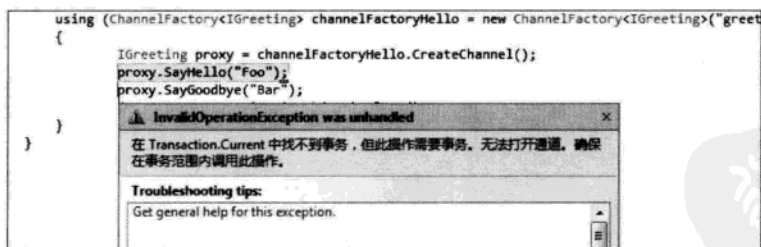


图 6-11 环境事务不存在的情况下调用会话服务导致的异常

会话的生命周期不能超出事务的生命周期。换句话说，必须在提交事务之前先将服务代理关闭。所以在上面的代码中在调用 `TransactionScope` 的 `Complete` 方法之前先关闭服务代理的代码是必需的。如果在正式提交事务（通过调用 `TransactionScope` 的 `Dispose` 方法）之前服务调用的代理对象不曾关闭，将会得到如图 6-12 所示的 `TransactionAbortedException` 异常。

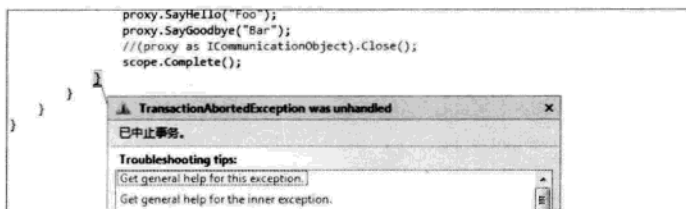


图 6-12 调用会话服务的代理没有关闭时提交事务导致的异常

2. 会话中的消息交换

WCF 是一个完全基于消息的通信框架，我们通过消息交换层面来分析其内部运行机制往往能够抓住其本质。一般会话服务体现在会话信道的重用，一次服务调用体现在一次单一的消息交换。而队列服务则完全不一样，因为不论一个会话中具有多少次服务调用，都只会进行唯一一次消息交换。

如果针对同一个服务代理对象在一个会话（事务）中具有多次服务调用，WCF 并不会为每次服务调用创建相应的 MSMQ 消息，而是在事务提交的时候将针对所有服务调用生成的 WCF 消息组装成一个 MSMQ 消息进行发送。所以一个会话对应一次单一的消息交换，而 WCF 消息在 MSMQ 消息中的顺序反映了操作调用的顺序。为了验证队列服务基于单一消息交换的会话特性，我们不妨做一个简单的测试实例。

我们针对上面定义的服务契约 `IGreeting` 进行服务调用，下面是客户端配置。我们采用的是一个基于 MSMQ 传输的自定义绑定，该绑定采用基于文本的消息编码。为了适应工作组模式环境，我们将认证方式和消息保护级别都设置为 `None`。而终结点的路径指向的是本地的事务性队列 `"XactQueue4Demo"`。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="myMsmqBinding">
          <textMessageEncoding/>
          <msmqTransport>
            <msmqTransportSecurity msmqAuthenticationMode="None"
                                  msmqProtectionLevel="None"/>
          </msmqTransport>
        </binding>
      </customBinding>
    </bindings>
    <client>
      <endpoint name="greetingService"
                address="net.msmq://localhost/private/XactQueue4Demo"
                binding="customBinding"
                bindingConfiguration="myMsmqBinding"
                contract="Artech.WcfServices.Service.Interface.IGreeting"/>
    </client>
  </system.serviceModel>
</configuration>
```

在控制台应用中,编写了如下的测试程序,旨在查看针对多次会话服务调用后的 MSMQ 消息的内容。按照上面规定的方式在一个事务内针对创建的同一个服务代理进行了两次服务调用(SayHello 和 SayGoodbye)。在调用之前我们对目标队列进行了清理(调用 MessageQueue 的 Purge 方法),在调用后接收并打印出 MSMQ 消息的内容。

```
var queue = new MessageQueue(@".\private$\XactQueue4Demo");
queue.Purge();

using (ChannelFactory<IGreeting> channelFactoryHello = new
ChannelFactory<IGreeting>("greetingService"))
{
    using (TransactionScope scope = new TransactionScope())
    {
        IGreeting proxy = channelFactoryHello.CreateChannel();
        proxy.SayHello("Foo");
        proxy.SayGoodbye("Bar");
        (proxy as ICommunicationObject).Close();
        scope.Complete();
    }
}

var message = queue.Receive();
byte[] buffer = new byte[message.BodyStream.Length];
message.BodyStream.Read(buffer, 0, buffer.Length);
Console.WriteLine(Encoding.Default.GetString(buffer));
```

下面的 XML 片段代表经过整理后的输出内容,从中可以看出里面包含两个 SOAP 消息,分别是对服务操作 SayHello 和 SayGoodbye 的调用(通过<a:Action>报头)。这充分证实了队列表服务基于单一消息交换的会话特性。(S102)

```
...<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">
      http://www.artech.com/IGreeting/SayHello</a:Action>
    <VsDebuggerCausalityData
      xmlns="http://schemas.microsoft.com/vstudio/diagnostics/servicemodelsink">...
    </VsDebuggerCausalityData>
    <a:To s:mustUnderstand="1">net.msmq://localhost/private/XactQueue4Demo</a:To>
  </s:Header>
  <s:Body>
    <SayHello xmlns="http://www.artech.com/">
      <name>Foo</name>
    </SayHello>
  </s:Body>
</s:Envelope>
...<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://www.artech.com/IGreeting/SayGoodbye
    </a:Action>
    <VsDebuggerCausalityData
      xmlns="http://schemas.microsoft.com/vstudio/diagnostics/servicemodelsink">...
    </VsDebuggerCausalityData>
    <a:To s:mustUnderstand="1">net.msmq://localhost/private/XactQueue4Demo</a:To>
```

```

</s:Header>
<s:Body>
  <SayGoodbye xmlns="http://www.artech.com/">
    <name>Bar</name>
  </SayGoodbye>
</s:Body>
</s:Envelope>

```

6.4.2 服务端会话

队列服务的会话基于事务的实现反映在客户端就是在会话内部的多个服务调用需要在—一个事务中进行，我们可以借助于 `TransactionScope` 来实现。而对于服务来说，需要将所有服务的操作纳入 MSMQ 的接收事务中执行，所以需要在会话服务类型的所有操作方法上应用 `OperationBehaviorAttribute` 特性并将 `TransactionScopeRequired` 属性设置为 `True`。

```

public class GreetingService : IGreeting
{
    [OperationBehavior(TransactionScopeRequired = true)]
    public void SayHello(string name)
    {
        Console.WriteLine("Hello, {0}", name);
    }
    [OperationBehavior(TransactionScopeRequired = true)]
    public void SayGoodbye(string name)
    {
        Console.WriteLine("Goodbye, {0}", name);
    }
}

```

在上面的代码片段中，我们通过实现契约接口 `IGreeting` 接口定义了服务类 `GreetingService`，并在实现的两个操作方法 `SayHello` 和 `SayGoodbye` 上应用了 `OperationBehaviorAttribute` 特性将 `TransactionScopeRequired` 属性显式设置为 `True`。那么至此就完事大吉了吗？下面不妨进行一个简单的实例演示如此定义的会话服务是否能够正常工作。

1. 实例演示：让会话服务操作 `TransactionScopeRequired=True` 就足够了吗 (S603)

为了让读者对如何正确地定义会话服务有深刻的印象，下面来演示一下前面我们定义的实现基于会话模式的契约接口 `IGreeting` 的服务 `GreetingService` 能否正常地工作。现在我们直接采用控制台应用对 `GreetingService` 服务进行寄宿，下面是配置和寄宿程序。

配置：

```

<configuration>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding name="bindingWithNoneSecurityMode">
          <security mode="None"/>
        </binding>
      </netMsmqBinding>
    </bindings>
  </system.serviceModel>
</configuration>

```



```

</bindings>
<services>
  <service name="Artech.WcfServices.Service.GreetingService">
    <endpoint address="net.msmq://./private/XactQueue4Demo"
      binding="netMsmqBinding"
      bindingConfiguration="bindingWithNoneSecurityMode"
      contract="Artech.WcfService.Service.Interface.IGreeting"/>
  </service>
</services>
</system.serviceModel>
</configuration>

```

寄宿程序:

```

string path = @".\private$\XactQueue4Demo";
if (!MessageQueue.Exists(path))
{
    MessageQueue.Create(path, true);
}

using (ServiceHost host = new ServiceHost(typeof(GreetingService)))
{
    host.Open();
    Console.Read();
}

```

接下来同样采用一个控制台应用程序作为客户端来调用这个队列服务,下面是配置和服务调用程序(采用前面介绍的正确会话队列服务调用方式)。

配置:

```

<configuration>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding name="bindingWithNoneSecurityMode">
          <security mode="None"/>
        </binding>
      </netMsmqBinding>
    </bindings>
    <client>
      <endpoint name="greetingService"
        address="net.msmq://localhost/private/XactQueue4Demo"
        binding="netMsmqBinding"
        bindingConfiguration="bindingWithNoneSecurityMode"
        contract="Artech.WcfServices.Service.Interface.IGreeting"/>
    </client>
  </system.serviceModel>
</configuration>

```

服务调用程序:

```

using (ChannelFactory<IGreeting> channelFactoryHello = new
ChannelFactory<IGreeting>("greetingService"))
{
    using (TransactionScope scope = new TransactionScope())
    {
        IGreeting proxy = channelFactoryHello.CreateChannel();
    }
}

```

```

        proxy.SayHello("Foo");
        proxy.SayGoodbye("Bar");
        (proxy as ICommunicationObject).Close();
        scope.Complete();
    }
}

```

输出结果:

```

Hello, Foo
Hello, Foo
Hello, Foo
Hello, Foo
Hello, Foo
Hello, Foo

```

分别启动服务寄宿程序和客户端应用程序 (先后顺序没有限制), 你会发现在服务端出现了无论如何也想象不到的结果。我们说这个输出的结果显得难以理解主要体现在如下三点。

- 没有任何异常发生;
- 客户端先后调用了 SayHello 和 SayGoodBye 操作, 但是在服务端只执行了 SayHello 操作;
- SayHello 被执行了 6 次。

2. 会话队列服务应该如何定义

从上面的例子中我们知道了定义一个支持会话的队列服务并没有我们想象得那么简单。那么真正的会话队列服务应该采用怎样的编程规范呢? 要回答这个问题, 先得搞清楚上面演示的实例中那个看似“无厘头”的输出结果是如何产生的。

通过前面的介绍我们知道, 客户端两次服务调用的结果只会向目标队列发送一个 MSMQ 消息, 但这个 MSMQ 消息包含两个基于两次服务调用的 WCF 消息 (SOAP 消息)。服务会接收这个 MSMQ 消息, 并将其拆分成两个 WCF 消息, 最后作为输入调用激活服务实例的两个操作方法 SayHello 和 SayGoodbye。由于默认情况下采用会话实例上下文模式, 所以两次操作调用针对的是同一个服务实例。

由于消息是从事务性队列中接收的, 并且在 GreetingService 的两个操作方法 SayHello 和 SayGoodbye 上都通过 OperationBehaviorAttribute 特性将 TransactionScopeRequired 设置为 True, 因此先后执行的两个方法都应该在消息的接收事务中执行。所以事务的提交应该在 SayHello 和 SayGoodbye 两个操作执行结束后进行。或者说应该在当前会话结束之后才能提交事务。中间针对单一操作的事务提交是不允许的。

但是在默认的情况下, 事务会在什么时候提交呢? 这就涉及分发操作 (DispatchOperation) 的 TransactionAutoComplete 属性了, 它控制着是否在操作方法执行完成之后自动提交当前事务。默认情况下该属性值为 True, 意味着如果当前的环境事务存在, 在执行完该操作之后当

前事务会自动被提交。这种基于操作的事务提交和我们期望的基于会话的事务提交是相违背的。

```
public sealed class DispatchOperation
{
    //其他成员
    public bool TransactionAutoComplete { get; set; }
}
```

由于会话服务需要基于会话的事务提交方式，因此在会话结束前试图提交事务会导致接收事务的中止，而事务的中止则会导致接收的消息再次回到目标队列之中。这也正是为什么在上面的例子中只有 SayHello 操作被执行的原因，因为 SayHello 操作方法执行完成之后对事务的自动提交会导致接收事务的中止，因此不会再继续执行后续的操作。

那么为什么 SayHello 会执行 6 次呢？这是源于前面我们提到过的为了实现消息的可靠传输而进行的消息重新接收机制。SayHello 执行后对事务的提交导致事务的中止，进而促使接收的消息最终回到消息队列。而消息的重新接收机制被启动，在默认的情况下重新尝试的次数为 5，所以 SayHello 总共会执行 6 次。

知道了问题的症结后需要解决问题。按照我们的分析，如果将事务的提交从操作方法执行结束转变成整个会话结束，那么就能够解决这个问题。我们需要阻止服务操作结束之后“自作主张”地提交事务，这可以通过将 OperationBehaviorAttribute 特性应用到操作方法上并将 TransactionAutoComplete 属性设置为 False 来解决。为了让会话结束之后对事务进行自动的提交，可以将 ServiceBehaviorAttribute 特性应用到服务类型上并将 TransactionAutoComplete OnSessionClose 属性设置为 True。

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute,
    IOperationBehavior
{
    //其他成员
    public bool TransactionAutoComplete { get; set; }
}

[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public bool TransactionAutoCompleteOnSessionClose { get; set; }
}
```

对于会话队列服务，不但需要让服务的操作自动加入到 MSMQ 的消息接收事务，还需要在会话结束之后自动地提交事务。针对上面演示的 GreetingService，下面才是正确的定义方式。（S104）

```
[ServiceBehavior(TransactionAutoCompleteOnSessionClose = true)]
public class GreetingService : IGreeting
{
    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
```

```

public void SayHello(string name)
{
    Console.WriteLine("Hello, {0}", name);
}
[OperationBehavior(TransactionScopeRequired = true,
    TransactionAutoComplete = false)]
public void SayGoodbye(string name)
{
    Console.WriteLine("Goodbye, {0}", name);
}
}

```

6.5 错误处理

虽然队列服务在客户端和服务端提供了消息的重传和重收机制，能够在一定程度上确保消息能够被成功地发送和接收，但是有的问题并不是简单地靠重试就能够解决的。这里介绍的错误处理分别指在服务端和客户端如何解决这些没能成功处理的消息。

6.5.1 接收重试

首先关注的是服务端的错误处理机制，先来看看我们提过很多遍的消息接收的重试机制。为了确保服务应用能够成功地从目标队列中接收消息，WCF 提供了多批次的消息接收重试机制。我们可以分批次地进行消息接收重试，批次数量、每个批次重试的次数及相邻批次之间的间隔时间都是可以在 `NetMsmqBinding` 上进行配置的。

如下面的代码片段所示，在 `NetMsmqBinding` 的基类 `MsmqBindingBase` 上定义了三个属性。其中整型的 `MaxRetryCycles` 属性代表的是重试批次的最大值，默认情况下为 2。整型属性 `ReceiveRetryCount` 代表在每个批次内重新尝试的次数，默认值为 5。在之前演示的实例中，因在操作执行后提交事务导致操作被自动执行 6 次的根源就是这个 `ReceiveRetryCount` 属性。`TimeSpan` 类型的 `RetryCycleDelay` 代表相邻的两个批次之间的间隔时间，默认值为 30 分钟。

```

public abstract class MsmqBindingBase : Binding, IBindingRuntimePreferences
{
    //其他成员
    [DefaultValue(2)]
    public int MaxRetryCycles { get; set; }
    [DefaultValue(5)]
    public int ReceiveRetryCount { get; set; }
    [DefaultValue(typeof(TimeSpan), "00:30:00")]
    public TimeSpan RetryCycleDelay { get; set; }
}

```

我们可以通过编程或者配置的方式为采用的 `NetMsmqBinding` 的这三个值进行相应的设置来定义适合具体需要的消息接收重试策略。在上面的配置配置中，定义了一个 `NetMsmqBinding`，并对上述的三个属性进行了相应的设置。

```
<configuration>
  <system.serviceModel>
    ...
    <bindings>
      <netMsmqBinding>
        <binding name="myBinding"
          maxRetryCycles="2"
          receiveRetryCount="2"
          retryCycleDelay="00:00:05"/>
      </netMsmqBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

就拿上面给出的这段配置来说，每个批次重试的次数和最大允许重试的批次均为 2。如果某个操作在执行的过程中总是抛出异常，那么该操作将会被执行 9 次。我们可以通过一个简单的实例来证明这个结论。

还是采用上面定义的 `GreetingService`，现在对它的 `SayHello` 方法进行如下的改写。我们在方法中人为地抛出一个异常，并在之前打印的文字中输出当前的时间。服务端使用的 `NetMsmqBinding` 采用上面的配置。运行程序，在服务端就会出现如下所示的输出结果。从中可以看出 `SayHello` 操作总共执行了 3 批共 9 次，每个批次之间的间隔为 5 秒。演示这个实例的目的在于强调 `NetMsmqBinding` 的 `MaxRetryCycles` 和 `ReceiveRetryCount` 属性设定的是“重试”的次数和批次。(S105)

```
[ServiceBehavior(TransactionAutoCompleteOnSessionClose = true)]
public class GreetingService : IGreeting
{
    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void SayHello(string name)
    {
        Console.WriteLine("Hello, {0} [at {1}]", name,
            DateTime.Now.ToString("hh:mm:ss"));
        throw new Exception();
    }
    //其他成员
}
```

输出结果:

```
Hello, Foo [at 05:24:03]
Hello, Foo [at 05:24:03]
Hello, Foo [at 05:24:03]
Hello, Foo [at 05:24:08]
Hello, Foo [at 05:24:08]
Hello, Foo [at 05:24:08]
Hello, Foo [at 05:24:13]
Hello, Foo [at 05:24:13]
Hello, Foo [at 05:24:13]
```

我们将终结点地址指向的队列称为终结点队列，它具有一个名称为 `retry` 的子队列。对于每个批次的接收重试，WCF 会在第 1 次接收失败之后进行 `ReceiveRetryCount` 次重试，然

后将消息从终结点队列移到 `retry` 子队列。等待 `RetryCycleDelay` 属性表示的一段时间之后, 再将消息从 `retry` 子队列中移到终结点队列。一旦消息重新回到终结点队列, 服务操作正常进行 (此次不算接收重试), 如果失败, 则按照相应的方式进行重试。

由于分批消息接收重试机制依赖于 `retry` 子队列, 而子队列是从 Windows Vista 才开始被引入 MSMQ 的, 所以分批次重试机制在 Windows XP 和 Windows 2003 中是不被支持的。换句话说, 如果操作系统为 Windows XP 或者 Windows 2003, `NetMsmqBinding` 的 `MaxRetryCycles` 会被忽略。

一个 MSMQ 消息本身具有两个所谓的计数器, 即当前中止次数 (`Current Abort Count`) 和当前移动次数 (`Current Move Count`)。前者代表消息参与的事务中止次数, 后者则代表消息在不同的队列之间移动的次数。WCF 服务端将 MSMQ 消息从队列中获取出来生成 WCF 消息的时候, 会将这两个计数获取出来作为消息的属性 (`MessageProperty`)。这个消息属性的类型为具有如下定义的 `System.ServiceModel.Channels.MsmqMessageProperty` 类。`MsmqMessageProperty` 的两个属性 (`AbortCount` 和 `MoveCount`) 分别代表上述两个计数器, 而常量 `Name` 则代表该消息属性在消息中的名称。

```
public sealed class MsmqMessageProperty
{
    //其他成员
    public const string Name = "MsmqMessageProperty";
    public int AbortCount { get; internal set; }
    public int MoveCount { get; internal set; }
}
```

对于每个接收重试批次来说, WCF 通过 `AbortCount` 判断是否达到了允许的重试次数上限 (`ReceiveRetryCount`)。而对于重试批次上限的判断, 则通过 `MoveCount` 来决定。为了深刻地理解这两个计数器在消息接收重试过程中的变化, 再来演示一个实例 (S606)。

为了更加容易地从当前入栈消息中获取以 `MsmqMessageProperty` 对象表示的消息属性, 为 `OperationContext` 编写了如下一个 `GetMsmqMessageProperty` 扩展方法。

```
public static class Extensions
{
    public static MsmqMessageProperty GetMsmqMessageProperty(
        this OperationContext context)
    {
        if (context.IncomingMessageProperties.ContainsKey(MsmqMessageProperty.
            Name))
        {
            return context.IncomingMessageProperties[MsmqMessageProperty.Name]
                as MsmqMessageProperty;
        }
        return null;
    }
}
```

同样使用上面 `GreetingService` 的例子, 现在我们将服务的 `SayHello` 操作方法进行如下的改写, 即在抛出异常之前将从当前入栈消息中取出的 `MsmqMessageProperty` 对象的

AbortCount 和 MoveCount 数值打印出来。

```
[ServiceBehavior(TransactionAutoCompleteOnSessionClose = true)]
public class GreetingService : IGreeting
{
    [OperationBehavior(TransactionScopeRequired = true,
        TransactionAutoComplete = false)]
    public void SayHello(string name)
    {
        MsmqMessageProperty msmqMessageProperty =
            OperationContext.Current.GetMsmqMessageProperty();
        Console.WriteLine("{0,-11}: {1}", "AbortCount",
            msmqMessageProperty.AbortCount);
        Console.WriteLine("{0,-11}: {1}", "MoveCount",
            msmqMessageProperty.MoveCount);
        throw new Exception();
    }
    //其他成员
}
```

采用的 NetMsmqBinding 依然采用上面的配置 (MaxRetryCycles= 2, RreceiveRetryCount=2, RetryCycleDelay=00:00:05)。运行程序, 在服务端将会得到如下所示的输出结果(为了更加直观, 我添加了空行将输出结果按照批次分组)。

```
AbortCount : 0
MoveCount  : 0
AbortCount : 1
MoveCount  : 0
AbortCount : 2
MoveCount  : 0

AbortCount : 0
MoveCount  : 2
AbortCount : 1
MoveCount  : 2
AbortCount : 2
MoveCount  : 2

AbortCount : 0
MoveCount  : 4
AbortCount : 1
MoveCount  : 4
AbortCount : 2
MoveCount  : 4
```

可以看到, AbortCount 是在重试批次内累积的, 而 MoveCount 则是针对重试批次的。由于在开始一个新的批次之前, 消息实际上经历两次移动(从终结点主队列到 retry 子队列, 然后再回到终结点主队列), 所以对于 MoveCount 来说, 后一个批次的值会比前一个大 2。而 WCF 在判断是否达到重试批次上限时会将 MoveCount 的值除以 2, 再和 MaxRetryCycles 进行比较。

对于 Windows Vista 之后的版本来说, AbortCount 和 MoveCount 是 MSMQ 消息的属性。但是对于 Windows XP 和 Windows 2003 来说, 由于不支持按批次接收重试, 因此不会维护

MoveCount。至于 AbortCount，也不是持久地存储于 MSMQ 消息中，而是维护在内存中。所以服务寄宿程序重启之后，AbortCount 的值会被重置。此外，WCF 只能维护不超过 256 个消息的 AbortCount 的值，如果超出这个数量，最老的那个消息的 AbortCount 也会被重置。

6.5.2 接收错误处理

对于由某些因素导致的问题（比如某个时刻负载过高），上述的接收重试机制可确保消息最终能够被成功处理。但是不可否认的是，即使按照配置的重试策略完成了所有批次的重试，有些消息依然得不到处理。那么对于这些未处理的消息应该如何处理呢？

我们将上述的消息称为毒性消息 (Poison Message)，针对毒性消息的策略决定于采用的 NetMsmqBinding。如下面的代码所示，MsmqBindingBase 具有一个可读写的属性 ReceiveErrorHandling，其类型为 System.ServiceModel.ReceiveErrorHandling 枚举。

```
public abstract class MsmqBindingBase : Binding, IBindingRuntimePreferences
{
    //其他成员
    public ReceiveErrorHandling ReceiveErrorHandling { get; set; }
}
public enum ReceiveErrorHandling
{
    Fault,
    Drop,
    Reject,
    Move
}
```

定义在 ReceiveErrorHandling 中的 4 个枚举值表示 4 种针对毒性消息的处理策略。

- **Fault:** 毒性消息再次回到终结点队列，但是该消息不会被继续处理，消息的发送者也不会得到任何确认。Fault 是默认采用的策略。
- **Drop:** 毒性消息会被直接丢弃，并且回复给发送者一个 ACK 确认，表明服务对消息成功进行了处理（其实没有）。
- **Reject:** 和 Drop 选项一样，毒性消息被直接丢弃。但是回复给发送者一个 NACK 确认，表明服务针对消息的处理失败。NACK 确认从 Windows Vista 开始才被用于 MSMQ，所以 Reject 接收错误处理策略不能用于 Windows XP 和 Windows 2003。
- **Move:** 将毒性消息转移到终结点队列的毒性子队列中，但不会向发送者发送任何确认消息。由于自 Windows Vista 之后才支持子队列，因此该选项不能在 Windows XP 和 Windows 2003 中使用。

可以通过编程或者配置的方式来设置 NetMsmqBinding 的 ReceiveErrorHandling 属性。在下面的配置中，我将定义的 NetMsmqBinding 的 ReceiveErrorHandling 属性设置为 Drop。


```

<configuration>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding receiveErrorHandling="Drop" ...>
          ...
        </binding>
      </netMsmqBinding>
    </bindings>
    ...
  </system.serviceModel>
</configuration>

```

6.5.3 死信消息处理

前面介绍了服务端针对毒性消息的处理，现在来看客户端针对“死信”的处理。所谓死信，就是不能正常投递的信件。对于队列服务来说，死信是针对客户端来讲的，不仅表示不能正常抵达目标队列的消息，还包括不能被服务正常处理的消息。

那么对于客户端来说，如何判断发送的消息是否被服务正常处理了呢？很多人会首先想到 MSMQ 的确认机制。当消息被服务成功接收之后，会向消息的发送端发送一个 ACK 确认。而在接收失败的情况下，又具有 4 种不同的策略。如果选择了 Drop 和 Reject，实际上会向发送端分别发送一个 ACK 和 NACK 确认。

对于 NACK 确认消息对应的消息，客户端无疑会将其视为死信。如果某个发送消息一直不曾接收到任何确认，那么是否会一直等待下去呢？通过上面的介绍，我们知道如果服务端选择了 Fault 和 Move 作为接收错误策略，是不会有确认消息发出的，所以无谓的等待是毫无意义的。

对于这种不具有确认的“未决 (In Doubt)”状态的消息，MSMQ 通过超时机制来决定是否应该视其为死信。可以设置一个时间间隔表示消息必须被处理的时间范围（从消息发送时间算起）。并且在消息发送的时候附加一个时间戳。如果在规定的时间内没有接收到任何确认，则视其为死信消息。

对于队列服务来说，这个超时时限是可以通过 NetMsmqBinding 进行动态设置的。如下面的代码所示，MsmqBindingBase 具有 TimeSpan 类型的属性 TimeToLive 就代表这个超时时限。该属性值的默认值为 1 天（24 小时）。

```

public abstract class MsmqBindingBase : Binding, IBindingRuntimePreferences
{
    // 其他成员
    [DefaultValue(typeof(TimeSpan), "1.00:00:00")]
    public TimeSpan TimeToLive { get; set; }
}

```

那么对于因超时或者接收到 NACK 确认而被认定为死信的消息应该做何处理呢？一般来说，我们会选择将其移到死信队列中，可以采用系统死信队列，也可以使用我们自行创建

的一个普通队列作为死信队列。具体的死信处理策略通过 `MsmqBindingBase` 的 `DeadLetterQueue` 属性来决定。

如下面的代码所示, `DeadLetterQueue` 属性是一个类型为 `System.ServiceModel.DeadLetterQueue` 的枚举。枚举值 `System` 和 `Custom` 分别代表采用系统死信队列和自定义死信队列。如果选择了 `Custom`, 那么必须要将自定义死信队列的 `net.msmq` 地址设置在 `CustomDeadLetterQueue` 上。由于作为死信队列的只能是本地队列, 所以地址中表示主机名称的部分必须是本机名称或者 `LocalHost` (连 “.” 都不可以)。在默认的情况下, `DeadLetterQueue` 的属性为 `None`, 意味着不对死信消息做任何处理。

```
public abstract class MsmqBindingBase : Binding, IBindingRuntimePreferences
{
    //其他成员
    [DefaultValue(null)]
    public Uri CustomDeadLetterQueue { get; set; }
    [DefaultValue(1)]
    public DeadLetterQueue DeadLetterQueue { get; set; }
}
public enum DeadLetterQueue
{
    None,
    System,
    Custom
}
```

`NetMsmqBinding` 的三个属性(`TimeToLive`、`DeadLetterQueue` 和 `CustomDeadLetterQueue`)都可以通过配置的方式来设置。在下面的配置中, 我们为 `NetMsmqBinding` 设置了一个自定义的死信队列, 并将 `TimeToLive` 设置为 5 个小时。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding timeToLive="05:00:00"
                  deadLetterQueue="Custom"
                  customDeadLetterQueue=
                    "net.msmq://localhost/private/dlq4demo"...>
          <security mode="None"/>
        </binding>
      </netMsmqBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

对于存储于死信队列(系统死信队列和自定义死信队列)中的死信消息, 我们一般会创建相应的服务来进行相应的处理。和处理毒性消息服务一样, 死信消息服务和业务服务实现相同的契约。对于死信服务操作处理的消息, 同样具有一个类型为 `MsmqMessageProperty` 的消息属性。除了上面介绍的 `AbortCount` 和 `MoveCount` 两个计数器之外, `MsmqMessageProperty` 还具有额外两个只读属性, 即 `DeliveryFailure` 和 `DeliveryStatus`, 分别表示投递失败的类型和状态。这两个属性最初都来源于获取的 MSMQ 消息。

```

public sealed class MsmqMessageProperty
{
    //其他成员
    public DeliveryFailure? DeliveryFailure { get; }
    public DeliveryStatus? DeliveryStatus { get; }
}
public enum DeliveryFailure
{
    AccessDenied = 0x8004,
    BadDestinationQueue = 0x8000,
    BadEncryption = 0x8007,
    BadSignature = 0x8006,
    CouldNotEncrypt = 0x8008,
    HopCountExceeded = 0x8005,
    NotTransactionalMessage = 0x800a,
    NotTransactionalQueue = 0x8009,
    Purged = 0x8001,
    QueueDeleted = 0xc000,
    QueueExceedMaximumSize = 0x8003,
    QueuePurged = 0xc001,
    ReachQueueTimeout = 0x8002,
    ReceiveTimeout = 0xc002,
    Unknown = 0
}
public enum DeliveryStatus
{
    InDoubt,
    NotDelivered
}

```

如上面的代码所示，`DeliveryFailure` 和 `DeliveryStatus` 都是枚举类型。枚举类型 `System.ServiceModel.Channels.DeliveryFailure` 定义了一系列导致投递失败的场景，而枚举类型 `System.ServiceModel.Channels.DeliveryStatus` 则定义了两种具体的投递状态。对于接收到 NACK 确认的死信消息，状态为 `NotDelivered`，而对于因超时而成为死信的消息，状态则为 `InDoubt`。

6.5.4 日志（Journaling）与跟踪（Tracing）

MSMQ 本身提供了日志功能，并且支持源日志（Source Journaling）和目标日志（Target Journaling）。对于队列服务的客户端也可以通过针对 `NetMsmqBinding` 的相关设置提供源日志的支持。如下面的代码片段所示，`MsmqBindingBase` 具有一个布尔类型的 `UseSourceJournal` 属性，代表是否启动源日志。该属性在默认的情况下值为 `False`。

```

public abstract class MsmqBindingBase : Binding, IBindingRuntimePreferences
{
    //其他成员
    [DefaultValue(false)]
    public bool UseSourceJournal { get; set; }
}

```

当然，`UseSourceJournal` 属性的设置同样体现在 `NetMsmqBinding` 的配置中。如下面的

XML 片段所示, NetMsmqBinding 的配置节点具有一个 useSourceJournal 配置属性供我们开启和关闭日志功能。当客户端终结点的 NetMsmqBinding 的 UseSourceJournal 被设置为 True 后, 被发送的消息副本将会自动存储于本机的系统日志队列。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding useSourceJournal="true" ...>
          ...
        </binding>
      </netMsmqBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

除了 UseSourceJournal 属性, MsmqBindingBase 还有另外一个布尔类型的属性 UseMsmqTracing。如下面的代码所示, 该属性的默认值为 False, 用于开启或者关闭消息的路由追踪 (Route Tracking) 功能。如果通过 UseMsmqTracing 开始了本机 MSMQ 的路由追踪, MSMQ 消息在离开或者抵达某台主机的时候都会向客户端主机的报表队列发送包含路由信息的报表消息。

```
public abstract class MsmqBindingBase : Binding, IBindingRuntimePreferences
{
    [DefaultValue(false)]
    public bool UseMsmqTracing { get; set; }
}
```

第7章 传输安全

(Transfer Security)

WCF 的传输安全主要涉及认证、消息一致性和机密性三个主题。认证不仅包括服务端对客户端的身份认证，也包括客户端对服务端的身份认证，即双向认证 (Mutual Authentication)。消息的一致性确保整个消息或者消息的某个部分在传输过程中内容不被篡改。而机密性则确保只有所希望的消息接收方才能读取其中的内容。



对于任何一个企业级应用来说,安全 (Security) 都是一个不可避免的话题。如何识别用户的身份?如何将用户可执行的操作和可访问的资源限制在其允许的权限范围之内?如何记录用户行为,让相应的操作都有据可查?这些都是应用的安全机制或安全框架需要考虑的典型问题,它们分别对应着三个安全行为,即认证 (Authentication)、授权 (Authorization) 和审核 (Auditing)。

除了这些典型的安全问题,对于一个以消息作为通信手段的分布式应用,还需要考虑消息保护 (Message Protection) 问题。而消息保护机制主要包括签名 (Signature) 和加密 (Encryption)。签名确保消息的一致性 (Message Integrity),即保证消息在最初发送者和最终接收者之间没有被第三方篡改。而加密确保消息的机密性 (Message Confidentiality),即保证消息的内容仅对发送者期望的接收者可见。

7.1 传输安全简介

WCF 是一个以消息作为通信手段的分布式平台,这使得我们可以将某些可复用的功能以服务的方式进行定义,并最终部署于分布式网络环境中的某个节点,供潜在的服务消费者调用。服务和调用服务的客户端可以同时存在一个相同的网络,也可以跨越不同的网络,甚至可以通过 Internet 进行互联的两台机器。网络的不确定性为分布式应用带来了一系列安全隐患,在正式介绍 WCF 的传输安全之前,先来介绍一下这些安全隐患。

7.1.1 分布式应用中的传输安全隐患

可以将 WCF 看成一个消息处理框架,整个框架大体分成两个部分,客户端和服务端。客户端负责请求消息的发送和回复消息的接收,而服务端则负责请求消息的接收和回复消息的发送。WCF 只能控制对消息发送前和接收后的处理,对发送后到接收前这一段消息传输过程是“失控的”。而正是消息传输的网络不能提供足够的安全保障,会带来如下一些典型的安全隐患。

- 消息的篡改:传输中的消息被某些网络拦截工具捕获,并被恶意篡改后转发给消息接收者。消息接收者如果直接对消息进行处理,就会做出错误操作或者返回给客户端错误的结果。
- 敏感信息的泄露:包含一些敏感信息(比如信用卡号、身份证号等)的消息如果以明文的方式在网络中传输,被网络拦截工具捕获后会被恶意的黑客看到。
- 钓鱼攻击 (Phishing Attack):针对服务 A 的请求被恶意重新定位到另一个服务 B,以执行一些损害访问者利益的操作,或者窃取访问者相关的一些敏感信息。
- 重放攻击 (Replay Attack):黑客利用网络拦截工具捕获针对某个服务的访问请求,然

后对该请求进行复制，并以一个非常高的频率对目标服务发起调用。这样将会耗尽服务端的可用资源并导致崩溃。

上面给出了分布式网络应用中由于网络环境的不确定性导致的几个比较典型的安全隐患。实际上由于网络协议本身并不能提供足够的安全保障，我们还会遇到其他很多网络安全问题，而且网络安全问题在 Internet 环境下尤为突出。为了弥补网络协议本身对安全保障的局限，我们不得不在应用级别重建安全体系。

但是安全是一个相对“高级和复杂”的话题，构建一个适合具体应用要求的安全体系对应用的开发和架构人员具有较高的要求。对于一般的中小规模的分布式应用，投入到安全架构方面的成本完全有可能超过实现业务应用模块的总和。而且花大力气构建的安全体系可能并不会如想象的那么安全。所以将安全的实现完全下放到具体的应用也是不太现实的。

既然在网络通信层面不能提供足够的安全保障，而在应用层面去实现安全保障也不太现实，那么我们只能将整个安全保障体系构建于两者之间。我们可以将解决方案姑且称为平台级别或者框架级别的安全保障体系。

作为分布式开发平台的 WCF 实现了一个功能齐全、可扩展的安全架构体系，能够满足绝大部分分布式应用场景的安全需求。作为建立在 WCF 上的分布式应用的架构人员，只需要根据自身的场景进行相应的定制即可。

WCF 提供了两种不同的安全模式，即 Transport 安全和 Message 安全。在正式介绍这两种不同的安全模式之前，我们简单地补充一些必要的关于非对称加密的知识。

7.1.2 非对称加密 (Asymmetric Cryptography)

数字签名和加密依赖于相应的加密算法 (Cryptographic Algorithm)。从数学的角度来讲，加密算法就是将被加密的数据和密钥作为自变量，将加密后的数据作为因变量的函数。按照加密和解密采用的密钥是否相同，我们将加密算法分为对称加密算法和非对称加密算法。前者采用相同的密钥进行加密和解密，后者则采用一组相互配对的密钥分别进行加密和解密。对于非对称加密，我们选择密钥对中某一个密钥对消息进行加密，该密文只有通过另一个密钥方能解密。

对于基于消息的通信来说，非对称加密具有两个典型的应用场景。一是直接通过对消息进行加密解决机密性问题；二是通过数字签名实现身份认证和数据一致性的问题。

1. 消息加密 (Encryption)

非对称加密依赖于一组由公钥/私钥 (Public Key /Private Key) 组成的密钥对，所以非对

称加密又被称为公钥加密 (Public Key Cryptography)。如果密钥对中的其中一个用于加密, 另一个则用于解密。公钥公诸于众, 不具有隐私性, 任何人都可以获取。而私钥专属于拥有该密钥对的实体, 属于绝对隐私。

对于消息交换来说, 通过非对称的方式对消息进行加密能够确保消息的机密性。消息的发送方采用接收方的公钥进行加密, 接收方通过自己的私钥进行解密。私钥仅供接收方所有, 所有其他人不能对密文进行解密。

2. 数字签名 (Digital Signature)

数字签名实际上包括两项主要工作, 签名 (Signing) 和检验 (Verification)。前者创建一个数字签名, 后者验证签名的有效性。签名的过程其实很简单, 整个流程如图 7-1 所示。整个流程包括如下两个步骤。

- 发送方采用某种算法对整个消息的内容实施哈希计算, 得到一个哈希码。
- 发送方使用自己的私钥对该哈希码进行加密, 加密后得到的密文就是数字签名。

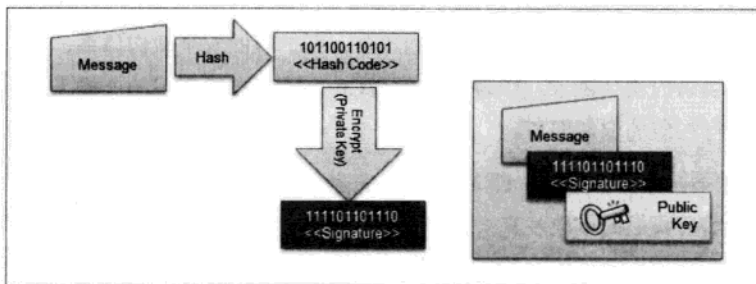


图 7-1 数字签名流程

该数字签名最终会连同发送方密钥对中的公钥 (该公钥一般会内嵌于一个数字证书中) 附加到原消息上一并发给接收方。

这三项被接收方接收之后, 就可以借助这个数字签名验证发送方的真实身份和消息的完整性, 这个过程被称为数字签名的检验。整个数字签名检验流程如图 7-2 所示。

- 源消息被提取出来, 通过相同的哈希算法得到一个哈希码。
- 数字签名被提取出来, 采用相同的算法利用公钥对数字签名进行解密, 得到生成数字签名的那个哈希码。

两个哈希码进行比较, 如果一致, 则可以证明数字签名的有效性及消息本身的完整性。

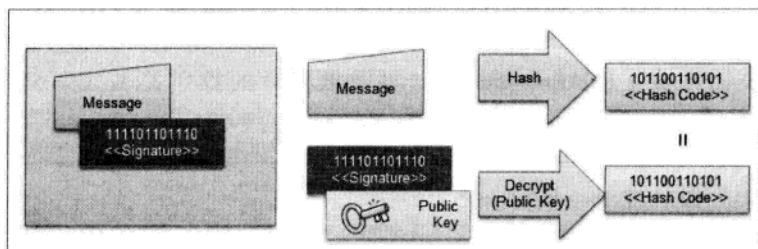


图 7-2 数字签名检验流程

采用非对称密码学对消息进行加密解决的是消息的机密性问题，而数字签名的作用则体现在如下三个方面。

- **身份认证 (Authentication):** 数字证书可以帮助我们验证消息发送源的真实身份。因为数字签名的内容是由一个私钥决定的，发送方只有通过专属于他本人密钥对中的私钥生成数字签名，才能通过对方利用公钥实施的数字签名检验。对数字证书的检验实际上就是确认消息的发送源是否是私钥的真正拥有者。
- **防止抵赖 (Non-repudiation):** 防止抵赖在这里代表——对于接收到的经过数字签名的消息，如果接收方采用某个实体的公钥对数字签名检验成功，那么这个实体就是消息的发送方，不容对方抵赖。原因很简单，能够通过公钥对某个数字签名成功检验，证明生成该数字签名使用的是正确的私钥。
- **消息一致性 (Integrity):** 数字签名确实可以确保整个消息内容的一致性，因为最初用于私钥加密的哈希码是针对整个消息的内容进行哈希计算获得的。消息的内容一旦出现任何改变，最终对数字签名的检验都将失败。

公钥在一般情况下是通过数字证书的形式进行传递的，数字证书在这里作为发送方的凭证。下面就简单介绍一下数字证书。

3. 数字证书 (Digital Certificate)

证书，又称数字证书 (Digital Certificate) 或者公钥证书 (Public Key Certificate)，是一种数字签名的声明。它将公钥值绑定到持有对应私钥的个人、设备或服务的标识信息上。由于大多数证书基于 X.509 V3 证书标准，因此我们又将其称为 X.509 证书。X.509 证书广泛地应用于加密和数字签名，以提供认证的实现和确保数据的一致性和机密性。

站在非对称加密的角度来讲，X.509 证书就是一个将某个密钥对中的公钥与某个主题 (Subject) 进行绑定的文件。和公钥进行绑定的不仅可以包括相应主题的可辨别名称 (DN, Distinguished Name)，也可以包括主题相关的其他名称，比如 E-mail 地址、DNS 名称等。

下面是一个 X.509 证书的大体结构，包括版本号 (V3)、序列号 (7829)、签名算法 (md5WithRSAEncryption)、颁发者 (CN=Root Agency)、有效日期 (April 07, 2011 3:37:45 PM

到 January 01, 2040 7:59:59 AM)、主题信息 (CN = www.artech.com)、公钥 (00:b4:31:98:...52:7e:41:8f) 和公钥算法 (RSAEncryption), 以及颁发者的数字签名 (93:5f:8f:5f:...b5:22:68:9f)。

Certificate:

Data:

```

Version: V3
Serial Number: 7829 (0x1e95)
Signature Algorithm: md5WithRSAEncryption
Issuer: CN=Root Agency
Validity
    Not Before: Thursday, April 07, 2011 3:37:45 PM
    Not After : Sunday, January 01, 2040 7:59:59 AM
Subject: CN = www.artech.com
Subject Public Key Info:
    Public Key Algorithm: rsaEncryption
    RSA Public Key: (1024 bit)
        Modulus (1024 bit):
            00:b4:31:98:...52:7e:41:8f.
        Exponent: 65537 (0x10001)
Signature Algorithm: md5WithRSAEncryption
93:5f:8f:5f:...b5:22:68:9f

```

(1) 数字证书的颁发机制

数字证书在大部分场景中作为证明某个实体身份的凭证来使用, 而证书的主题部分的内容代表了该用户凭证所代表的身份。那么我们的第一个问题是: 为什么要信任这个证书?

我们可以将其与日常生活中使用的证书进行类比。比如居民身份证就是一种典型的证书, 它的一个重要的特征就是该证书是由官方认可的合法机构 (户口所在地的公安机关) 颁发的。对于数字证书, 尤其是用于商业用途的数字证书, 也具有相应的官方颁发机构, 我们将这样的机构称为认证权威机构 (CA, Certification Authority)。我们熟悉的 CA 包括 VeriSign、Thawte (OpenSSL) 等。证书的颁发机构体系是一个树形结构, 每一个 CA 可以具有一到多个子 CA, 最上层的 CA 被称为根 CA。

人们对居民身份证的普遍认可来源于对颁发机构的信任, 这同样适用于数字证书。认证方可以通过检验数字证书的 CA 的信任程度而做出对证书合法性的判断。不过现在的问题是: 居民身份证具有若干防伪标识帮助认证方鉴别真伪, 对于数字证书, 我们采用怎样的方式来判断它是不是伪造的呢? 要验证数字证书的有效性, 需要防止以下两种情况:

- 用户伪造一个证书以假冒与证书公钥绑定的那个身份, 并且该证书具有一个我们普遍认可的 CA;
- 用户对 CA 颁发的证书进行篡改, 改变公钥或者其他身份信息。

这两个问题都可以通过数字签名技术来解决。证书中不仅包括 CA 的基本信息, 还包括一个数字签名和签名采用的算法。CA 通过自己的私钥对证书的数据部分进行签名, 并将此签名连同签名采用的算法置于证书之中。按照前面介绍的关于数字签名的原理, 如果我们具有 CA 的公钥, 不仅可以验证证书的 CA, 也能校验证书的内容是否被篡改。那么在对证书

进行验证的时候，CA 的公钥从何而来呢？

CA 的公钥也保存在一个数字证书之中，并被存储于一个受信任的证书存储之中。按照证书代表身份的不同，我们可以将其分为两种类型：CA 证书（CA Certificate）和终端实体证书（End Entity Certificate），其中前者代表 CA，后者代表接受 CA 颁发证书的最终实体。实际上 CA 证书和终端实体证书并没有本质的区别。除了顶层的根 CA，所有的 CA 证书颁发者是它的上一级 CA，即上级的 CA 作为该 CA 证书的 CA。CA 的这种层级关系组成了一种信任链（Trust Chain）。

为了存储数字证书，Windows 中具有相应的证书存储区（Certificate Store）。根据目的或者信任范围的不同，不同的证书被存储于不同的存储区。关于证书存储，由于篇幅所限，这里不做过多的介绍，有兴趣的朋友可以查阅 MSDN 在线文档。对于证书存储管理，MMC 提供了一个可视化的管理工具，也可以通过 Certmgr.exe 工具以命令行的方式进行。

在若干证书存储区中，有一个被称为“受信任的根证书颁发机构”（Trusted Root Certification Authorities）的存储区，它里面存储的所有 CA 证书代表所信任的证书颁发机构。在默认情况下，对于一个待验证的证书，如果基于该证书 CA 信任链上的任何一个 CA 在该存储区中存在一个证书，那么这个证书是合法的。

（2）创建数字证书

用户对数字证书的认可取决于对证书颁发机构的信任，所以证书颁发机构决定了数字证书的可用范围。由官方认可的数字证书颁发机构，比如 VeriSign、Thawte（OpenSSL），具有普遍的信任度，在大部分情况下是理想的选择。但是对于学习研究或者开发测试，没有必要去购买这些商用证书，可以利用一些工具以手工的方式创建证书。

WCF 的安全机制广泛用到数字证书，因此很有必要学会手工创建数字证书。微软提供了一个创建数字证书的工具 MakeCert.exe，可以借助这个工具采用命令行的方式创建我们需要的数字证书。MakeCert.exe 具有很多命令行开关，在这里仅对几个常用的开关做一下简单的介绍。

- **-n x509name**：指定证书的主题名称。最简单的方法是在双引号中指定此名称，并加上前缀 CN=。例如，"CN=My Name"（CN 是 Common Name 的简写）。
- **-pe**：将所生成的私钥标记为可导出，这样可将私钥包括在证书中。
- **-sr location**：数字证书的存储位置，具有 CurrentUser 和 LocalMachine 两个可选值。前者基于当前登录用户，后者基于本机。
- **-ss store**：数字证书的存储区。
- **-sky keytype**：指定密钥类型，必须是 signature、exchange 或一个表示提供程序类型的整数（1 表示交换密钥，2 表示签名密钥）。

比如通过下面的命令会创建一个主题名称为 www.artech.com 的数字证书，密钥类型为

交换密钥, 并且包含私钥。

```
MakeCert -n "CN=www.artech.com" -pe -sr LocalMachine -ss My -sky exchange
```

一旦上面的命令成功执行, 生成的证书会自动保存到基于本机的个人 (Personal) 存储区中。可以通过 MMC 的证书管理单元 (Snap-in) 查看该证书。图 7-3 是证书 MMC 管理单元的截图, 可以看到我们创建的数字证书已经被存储到了在命令行中指定的存储区, 颁发机构被默认设定为 Root Agency。

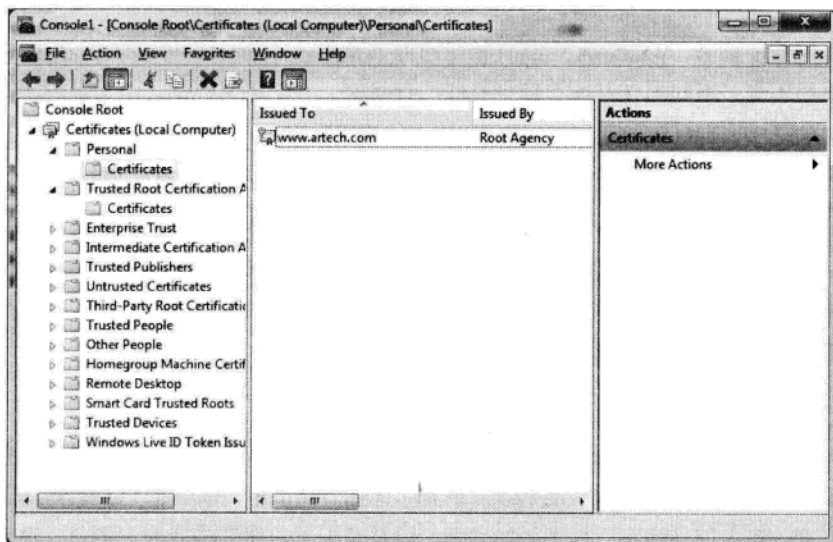


图 7-3 通过证书 MMC 证书管理单元查看创建的证书

7.1.3 Transport 与 Message 安全模式

WCF 的传输安全主要涉及认证、消息一致性和机密性三个主题。认证不仅包括服务端对客户端的身份认证, 也包括客户端对服务的身份认证, 即双向认证 (Mutual Authentication)。消息的一致性确保整个消息或者消息的某个部分在传输过程中内容不被篡改。而机密性则确保只有希望的消息接收方才能读取其中的内容。

注: 由于英文中的 Transfer Security 和 Transport Security 都可以翻译成传输安全, 为示区别, 我用中文的“传输安全”表示涉及认证、消息一致性和机密性的 Transfer Security, 而采用“Transport 安全 (模式)”表示基于某种网络协议的传输层安全 (模式), 与“Transport 安全 (模式)”相对的是“Message 安全 (模式)”。

WCF 采用两种不同的机制来解决这三个涉及传输安全的问题, 一般将它们称为不同的安全模式, 即 Transport 安全模式和 Message 安全模式。

1. Transport 安全模式

Transport 安全模式利用基于传输层协议的安全机制解决传输安全涉及的三个问题（认证、消息一致性和进行性）。而 TLS/SSL 是实现 Transport 安全最常用的方式（不是唯一的方式）。

（1）TLS、SSL 和 HTTPS

任何一本介绍 WCF 的书，在介绍 Transport 安全模式的时候，必然会提到 SSL 或 HTTPS，有时还会提到 TLS。有些人不太明白这三者到底有什么区别，尤其是不能很好地区分 TLS 和 SSL 的差别。在这里我们就先来简单介绍一下这三个相关的概念。

SSL(Secure Sockets Layer)最初是由 Netscape 公司开发的一种安全协议，应用于 Netscape 浏览器以解决与 Web 服务器之间的安全传输问题。SSL 先后经历了三个主要的版本（1.0、2.0 和 3.0）。之后 SSL 被 IETF（Internet Engineering Task Force）接管，正式更名为 TLS（Transport Layer Security）。可以这么说，SSL 是 TLS 的前身，TLS 1.0 就是 SSL 3.1。

TLS/SSL 本身和具体的网络传输协议无关，既可以用于 HTTP，也可以用于 TCP。而 HTTPS（Hypertext Transfer Protocol Secure）则将 HTTP 和 TLS/SSL 两者结合起来。在一般情况下，HTTPS 通常采用 443 端口进行通信。对于 WCF 来说，所有基于 HTTP 协议的绑定采用的 Transport 安全都是通过 HTTPS 来实现的。而 NetTcpBinding 也提供了对 TLS/SSL 的支持，一般将 TLS/SSL 在 TCP 上的应用称为 SSL Over TCP。TLS/SSL 帮助我们解决如下两个问题：

- 客户端对服务端的验证；
- 通过对传输层传输的数据段（Segment）进行加密确保消息的机密性。

这里采用的是对称加密而不是非对称加密。接下来从消息交换的角度来说明上述的两个问题是如何通过 TLS/SSL 解决的。

以访问一个 HTTPS 站点为例。当客户端和这个 HTTPS 站点所在的 Web 服务器进行正式的访问请求之前，在它们之间必须建立了安全的 HTTP 连接。而这样一个安全的连接的创建通过客户端和 Web 站点之间的多次握手或协商（Negotiation）来完成。如图 7-4 所示，整个协商过程主要包括三个步骤。

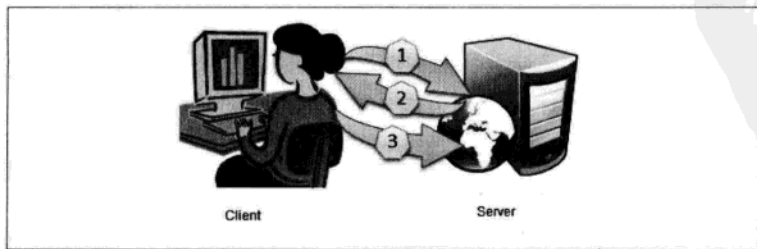


图 7-4 TLS/SSL 安全连接创建过程

- 步骤一：客户端向 HTTPS 站点发送协商请求，该请求中包括客户端所能够支持的加密算法列表。
- 步骤二：HTTPS 站点从加密算法列表中选择自己支持的并且安全级别最高的算法（有时候站点也可能综合考虑性能和安全两者之间的平衡，从中选择一个“最佳”的加密算法），连同绑定到该站点的数字证书（所有 HTTPS 站点在部署的时候都会绑定一个 X.509 证书）一并发送给客户端。
- 步骤三：客户端接收到站点发回的数字证书之后，通过验证证书进而确定站点身份。在验证成功的情况下，客户端会生成一个随机数，作为会话密钥（Session Key），缓存在客户端。客户端随后采用站点发回的加密算法，利用从证书中提取的公钥进行加密。加密后的会话密钥被发送给站点，而站点使用自己的私钥采用相对应的算法进行解密得到该会话密钥。至此客户端和服务端具有一个只有它们彼此知晓的会话密钥，所有的请求和回复消息均通过该会话密钥进行加密和解密。

有人可能会说，客户端为何不直接用从数字证书提取的公钥对所有的请求消息进行加密，服务端采用私钥进行解密？之所以选择对称加密而不是非对称加密，主要有两方面的原因：

- 对称加密/解密比非对称加密/解密需要更少的计算，所以具有更好的性能；
- 在客户端不提供自身证书的情况下，非对称的加密方式只能确保客户端向服务端请求消息的机密性，而不能保证服务端向客户端回复消息的机密性。

（2）Transport 安全模型的优缺点

与我们后面将介绍的 Message 安全模式相比，Transport 安全模式具有一个最大的优点，那就是高性能。虽然 TLS/SSL 在正式进行消息交换之前需要通过协商建立一个安全的连接，但是这个协商过程完全通过传输层协议来完成，而且可以利用网络适配器的硬件加速。但是受限于传输层安全协议的特点，Transport 安全模式也具有一些不可避免的局限。

- Transport 安全模式依赖于具体的传输协议。
- 它只能提供基于点对点（Point-to-Point）的安全传输保障，即客户端直接连接服务的场景。如果在客户端与服务端之间的网络需要一些用于消息路由的中间节点，Transport 安全模式则没有了用武之地。
- 如果采用 Transport 安全模式，意味着我们不得不在传输层（而不是应用层）解决对客户端的认证。这就决定了可供选择的认证方式（或者说可以采用的凭证）比较少。

也正是由于上述的这些局限（主要还是只能提供点对点的安全传输保障），决定了 Intranet 是 Transport 安全模式主要的应用环境。为了克服这些局限，我们需要一种与传输协议无关的、能够提供端到端（End-to-End）安全传输保障的、并且具有多种认证解决方案的安全模式，那就是 Message 安全模式。

2. Message 安全模式

Message 模式直接将安全策略的目标对象转移到消息本身，通过对消息进行签名、加密实现消息安全传输。所以 Message 安全模式不会因底层是 HTTP 或 TCP 传输协议而采用不同的安全机制。它还能够提供从消息最初发送端到最终接收端之间的安全传输，即端到端（End-To-End）安全传输。Message 模式下的安全协议是一种应用层协议，可以在应用层上实现对客户端的验证，因而具有更多的认证解决方案可供选择。

WCF 的 Message 安全模式并不是微软在 Windows 平台下的闭门造车，而是遵循了一系列开放的标准或规范，那就是围绕着 WS-Security 的 4 个 WS-* 规范（包含 WS-Security），即 WS-Trust、WS-Secure Conversation 和 WS-Security Policy。这就意味着 WCF 的 Message 安全具有很好的互操作性或平台无关性。

（1）WS-Security

WS-Security 是由结构化信息标准促进组织（OASIS, Organization for the Advancement of Structured Information Standards）制定的，有时候又被简称为 WSS。到目前为止，WS-Security 具有两个版本，第一个正式的版本于 2004 年 3 月发布，即 WS-Security 1.0（有时候被称为 WS-Security 2004）。2006 年 2 月，OASIS 发布了 WS-Security 1.1。

定义在 WS-Security 中的 SOAP 的安全机制可以广泛地应用于现有的多种体系，比如 PKI、Kerberos 和 SSL 等。WS-Security 支持多种安全令牌（Security Token）格式（比如用户名/密码、SAML、X509 证书和 Kerberos 票据等），以及多种签名格式和加密技术。WS-Security 提供了关于 SOAP 安全交换的三个主要机制：

- 如何将安全令牌作为消息的一部分进行传输；
- 如何检测接收到的消息是否和原始发送的一致；
- 如何确保消息的真实内容仅对真正的接收者可见。

安全令牌的传输主要解决身份认证的问题，所以这三个方面就是传输安全面对的三个问题：身份认证、消息一致性和消息机密性。WS-Security 提供了一个抽象的消息安全模型。在这个安全模型中，通过安全令牌，结合数字签名和加密技术实现对消息交换实体的认证和对消息本身的保护。

（2）WS-Trust

WS-Trust 定义了一系列 SOAP 扩展，旨在为消息交换相关方之间建立一个信任的关系。站在消息交换的角度来讲，信任关系不仅包括消息接收者对请求者的信任，也包括请求者对接收者的信任。要建立起彼此之间的信任关系，一个前提是能够互相验证对方的真实身份，所以这里也就涉及一个双向验证的问题。

在 Web 服务的世界中，消息交换为通信的唯一手段，那么相关方之间的信任关系的建立也只能围绕着消息交换来实现。定义在 WS-Trust 中的 Web 服务的信任模型基于这样的处

理机制: Web 服务要求接收的消息中包含证明身份所需的声明 (包括身份、权限或能力等)。如果 Web 服务接收到的消息不具有这些证明信息, 它可以选择忽略或者拒绝该消息。

这些证明信息以安全令牌 (Security Token) 的方式存在。实际上 WS-Trust 为我们提供了一种消息交换机制以实现对安全令牌的颁发 (Issuance)、续订 (Renewal) 和终止 (Cancel) 等操作。而完成这些操作则是按照 WS-Security 中的消息交换方式进行的。

WS-Trust 具有两个主要的版本, 即 WS-Trust 1.3 和 WS-Trust 1.4, 它们发布的时间分别是 2005 年和 2008 年。

(3) WS-Secure Conversation

通过上面的介绍, 我们知道安全传输旨在解决两个方面的问题, 即身份认证和消息保护 (消息的一致性和机密性)。我们假设客户端和服务端分别采用用户名/密码和 X.509 证书作为各自的用户凭证, 可以通过下面的方式解决上述两个问题:

- 客户端采用服务端证书的公钥对消息进行加密, 服务端在接收到消息的时候通过自己的私钥进行解密;
- 客户端的每次服务调用均附加一个基于用户名/密码的安全令牌, 服务端提取它用于验证访问者的身份。

这好像是一个“完美”的解决方案, 但是不知道你是否考虑过, 如果客户端和服务端在一段时间内需要进行频繁的通信, 会产生性能问题。影响性能的因素主要来源于两个方面, 即服务端需要对客户端进行频繁的认证和频繁地进行非对称加密/解密。

我们更加希望客户端和服务端在进行正式的消息交换之前, 在它们之间通过彼此的认证, 建立起一个安全的上下文 (或者说一个安全会话)。在这个上下文中, 服务端无须对客户端进行重复的认证。此外一个仅在当前上下文中被双方共享的密钥被创建出来, 采用对称加密技术对消息进行签名和加密。

这个机制基本类似于 TLS/SSL, 不过 TLS/SSL 只是在传输层针对数据段提供安全传输保障, 而我们现在介绍的则是针对 SOAP 消息的安全传输。由于这个机制主要为交互双方在同个上下文环境中的多次消息交换提供安全传输的保障, 因此将其称为 Secure Conversation。OASIS 为此制定了相应的规范, 也就是我们本节介绍的 WS-SecureConversation (简称 WS-SC)。而 WCF 通过 Secure Sessions 机制提供对 WS-SC 的实现。

WS-SC 具有 1.3 和 1.4 两个主要的版本, 分别在 2007 和 2009 年发布。WS-SC 建立在 WS-Security 和 WS-Trust 基础之上, 旨在提供一种机制, 实现对安全上下文的创建和对整个生命周期的控制。

(4) WS-Security Policy

一个 Web 服务除了实现通过服务契约定义的业务功能之外, 为了实现一些额外的功能 (比如安全、事务和可靠传输等), 需要具有一些与业务无关的行为 (Behavior) 和能力

(Capability)，我们可以将这些统称为 Web 服务的策略 (Policy)。WS-Policy 提供了一个基于 XML 的框架模型和语法，用于描述 Web 服务的能力、要求和行为属性。关于 WS-Policy，在本书第 2 章“元数据 (Metadata)”中有相应的介绍。

这里介绍的 WS-SecurityPolicy (简称为 WS-SP) 建立在 WS-Policy 基础上，定义了一系列关于安全传输的策略断言 (Policy Assertion)。这些策略断言最终应用在 WS-Security、WS-Trust 和 WS-SC 中。WS-SP 具有 1.2 和 1.3 两个主要的版本，发布时间分别为 2007 年和 2009 年。

到此为止，我们已经介绍了 WS-* 体系中关于安全的 4 个重要的规范，WS-Security、WS-Trust、WS-SC 和 WS-SP。而 WCF 的消息安全模式是这 4 个 WS-* 规范的实现者。如果想深刻地理解 WCF 的安全体系，对这 4 个安全规范的了解是必需的。

(5) Message 安全模式的优缺点

关于 WCF 的 Message 安全模式的优点，实际上在前面已经有所提及，在这里做一个总结。较之 Transport 安全模式，Message 安全模式具有如下优点。

- 由于 Message 安全模式是在应用层通过对消息实施加密、签名等安全机制实现的，所以这是一种与具体传输协议无关的安全机制，不会因底层采用的是 TCP 或 HTTP 而有所不同。较之 Transport 安全，这种基于应用层实现的安全机制在认证方式上具有更多的选择。
- 由于 Message 安全模式下的各种安全机制都是直接应用在消息 (SOAP) 级别的，因此无论消息路由的路径有多复杂，都能够保证消息的安全传输。不同于 Transport 安全模式只能提供点对点 (Point-to-Point) 的安全，Message 安全模式能够提供端到端 (End-to-End) 的安全。
- 由于 Message 安全模式是对 WS-Security、WS-Trust、WS-SC 和 WS-SP 这 4 个 WS-* 规范的实现，所有具有很好的互操作性，能够提供跨平台的支持。

但是 Transport 安全模式有一点是 Message 安全模式不能比的，那就是性能。

3. 混合安全模式 (Mixed Security Mode)

由于 WCF 的两种安全模式 (Transport 和 Message) 具有各自的优缺点，因此可以通过两者的结合构成一种混合的安全传输解决方案，我们称之为混合 (Mixed) 安全模式。那么这种新的安全模式是如何对 Transport 和 Message 安全模式进行“混合”的呢？

安全传输旨在解决认证、消息一致性和机密性，而认证既包括服务端对客户端的认证，也包括客户端对服务端的认证。对于混合安全模式，消息的一致性、机密性和客户端对服务端的认证通过 Transport 安全模式来实现，而采用 Message 安全模式实现服务端对客户端的认证。

混合 (以下简称 Mixed) 安全模式充分利用了 Transport 安全模式硬件加速优势, 以提供高性能和具有高吞吐量的服务。由于服务端对客户端的验证是通过 Message 安全模式来实现的, 因此我们具有更多关于客户端安全凭证和认证方式的选择。由于 Transport 安全模式不可回避的局限性, 混合安全模式也只能提供点到点的安全。

7.2 认证

对实体进行身份识别和鉴定的行为被称为认证 (Authentication)。认证帮助我们确认“谁在敲打我的门”。应用的访问者以一个它声明的身份叩响第一道门, 看门人只有在成功确定对方身份无误之后方能为其开启方便之门, 否则直接将其扫地出门。

如果要给认证下一个定义, 我个人倾向于这样的定义: 认证是确定被认证方的真实身份是否和他或她声明 (Claim) 的身份相符的行为。认证方需要被认证方提供相应的身份证明材料, 以鉴定本身的身份是否与声称的身份相符。在计算机的语言中, 这里的身份证明有一个专有的名称, 即“凭证 (Credential)”, 或者用户凭证 (User Credential)、认证凭证 (Authentication Credential)。

7.2.1 认证与凭证 (User Credential)

最好的设计就是能够尽可能地模拟现实。对于安全认证来说, 在现实生活中有无数现成的例子。比如我对一个不认识的人说“我是张三”, 对方如何才能相信我是张三而非李四呢? 虽然我们未必全都是拥有身份的人, 但无疑我们都是拥有身份证的人。身份证可以证明我们的真实身份, 这里的身份证就是一种典型的凭证。认证方能够根据被认证方提供的身份证识别对方的真实身份, 必须满足三个条件:

- 被认证人声称是身份证上注明的那个人;
- 身份证的持有者就是身份证的拥有者;
- 身份证本身是合法有效的, 即是通过公安机关颁发的, 而不是通过拨打“办证”电话办理的。

第一个问题一般不是问题, 因为对于一个神经稍微正常的人来说, 他不会拿着李四的身份证去证明自己是张三。第二个问题可以根据身份证上面的照片来判断。第三个问题就依赖于身份证本身的防伪标识和认证方的鉴别能力了。

上述的三个条件在本质上反映了用户凭证本身应该具有的属性, 即凭证与声明的一致性, 被认证人对凭证的拥有性, 以及凭证的合法性。我们不妨将此三点简称为用户凭证的三个属性。用户凭证的类型决定了认证的方式, WCF 支持一系列不同类型的用户凭证, 以满足不同认证需求。接下来按照上述的这三点来简单介绍几种使用比较普遍的凭证及相应的认证方式。

1. 用户名/密码认证

我们最常使用的认证方式莫过于采用验证用户名和密码的形式，以至于我们提到身份验证，很多人会马上想到密码。不妨通过前面讲到的用户凭证的三属性来分析用户名/密码凭证。

用户名代表身份（Identity），凭证与声明的一致性意味着被认证方声明的身份与用户名一致。被认证人对凭证的拥有性通过密码证明，密码属于绝对隐私信息，被认证人如果能够提供与所声明的身份相匹配的密码，就能够证明他是凭证的真正拥有者。由于用户名/密码凭证不属于证书型凭证，不需要合法机构颁发，合法性则无从说起。如果你选择了用户名/密码凭证，则 WCF 为你提供了三种认证模式：

- 将用户名映射为 Windows 账号，采用 Windows 认证；
- 采用 ASP.NET 的成员资格（Membership）模块；
- 自定义认证逻辑。

2. NTLM

Windows 认证是实现单点登录（SSO，Single Sign-On）最理想的方式。无论是采用域（Domain）模式还是工作组（Workgroup）模式，只要你以 Windows 账号和密码登录到某一台机器，就会得到一个凭证。在当前会话超时之前，你就可以携带该 Windows 凭证，自动登录到集成了 Windows 认证方式的所有应用，而无须频繁地输入相同的 Windows 账号和密码。

Windows 具有两种不同的认证协议，即 NTLM（NT LAN Manager）和 Kerberos，先来谈谈 NTLM。NTLM 是用在 Windows NT 和 Windows 2000 Server（或者之后版本）工作组环境及 AD 域环境中对 Windows NT 系统的认证。NTLM 采用一种质询/应答（Challenge/Response）消息交换模式。图 7-5 反映了当一个登录到客户端的用户访问服务端时，服务端如何验证访问者的真实身份。在 NTLM 中真正完成认证工作的是域控制器（DC，Domain Controller），它具有一个数据库，保存所有用户账号相关信息。

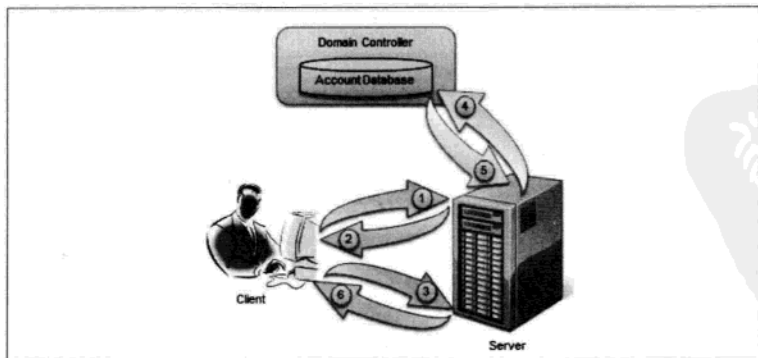


图 7-5 NTLM 的整个认证流程

- 步骤一：用户通过输入 Windows 账号和密码登录客户端主机。在成功登录后，客户端会缓存输入密码的哈希值，原始密码会被丢弃。如果试图访问服务器资源，需要向对方发送一个请求。该请求中包含一个以明文表示的用户名。
- 步骤二：服务端接收到请求后，生成一个 16 位的随机数，这个随机数被称为质询 (Challenge)。质询以明文的形式发送给客户端，在这之前服务端会先将其缓存起来。
- 步骤三：客户端在接收到服务端发回的质询后，用在步骤一中保存的密码哈希值对其加密，然后再将加密后的质询发送给服务端。
- 步骤四：服务端接收到客户端发送回来的加密后的质询后，会向域控制器发送针对客户端的验证请求。该请求主要包含以下三方面的内容：客户端用户名、客户端密码哈希值加密的质询和原始的质询。
- 步骤五、六：域控制器根据用户名获取该账号的密码哈希值，对原始的质询进行加密。如果加密后的质询和服务端发送的一致，则意味着用户拥有正确的密码，验证通过，否则验证失败。域控制器将验证结果发给服务端，并最终反馈给客户端。

3. Kerberos

Kerberos 比 NTLM 更高效、更安全，同时认证过程也相对复杂。Kerberos 这个名字来源于希腊神话，是冥界守护神兽的名字。Kerberos 是一个三头怪兽，之所以用它来命名一种完全认证协议，是因为整个认证过程涉及客户端、服务端和密钥分发中心 (KDC, Key Distribution Center) 三方。在 Windows 域环境中，KDC 的角色由域控制器来担当。

Kerberos 实际上是一种基于票据 (Ticket) 的认证方式。客户端要访问服务端的资源，需要首先购买服务端认可的票据。也就是说，客户端在访问服务端之前需要预先买好票，等待服务验票之后才能入场。但是这张票不能直接购买，需要一张认购权证。客户端在买票之前需要预先获得一张认购权证。这张认购权证和进入服务端的入场券均由 KDC 发售。图 7-6 基本揭示了 Kerberos 的整个认证流程。

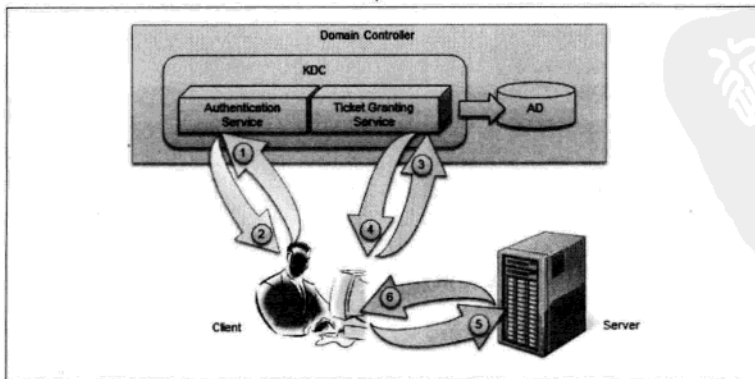


图 7-6 Kerberos 的整个认证流程

(1) 如何获得“认购权证”

先来看看客户端如何获得“认购权证”。这里的认购权证有个专有的名称，即 TGT (Ticket Granting Ticket)，而 TGT 的获取涉及 KDC 一个重要的服务，即 Kerberos 认证服务 (KAS, Kerberos Authentication Service)。在联机的情况下，当某个用户通过输入域账号和密码试图登录某台主机的时候，本机的 Kerberos 服务会向 KDC 的认证服务发送一个认证请求。该请求主要包括两部分内容：明文形式的用户名和经过密码派生的密钥加密的用于证明访问者身份的 Authenticator (很难找到一个比较贴切的中文翻译，Authenticator 在这里可以理解为仅限于验证双方预先知晓的内容，相当于联络暗号)。

当 KDC 接收到请求之后，通过 AD 获取该用户的信息，通过获取的密码信息生成一个密钥对 Authenticator 进行解密。如果解密后的内容和已知的内容一致，则证明请求者提供的密码正确，即确定了登录者的真实身份。

KAS 成功认证对方的身份之后，会先生成一个用于确保该用户和 KDC 之间通信安全的会话密钥，即登录会话密钥 (Logon Session Key)。该密钥被该用户密码派生的密钥进行加密。KAS 接着为该用户创建作为“认购权证”的 TGT。TGT 主要包含两方面的内容：用户相关信息和登录会话密钥。整个 TGT 则通过 KDC 自己的密钥进行加密后伴随加密后的登录会话密钥返回给客户端。KAS 的请求和回复的内容如图 7-7 所示。

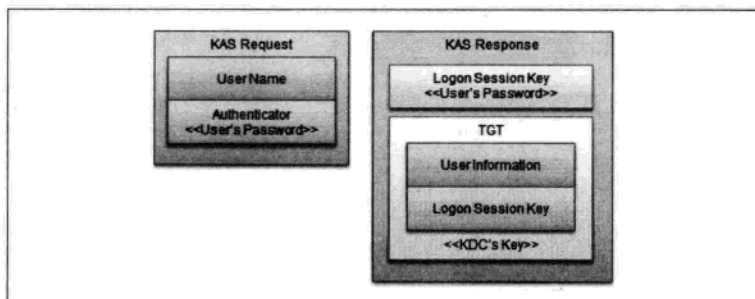


图 7-7 KAS 请求与回复

(2) 如何通过“认购权证”购买“入场券”

经过上面的步骤，客户端获取了购买进入同域中其他主机入场券的“认购凭证” TGT，并通过自己的密码派生的密钥解密得到登录会话密钥，客户端会在本地对它们进行缓存。如果现在它需要访问某台服务器的资源，就需要凭借这张 TGT 向 KDC 购买相应的入场券。这里的入场券也有一个专有的名称，即服务票据 (ST, Service Ticket)。ST 是通过 KDC 的另一个服务 TGS (Ticket Granting Service) 出售的。客户端先向 TGS 发送一个 ST 购买请求，该请求主要包含如下的内容：客户端用户名、通过登录会话密钥加密的 Authenticator、TGT 和访问的服务器 (其实是服务)。

TGS 接收到请求之后，先通过自己的密钥解密 TGT 并获取登录会话密钥。然后通过登

录会话密钥解密 Authenticator, 进而验证对方的真实身份。TGS 存在的一个根本目的有两点:

- 避免让用户的密码在客户端和 KDC 之间频繁传输而被窃取。
- 密码属于长生命周期密钥 (LongTerm Key, 我们一般不会频繁地更新自己的密码), 让它作为加密密钥的安全系数肯定小于一个频繁变换的短生命周期密钥 (Short Term Key)。而这个短生命周期密钥就是登录会话密钥, 它确保了客户端和 KDC 之间的通信安全。

TGS 完成对客户端的认证之后, 会生成一个用于确保客户端-服务端之间通信安全的会话密钥, 即服务会话密钥 (Service Session Key), 该会话密钥通过登录会话密钥进行加密, 然后出售给客户端需要的作为入场券的 ST。ST 主要包含两方面的内容: 客户端用户信息和服务会话密钥。整个 ST 通过服务端密码派生的密钥进行加密。最终两个被加密的服务会话密钥和 ST 回复给客户端。图 7-8 反映了 TGS 请求和返回消息的内容。

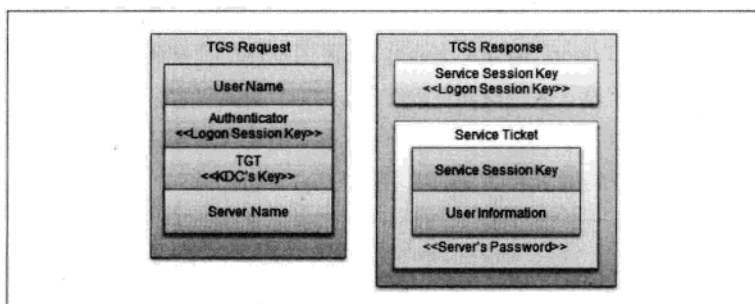


图 7-8 TGS 请求与回复

(3) 凭票入场

客户端接收到 TGS 回复后, 通过缓存的登录会话密钥解密获取服务会话密钥。同时它也得到了进入服务器的入场券 ST。它在进行服务访问的时候就可以借助这张 ST 凭票入场了。该服务会话密钥和 ST 会被客户端缓存。

但是服务端在接收到 ST 之后, 如何确保它是通过 TGS 购买, 而不是自己伪造的呢? 不要忘了 ST 是通过自己的密码派生的密钥进行加密的。具体的操作过程是这样的: 除了 ST 之外, 服务请求还附加一份通过服务会话密钥加密的 Authenticator。服务端在接收到请求之后, 先通过自己密码派生的密钥解密 ST, 并从中提取服务会话密钥, 然后通过提取出来的服务会话密钥解密 Authenticator, 进而验证客户端的真实身份。

到目前为止, 服务端已经完成了对客户端的验证, 但是整个认证过程还没有结束。因为我们需要的是双向验证 (Mutual Authentication)。现在服务端已经可以确保客户端是它所声称的那个用户, 客户端还没有确认它所访问的不是一个钓鱼服务呢。

为了解决客户端对服务端的验证, 客户端会在请求中内嵌另一个 Authenticator, 服务需要将其提取出来并使用服务会话密钥进行加密后发回给客户端。客户端再用缓存的服务

会话密钥进行解密，如果和之前的内容完全一样，则可以证明自己正在访问的服务端和自己拥有相同的服务会话密钥，而这个会话密钥不被外人知晓。图 7-9 反映了服务请求和回复的内容。

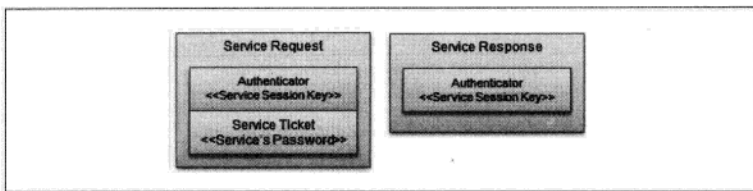


图 7-9 服务请求与回复

NTLM 仅仅帮助服务端认证客户端，而不能帮助客户端认证服务端，这可以从 NTLM 的整个认证流程看出来。所以只有当你采用基于 Kerberos 的 Windows 认证的时候，双向认证才被支持。

4. 数字证书 (Digital Certificate) 认证

在 7.1 节中我们站在非对称加密角度对数字证书进行了相应的介绍，在这里从用户凭证的角度进一步认识数字证书。照例采用用户凭证的三个属性来分析数字证书。

- 凭证与声明的一致性：证书的声明反映在与公钥绑定的与主题相关的信息。
- 持有人对凭证的拥有性：被认证方提供的数字证书具有相应的私钥。私钥的私有性在某种程度上证明了数字证书持有者就是该证书的拥有者。
- 证书的合法性：这可以通过颁发者对证书的数字签名来验证。

如果被认证方以数字证书作为用户凭证，认证方一般采用信任链 (Trust Chain) 模式对其实施认证。在该模式下，认证方从数字证书的直接颁发机构向上追溯，如果任何一个颁发机构是受信任的，那么认证成功。不过有时还是会采用其他的认证模式，比如严格比较证书主题信息甚至是序列号。

对于 WCF 来说，不仅客户端可以将数字证书作为证明自己身份的凭证，提供给服务端对自己进行认证，也可以将服务和某个数字证书绑定起来，通过证书代表服务的身份，供客户端进行验证。数字证书在 WCF 中具有十分广泛的应用。

7.2.2 绑定、安全模式与客户端凭证类型

整个安全传输是在 WCF 的信道层进行的，所以终结点采用哪种类型的绑定，以及对绑定的属性进行怎样的设置，决定了信道层最终采用何种机制实现消息的安全传输。我们可以通过绑定设置最终采用的安全模式，以及基于相应安全模式下进行认证和消息保护的行为。

不同的绑定类型由于其采用的传输协议不同,应用的场景也各有侧重,很难提供一种统一的应用编程接口完成基于不同绑定的安全设置,所以每一种绑定都具有各自用于安全设置相关的类型。但是对安全的设置,大部分系统预定义绑定(不是所有)都具有类似于如下代码片段所示的属性定义。

```
Public class XxxBinding
{
    Public XxxSecurity Security {}
}
Public class XxxSecurity
{
    Public XxxSecurityMode           Mode{get;set;}
    Public XxxTransportSecurity      Transport{get;set;}
    Public XxxMessageSecurity        Message{get;set;}
}
```

对于某个绑定 `XxxBinding` (`Xxx` 泛指某种绑定类型,所有带 `Xxx` 前缀的类型并不意味着它们代表完全一样的字符),它具有一个专属的 `XxxSecurity` 类型的 `Security` 属性。而这个 `XxxSecurity` 类型一般具有如下三个属性。

- **Mode:** 表示采用的安全模式;
- **Transport:** 用于针对 Transport 安全模式下的设置;
- **Message:** 用于针对 Message 安全模式下的设置。

对于围绕着绑定进行的安全设置,首要的任务就是指定采用的安全模式。在安全模式确定之后,客户端凭证的选择决定了认证方最终采用怎样的认证机制。接下来就来谈谈针对各种常用的系统预定义绑定、安全模式和基于安全模式的客户端凭证如何设置。

1. BasicHttpBinding

下面的代码片段表示 `BasicHttpBinding` 安全相关应用编程接口的定义,这和上面给出的“泛型绑定”的定义完全一致。属性 `Security` 返回一个 `System.ServiceModel.BasicHttpSecurity` 对象,用于针对 `BasicHttpBinding` 的安全设置。`BasicHttpSecurity` 的 `Mode`、`Transport` 和 `Message` 三种属性的类型分别为 `System.ServiceModel.BasicHttpSecurityMode`、`System.ServiceModel.HttpTransportSecurity` 和 `System.ServiceModel.BasicHttpMessageSecurity`。

```
public class BasicHttpBinding : Binding, IBindingRuntimePreferences
{
    //其他成员
    public BasicHttpSecurity Security { get; set; }
}
public sealed class BasicHttpSecurity
{
    //其他成员
    public BasicHttpSecurityMode           Mode { get; set; }
    public HttpTransportSecurity          Transport { get; set; }
    public BasicHttpMessageSecurity        Message { get; set; }
}
```


枚举 `BasicHttpSecurityMode` 中定义了 `BasicHttpBinding` 支持的 5 种安全模式。其中 `None` 为默认选项，表示并不采用任何安全机制。`Transport`、`Message` 和 `TransportWithMessageCredential` 分别表示之前介绍的 `Transport`、`Message` 和 `Mixed` 安全模式。

```
public enum BasicHttpSecurityMode
{
    None,
    Transport,
    Message,
    TransportWithMessageCredential,
    TransportCredentialOnly
}
```

`TransportWithMessageCredential` 表示“使用 `Message` 安全模式凭证的 `Transport` 模式”。由于在 `Mixed` 安全模式下，对客户端的认证是通过 `Message` 安全模式实现的，所以要求客户端采用基于 `Message` 安全模式的凭证。而除客户端认证外的其他安全要素的实现则都是采用 `Transport` 安全模式。所以 `TransportWithMessageCredential` 和我们讲的 `Mixed` 模式从语义上讲是一致的。

`TransportCredentialOnly` 是 `BasicHttpBinding` 所独有的安全模式。它只提供针对 HTTP 的客户端认证，并不能提供消息一致性和机密性的保证。

(1) `HttpTransportSecurity`

`HttpTransportSecurity` 表示 `Transport` 模式下的安全设置。通过 `ClientCredentialType` 属性，可以设置客户端凭证的类型。该属性类型为 `System.ServiceModel.HttpClientCredentialType` 枚举，6 个枚举值表示支持的 6 种客户端凭证类型。

```
public sealed class HttpTransportSecurity
{
    //其他成员
    public HttpClientCredentialType ClientCredentialType {get; set; }
}
public enum HttpClientCredentialType
{
    None,
    Basic,
    Digest,
    Ntlm,
    Windows,
    Certificate
}
```

定义在 `HttpClientCredentialType` 中的 6 种不同的客户端用户凭证类型体现了服务端针对客户端的不同的认证方式。

- **None:** 客户端无须指定用户凭证，即匿名认证。为默认值。
- **Basic:** 采用 `Basic` 认证方式进行客户端认证。在这种认证方式下，客户端需要提供有效的用户名和密码，但是仅采用较弱的方式对密码进行加密。当且仅当确定客户端和服务端之间的连接绝对安全的前提下，才能用这种认证方式。

- **Digest**: 采用 Digest 认证方式进行客户端认证。Digest 认证提供与 Basic 类似的认证功能,但是在安全性上有所提升。主要体现在并不是直接将用户名和密码直接进行网络传输,而是对其进行哈希计算 (MD5) 得到一个哈希码 (此过程又称为 Message Digest),最终传输的是该哈希码。
- **Ntlm**: 表示使用基于 NTLM 方式的 Windows 集成认证。
- **Windows**: 表示使用 Windows 集成认证。如果能够使用 Kerberos,则直接采用 Kerberos 进行认证,否则才使用 NTLM。
- **Certificate**: 表示客户端的身份通过一个 X.509 数字证书表示,服务端通过校验证书的方式来确定客户端的真实身份。

可以通过编程的方式来设置绑定的安全模式和客户端用户凭证类型。如下面的代码片段所示,我们为 BasicHttpBinding 设置了 Transport 安全模式,并将其客户端凭证设置成 Windows。所有基于 HTTP 的绑定 Transport 安全模式都是通过 HTTPS 实现的,所以在选择 Transport 和 TransportWithMessageCredential 安全模式的情况下,终结点地址必须是一个 HTTPS 地址。

服务寄宿代码:

```
using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    var binding = new BasicHttpBinding(BasicHttpSecurityMode.Transport);
    binding.Security.Transport.ClientCredentialType =
        HttpClientCredentialType.Windows;
    host.AddServiceEndpoint(typeof(ICalculator),
                            binding,
                            "https://localhost/calculatorservice");

    host.Open();
    //...
}
```

服务调用代码:

```
var binding = new BasicHttpBinding(BasicHttpSecurityMode.Transport);
binding.Security.Transport.ClientCredentialType =
    HttpClientCredentialType.Windows;
using (ChannelFactory<ICalculator> channelFactory = new
    ChannelFactory<ICalculator>(binding, "https://localhost/calculatorservice"))
{
    ICalculator calculator = channelFactory.CreateChannel();
    double result = calculator.Add(1, 2);
    //...
}
```

BasicHttpBinding 的配置节中具有一个 <security> 的子节点,用于进行安全相关的设置。安全模式通过该节点的 mode 属性设置。而基于 Transport 模式相关的设置则配置在 <security>/<transport> 配置节中,其中配置属性 clientCredentialType 表示客户端凭证类型。在下面给出的配置片段中,定义了一个采用 Transport 安全模式,并采用 Certificate 客户端凭证

类型的 `BasicHttpBinding`。

```
<system.serviceModel>
  <bindings>
    <basicHttpBinding>
      <binding name="transportBinding">
        <security mode="Transport">
          <transport clientCredentialType="Certificate"/>
        </security>
      </binding>
    </basicHttpBinding>
  </bindings>
  ...
</system.serviceModel>
```

(2) BasicHttpMessageSecurity

`BasicHttpMessageSecurity` 用于对 `BasicHttpBinding` 关于 Message 安全模式进行相关设置。类型为 `System.ServiceModel.BasicHttpClientCredentialType` 枚举的 `ClientCredentialType` 属性表示客户端凭证类型。`BasicHttpClientCredentialType` 的两个枚举值 (`UserName` 和 `Certificate`) 分别表示用户名/密码凭证和 X.509 证书凭证。在默认的情况下采用用户名/密码凭证。`BasicHttpMessageSecurity` 和 `BasicHttpClientCredentialType` 的相关定义如下面的代码片段所示。

```
public sealed class BasicHttpClientSecurity
{
    public BasicHttpClientCredentialType ClientCredentialType { get; set; }
}
public enum BasicHttpClientCredentialType
{
    UserName,
    Certificate
}
```

对于上述的两种客户端凭证, `UserName` 只能用在 `Mixed` 模式下。当选择了 `Message` 模式后, 只能选择 `Certificate` 作为客户端凭证。举个例子, 通过如下一段代码对服务 `CalculatorService` 进行寄宿, 终结点采用的 `BasicHttpBinding` 的安全模式为 `Message`。

```
using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    var binding = new BasicHttpBinding(BasicHttpSecurityMode.Message);
    host.AddServiceEndpoint(typeof(ICalculator), binding,
        "http://localhost/calculatorservice");
    host.Open();
    //...
}
```

当 `ServiceHost` 被开启的时候, 如图 7-10 所示的 `InvalidOperationException` 异常被抛出来, 并提示 “`BasicHttpBinding` 绑定要求 `BasicHttpClientSecurity.Message.ClientCredentialType` 等效于安全消息的 `BasicHttpClientCredentialType.Certificate` 凭据类型。为 `UserName` 凭据选择 `Transport` 或 `TransportWithMessageCredential` 安全性”。实际上这个异常消息不太正

确, 因为 Transport 模式下根本就不存在 UserName 凭证类型。

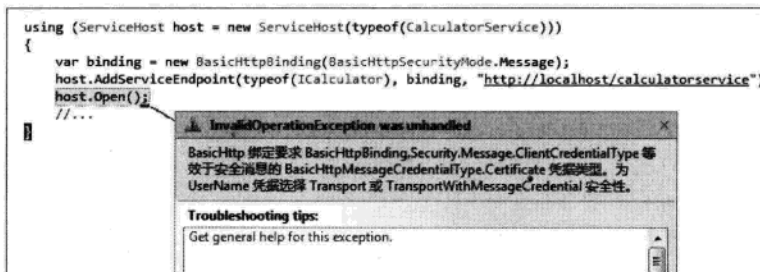


图 7-10 Message 模式下将客户端凭证类型设置为 UserName 抛出的异常

BasicHttpBinding 配置节下的<security>/<message>节点用于 Message 安全模式的设置, 其 clientCredentialType 属性表示客户端凭证类型。下面的配置片段定义了两个 BasicHttpBinding, 前者采用基于 Message 模式+X.509 证书凭证, 后者采用基于 Mixed 模式+用户名/密码凭证。

```
<system.serviceModel>
  <bindings>
    <basicHttpBinding>
      <binding name="messageBinding">
        <security mode="Message">
          <message clientCredentialType="Certificate"/>
        </security>
      </binding>
      <binding name="transportWithMessageCredentialBinding">
        <security mode="TransportWithMessageCredential">
          <message clientCredentialType="UserName" />
        </security>
      </binding>
    </basicHttpBinding>
  </bindings>
  ...
</system.serviceModel>
```

2. WSHttpBinding、WS2007HttpBinding 和 WSDualHttpBinding

接下来分析另外三个基于 HTTP 的绑定 (WSHttpBinding、WS2007HttpBinding 和 WSDualHttpBinding), 先来介绍 WSDualHttpBinding。

(1) WSDualHttpBinding

所有 HTTP 绑定的 Transport 安全模式都是通过 HTTPS 实现的。HTTPS 要确保从 A 到 B 的安全连接, 要求 B 是一个绑定了数字证书的 HTTPS 站点。正是由于这样的原因, Transport 安全模式不能应用于基于 HTTP 的双向通信, 因为服务端对客户的安全连接得不到保障。所以 WSDualHttpBinding 只能提供基于 Message 安全模式的支持。

我们通过 System.ServiceModel.WSDualHttpSecurity 进行针对 WSDualHttpBinding 的安全设置。WSDualHttpSecurity 通过类型为 System.ServiceModel.WSDualHttpSecurityMode 的

Mode 属性返回采用的安全模式。WSDualHttpSecurityMode 的两个枚举值 (None 和 Message) 表示 WSDualHttpBinding 支持的安全模式。这也印证了关于 WSDualHttpBinding 不能够提供针对 Transport 安全的论点。在默认情况下, WSDualHttpBinding 采用 Message 安全模式。

用于进行 Message 安全设置的 Message 属性返回的是一个 System.ServiceModel.MessageSecurityOverHttp 类型对象。表示客户端凭证类型的 ClientCredentialType 属性类型为 System.ServiceModel.MessageCredentialType 枚举, 5 个枚举值 (None、Windows、UserName、Certificate 和 IssuedToken) 表示支持的 5 种客户端凭证类型, 其中 Windows 为默认的选项。WSDualHttpBinding 安全相关的应用编程接口的定义体现在如下所示的代码片段中。

```
public sealed class WSDualHttpSecurity
{
    //其他成员
    public MessageSecurityOverHttp Message { get; set; }
    public WSDualHttpSecurityMode Mode {get; set; }
}
public class MessageSecurityOverHttp
{
    //其他成员
    public MessageCredentialType ClientCredentialType { get; set; }
}
public enum MessageCredentialType
{
    None,
    Windows,
    UserName,
    Certificate,
    IssuedToken
}
public enum WSDualHttpSecurityMode
{
    None,
    Message
}
```

(2) WSHttpBinding 和 WS2007HttpBinding

WSHttpBinding 的 Security 属性返回的是一个 System.ServiceModel.WSHttpSecurity 类型的对象。由于 WS2007HttpBinding 直接继承自 WSHttpBinding, 所以它直接将该属性继承下来。表示安全模式的 Mode 属性类型为 System.ServiceModel.SecurityMode 枚举, 4 个枚举项 (None、Transport、Message 和 TransportWithMessageCredential) 表示 WSHttpBinding 和 WS2007HttpBinding 支持的 4 种安全模式, 其中 Message 为默认选项。

WSHttpSecurity 的 Transport 属性返回一个 System.ServiceModel.HttpTransportSecurity 类型的对象, 用于进行 Transport 安全的相关设置。表示客户端凭证类型的 ClientCredentialType 属性类型为 HttpClientCredentialType 枚举, 意味着 WSHttpBinding/WS2007HttpBinding 和 BasicHttpBinding 在 Transport 模式下具有相同的客户端凭证类型集。不同的是 WSHttpBinding

和 WS2007HttpBinding 默认情况下采用 Windows 凭证。

WSHttpSecurity 用户具有 Message 安全模式设置的 Message 属性类型 System.ServiceModel.NonDualMessageSecurityOverHttp 是 MessageSecurityOverHttp 的子类, 所以它直接继承了定义在 MessageSecurityOverHttp 中的 ClientCredentialType 属性。这意味着三种 WS 绑定 (WSHttpBinding、WS2007HttpBinding 和 WSDualHttpBinding) 在 Message 或者 Mixed 安全模式下具有相同的客户端凭证类型集, 默认采用 Windows 凭证。WSHttpBinding 和 WS2007HttpBinding 安全相关的应用编程接口的定义反映在如下的代码片段中。

```
public class WSHttpBinding : WSHttpBindingBase
{
    //其他成员
    public WSHttpSecurity Security { get; set; }
}
public class WS2007HttpBinding:WSHttpBinding
{
    //省略成员
}
public sealed class WSHttpSecurity
{
    //其他成员
    public SecurityMode Mode { get; set; }
    public HttpTransportSecurity Transport { get; set; }
    public NonDualMessageSecurityOverHttp Message {get; set; }
}
public enum SecurityMode
{
    None,
    Transport,
    Message,
    TransportWithMessageCredential
}
public sealed class HttpTransportSecurity
{
    //其他成员
    public HttpClientCredentialType ClientCredentialType { get; set; }
}
public sealed class NonDualMessageSecurityOverHttp : MessageSecurityOverHttp
{
    //省略成员
}
```

WSHttpBinding、WS2007HttpBinding 和 WSDualHttpBinding 采用和 BasicHttpBinding 一样的编程方式和配置结构, 所以在这里就不再对此进行重复介绍了。IssuedToken 代表的是一种基于 WCS (Windows Card Space) 的认证方式, 而这又涉及一个更加宽泛的话题, 即安全联合 (Security Federation), 本书并不涉及此内容。

3. NetNamedPipeBinding 和 NetTcpBinding

由于 NetNamedPipeBinding 和 NetTcpBinding 在安全传输上具有相似的实现, 因此我们将它们放在一起介绍。先来看看 NetNamedPipeBinding。

(1) NetNamedPipeBinding

NetNamedPipeBinding 只能用于同一台机器上的不同进程之间的通信 (IPC, Inter-Process Communication), 所以根本不需要 Message 模式的安全。因此在表示 NetNamedPipeBinding 安全的 System.ServiceModel.NetNamedPipeSecurity 类型中, 表示支持的安全模式的 Mode 属性对应的 System.ServiceModel.NetNamedPipeSecurityMode 枚举仅具有 None 和 Transport 两个选项。在默认的情况下, NetNamedPipeBinding 采用 Transport 安全模式。

表示 Transport 模式安全的 NamedPipeTransportSecurity 类并不存在 ClientCredentialType 属性, 因为它总是采用 Windows 作为其客户端凭证。NetNamedPipeBinding 安全相关的应用编程接口如下面的代码片段所示。

```
public class NetNamedPipeBinding : Binding, IBindingRuntimePreferences
{
    //其他成员
    public NetNamedPipeSecurity Security { get; set; }
}
public sealed class NetNamedPipeSecurity
{
    //其他成员
    public NetNamedPipeSecurityMode Mode { get; set; }
    public NamedPipeTransportSecurity Transport { get; set; }
}
public enum NetNamedPipeSecurityMode
{
    None,
    Transport
}
public sealed class NamedPipeTransportSecurity
{
    //不存在 ClientCredentialType 属性
}
```

(2) NetTcpBinding

NetTcpBinding 的 Security 属性返回的是 System.ServiceModel.NetTcpSecurity 对象。表示安全模式的 NetTcpSecurity 的 Mode 属性返回的是我们提到过的 SecurityMode 枚举, 意味着 NetTcpSecurity 和 WSHttpBinding/WS2007HttpBinding 支持相同的安全模式集 (None、Transport、Message 和 Mixed)。在默认的情况下, NetTcpBinding 采用 Transport 安全模式。

NetTcpSecurity 的 Transport 属性返回的是一个用于进行 Transport 安全设置的 System.ServiceModel.TcpTransportSecurity 类型对象。TcpTransportSecurity 的 ClientCredentialType 属性以 System.ServiceModel.TcpClientCredentialType 枚举的形式表示采用的客户端凭证类型。定义在 TcpClientCredentialType 中的三个枚举值 (None、Windows 和 Certificate) 表示 NetTcpBinding 在 Transport 模式下支持的所有客户端凭证类型。在默认情况下, NetTcpBinding 采用 Windows 凭证。

通过 Message 属性返回的用于进行 Message 安全设置的则是一个 System.ServiceModel.

MessageSecurityOverTcp 类型对象。MessageSecurityOverTcp 用于表示客户端凭证类型的 ClientCredentialType 属性的依然是 MessageCredentialType, 意味着 NetTcpBinding 和上述的三个 WS 绑定在 Message 模式下, 具有相同的客户端凭证集。在默认情况下, NetTcpBinding 采用 Windows 凭证。NetTcpBinding 安全相关的应用编程接口如下面的代码片段所示。

```
public class NetTcpBinding : Binding, IBindingRuntimePreferences
{
    //其他成员
    public NetTcpSecurity Security { get;set}
}
public sealed class NetTcpSecurity
{
    //其他成员
    public SecurityMode Mode { get; set; }
    public TcpTransportSecurity Transport { get; set; }
    public MessageSecurityOverTcp Message { get; set; }
}
public sealed class TcpTransportSecurity
{
    //其他成员
    public TcpClientCredentialType ClientCredentialType { get; set; }
}
public sealed class MessageSecurityOverTcp
{
    //其他成员
    public MessageCredentialType ClientCredentialType { get; set; }
}
public enum TcpClientCredentialType
{
    None,
    Windows,
    Certificate
}
```

4. NetMsmqBinding

NetMsmqBinding 的 Security 属性的类型为 System.ServiceModel.NetMsmqSecurity。表示安全模式的 Mode 属性返回一个 System.ServiceModel.NetMsmqSecurityMode 枚举。NetMsmqSecurityMode 枚举的定义反映了 NetMsmqBinding 支持的安全模式集与其他系统定义的绑定都不太一样。定义在 NetMsmqSecurityMode 的 4 个枚举值 (None、Transport、Message 和 Both) 反映了 NetMsmqBinding 支持的 4 种安全模式。

NetMsmqBinding 具有一种独有的安全模式 Both。这种模式意味着同时采用 Transport 和 Message。NetMsmqBinding 并不支持 Mixed (TransportWithMessageCredential) 模式。在默认情况下, NetMsmqBinding 采用 Transport 安全模式。

通过 NetMsmqSecurity 的 Transport 属性返回的用于进行 Transport 安全设置的是一个类型为 System.ServiceModel.MsmqTransportSecurity 的对象。和 NetNamedPipeBinding 类似, MsmqTransportSecurity 并没有一个 ClientCredentialType 属性。这是因为在 Transport 模式下,

NetMsmqBinding 总是采用 Windows 凭证。而用于进行 Message 安全设置的 Message 属性类型为 System.ServiceModel.MessageSecurityOverMsmq。MessageSecurityOverMsmq 具有一个类型为 MessageCredentialType 的 ClientCredentialType 属性。NetMsmqSecurity 安全相关的应用编程接口定义反映在下面的代码片段中。

```
public class NetMsmqBinding : MsmqBindingBase
{
    //其他成员
    public NetMsmqSecurity Security {get; set; }
}
public sealed class NetMsmqSecurity
{
    //其他成员
    public NetMsmqSecurityMode Mode { get; set; }
    public MsmqTransportSecurity Transport { get; set; }
    public MessageSecurityOverMsmq Message { get; set; }
}
public enum NetMsmqSecurityMode
{
    None,
    Transport,
    Message,
    Both
}
public sealed class MsmqTransportSecurity
{
    //不存在 ClientCredentialType 属性
}
public sealed class MessageSecurityOverMsmq
{
    //其他成员
    public MessageCredentialType ClientCredentialType {get; set; }
}
```

5. 总结

接下来我们通过表格的形式对不同类型的系统预定义针对不同的安全模式下具有的特性进行一下总结。表 7-1 表示系统定义绑定对不同安全模式的支持（WSHttpBinding 与 WS2007HttpBinding 具有相同的安全模式支持策略，这里仅列出了 WSHttpBinding）。针对这个表格，可以看出：

- 所有的绑定都可以不采用任何安全传输机制，即支持 None 安全模式。
- BasicHttpBinding 的默认模式为 None，WS 相关的绑定默认模式为 Message，而局域网相关绑定的默认模式为 Transport。
- 除了 NetNamedPipeBinding，所有的绑定都支持 Message 安全模式。
- 对于所有支持 Message 模式的绑定，除了 NetMsmqBinding，都支持 Mixed 模式。
- 除了 WSDualHttpBinding，所有的绑定都支持 Transport 模式。

- 只有 BasicHttpBinding 支持 TransportCredentialOnly 模式。
- 只有 NetMsmqBinding 支持 Both 安全模式。

表 7-1 系统定义绑定对不同安全模式的支持

	BasicHttp-Binding	WSHttp-Binding	WSDualHttp-Binding	NetNamedPipeBinding	NetTcp-Binding	NetMsmq-Binding
None	Default	Yes	Yes	Yes	Yes	Yes
Transport	Yes	Yes	No	Default	Default	Default
Message	Yes	Default	Default	No	Yes	Yes
Mixed	Yes	Yes	Yes	No	Yes	No
Both	No	No	No	No	No	Yes
Transport-Credential-Only	Yes	No	No	No	No	No

接下来比较一下对应之前介绍的这些常用的系统定义绑定在 Transport 安全模式下，对客户端凭证类型集的支持有何不同。从表 7-2 反映的数据可以看出：

- 三种基于 HTTP 的绑定（不包括不支持 Transport 安全模式的 WSDualHttpBinding）支持所有类型的客户端凭证（实际上客户端凭证类型都是通过枚举 HttpClientCredentialType 表示的）。
- 除了 BasicHttpBinding 在默认的情况下 None 作为客户端凭证类型（匿名客户端）之外，其他绑定的默认客户端凭证类型都是 Windows。
- NetTcpBinding 支持三种客户端凭证类型：None、Windows 和 Certificate。
- NetNamedPipeBinding 和 NetMsmqBinding 支持唯一的客户端凭证类型 Windows。

表 7-2 常用的系统定义绑定在 Transport 模式下对客户端凭证类型集的支持

	BasicHttp-Binding	WSHttp-Binding	WSDualHttp-Binding	NetNamedPipeBinding	NetTcp-Binding	NetMsmq-Binding
None	Default	Yes	-	No	Yes	No
Basic	Yes	Yes	-	No	No	No
Digest	Yes	Yes	-	No	No	No
Windows	Yes	Default	-	Default	Default	Default
Ntlm	Yes	Yes	-	No	No	No
Certificate	Yes	Yes	-	No	Yes	No

不同的绑定针对于 Message 模式（或者 Mixed 模式）下的客户端凭证类型的支持就非常清晰了。因为除了 BasicHttpBinding 采用 BasicHttpMessageCredentialType 枚举表示其支持的

客户端凭证类型之外，其他所有的绑定（不包括不支持 Message 安全模式的 NetNamedPipeBinding）都使用 MessageCredentialType 表示客户端凭证类型。并且，在 Message 模式下，不同类型的绑定对不同客户端凭证类型集的支持反映在表 7-3 中。

表 7-3 在 Message 模式下不同类型的绑定对不同客户端凭证类型集的支持

	BasicHttp- Binding	WSHttp- Binding	WSDual- HttpBinding	NetNamed- PipeBinding	NetTcp- Binding	NetMsmq- Binding
None	No	Yes	Yes	-	Yes	Yes
User Name	Default	Yes	Yes	-	Yes	Yes
Windows	No	Default	Default	-	Default	Default
Certificate	Yes	Yes	Yes	-	Yes	Yes
Issued-Token	No	Yes	Yes	-	Yes	Yes

7.2.3 服务认证

接下来正式讨论身份认证的主题。WCF 中的认证属于“双向认证”，既包括服务对客户端的认证（以下简称客户端认证），也包括客户端对服务的认证（以下简称服务认证）。客户端认证和服务认证从本质上并没有什么不同，无非都是被认证一方提供相应的用户凭证供对方对自己的身份进行验证。

1. TLS/SSL 对服务的认证

对于 TLS/SSL，客户端和服务在为建立安全上下文而进行的协商过程中会验证服务端的 X.509 证书是否值得信任。对于服务证书的验证实际上可以看成是一种服务认证。

TLS/SSL 是实现 Transport 安全模式的一种主要的方式（不是唯一方式）。对于所有基于 HTTP 的绑定（主要指 BasicHttpBinding、WSHttpBinding 和 WS2007HttpBinding，而 WSDualHttpBinding 不支持 Transport 安全模式），如果选择了 Transport 或 Mixed 安全模式，不论采用怎样的认证方式，底层的实现总是基于 TLS/SSL（HTTPS）的。

对于 NetTcpBinding 来说，如果采用 Transport 安全模式，并且采用非 Windows 认证（客户端凭证类型选择 None 或 Certificate），最终的传输安全的实现也是基于 TLS/SSL（SSL Over TCP）的。如果选择 Mixed 安全模式，不论选择怎样的客户端凭证类型，WCF 最终都会采用 TLS/SSL 来提供对传输安全的实现。在这两种情况下，总是需要选择一个 X.509 证书作为服务的凭证。

```
<system.serviceModel>
  <bindings>
    <netTcpBinding>
      <binding name="transportTcpBinding">
        <security mode="Transport">
```

```

        <transport clientCredentialType="None"/>
    </security>
</binding>
</netTcpBinding>
</bindings>
<services>
    <service name="Artech.WcfServices.Service.CalculatorService">
        <endpoint address="net.tcp://127.0.0.1/calculatorservice"
            binding="netTcpBinding"
            bindingConfiguration="transportTcpBinding"
            contract="Artech.WcfServices.Service.Interface.
                ICalculator" />
    </service>
</services>
</system.serviceModel>

```

在上面的配置中，寄宿服务的终结点采用 Transport 安全模式的 NetTcpBinding 绑定，但是却采用匿名的认证方式（客户端凭证类型为 None）。在对服务进行寄宿时，会抛出如图 7-11 所示的 `InvalidOperationException` 异常，提示“未提供服务证书。请在 `ServiceCredentials` 中指定服务证书”。

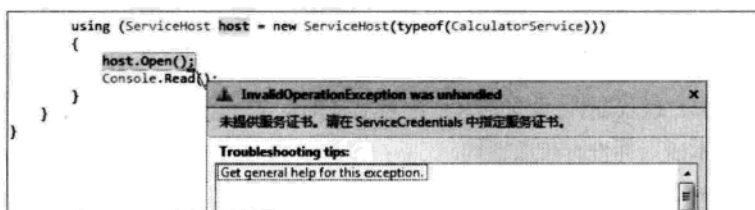


图 7-11 采用 Transport 安全模式未指定服务证书抛出的异常

作为服务凭证的证书通过服务行为 `System.ServiceModel.Description.ServiceCredentials` 来指定。具体来说需要用到类型为 `System.ServiceModel.Security.X509CertificateRecipientServiceCredential` 的只读属性 `ServiceCertificate`。`X509CertificateRecipientServiceCredential` 对象实际上是对一个 `System.Security.Cryptography.X509Certificates.X509Certificate2` 对象的封装，它定义了若干 `SetCertificate` 方法重载用于指定一个 X.509 证书作为服务的凭证。`ServiceCredentials` 和 `X509CertificateRecipientServiceCredential` 的相关定义反映在如下所示的代码片段中。

```

public class ServiceCredentials : SecurityCredentialsManager,
    IServiceBehavior
{
    //其他成员
    public X509CertificateRecipientServiceCredential ServiceCertificate
    { get; }
}
public sealed class X509CertificateRecipientServiceCredential
{
    //其他成员
    public void SetCertificate(string subjectName);
    public void SetCertificate(string subjectName, StoreLocation storeLocation,

```

```

        StoreName storeName);
    public void SetCertificate(StoreLocation storeLocation, StoreName storeName,
        X509FindType findType, object findValue);

    public X509Certificate2 Certificate { get; set; }
}

```

可以通过编程的方式来为寄宿的服务设置一个代表服务凭证的 X.509 证书。在下面给出的代码片段中, 为服务设置一个主体名称为 Jinnan-PC (我的机器名) 的 X.509 证书, 该证书是一个基于个人存储 (Personal Store, 通过 StoreName.My 表示) 的本机 (StoreLocation.LocalMachine) 证书。

```

using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    ServiceCredentials serviceCredentials =
        host.Description.Behaviors.Find<ServiceCredentials>();
    if (null == serviceCredentials)
    {
        serviceCredentials = new ServiceCredentials();
        host.Description.Behaviors.Add(serviceCredentials);
    }
    serviceCredentials.ServiceCertificate.SetCertificate(
        StoreLocation.LocalMachine,
        StoreName.My,
        X509FindType.FindBySubjectName,
        "Jinnan-PC");
    host.Open();
    //...
}

```

对于上面一段设置服务证书的代码, 可以通过下面的一段配置来代替。

```

<system.serviceModel>
...
<services>
  <service name="Artech.WcfServices.Service.CalculatorService"
    behaviorConfiguration="serviceCertificateBehavior">
    ...
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior name="serviceCertificateBehavior">
      <serviceCredentials>
        <serviceCertificate storeLocation="LocalMachine"
          storeName="My"
          x509FindType="FindBySubjectName"
          findValue="Jinnan-PC" />
      </serviceCredentials>
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>

```

如果采用基于 TLS/SSL 的 Transport 安全模式, 对服务证书的验证方式会因为绑定类型的不同而具有小小的差异。

2. 实例演示：创建基于 TLS/SSL 的服务

接下来通过一个简单的例子来演示如何在 WCF 服务中使用基于 TLS/SSL 的 Transport 安全。该实例会涉及两种不同的绑定类型 (WS2007HttpBinding 和 NetTcpBinding) 和寄宿方式 (自我寄宿和 IIS 寄宿)。我们采用惯用的计算服务的例子，演示实例的解决方案具有如图 7-12 所示的结构。Service.Interface 和 Service 为两个类库项目，分别用于定义服务契约和实现契约的服务类型。而 Hosting 和 Client 为两个控制台应用，前者用于进行服务寄宿 (自我寄宿)，后者用于模拟客户端程序。

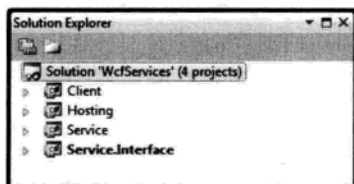


图 7-12 演示实例解决方案的结构

下面的代码片段代码了分别定义在 Service.Interface 和 Service 项目中的服务契约接口 ICalculator 和具体的服务类型 CalculatorService。

服务契约：

```
using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface ICalculator
    {
        [OperationContract]
        double Add(double x, double y);
    }
}
```

服务类型：

```
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Service
{
    public class CalculatorService : ICalculator
    {
        public double Add(double x, double y)
        {
            return x + y;
        }
    }
}
```

TLS/SSL 需要通过协商的方式生成一个用于消息签名和加密的会话密钥，而会话密钥的交换依赖一个 X.509 证书以确保安全。我们需要通过 MakeCert.exe 工具创建一个证书。为了演示证书正常的信任链，我们先通过下面的命令行生成一个表示 CA (名称为 RootCA) 的证书。

```
Makecert -n "CN=RootCA" -r -sv C:\RootCA.pvk C:\RootCA.cer
```

上面的命令行在执行的过程中,会弹出两个用于输入密码的对话框。需要输入相应的密码用于包括生成的两个文件,一个是包含私钥的文件 RootCA.pvk,另一个是证书文件 RootCA.cer,它们都保存在 C 盘根目录下。

然后通过如下的命令行创建一个以机器名作为主题名称(Jinnan-PC 是我的本机名称)的证书,并以上面创建证书对应的 CA(RootCA)作为该证书的颁发者(-ic C:\RootCA.cer -iv C:\RootCA.pvk)。该证书最终自动保存到本机(-sr LocalMachine)的个人存储区(-ss My)。

```
Makecert -n "CN=Jinnan-PC" -ic C:\RootCA.cer -iv C:\RootCA.pvk -sr  
LocalMachine -ss My -pe -sky exchange
```

(1) SSL Over TCP

我们先采用 NetTcpBinding+SSL Over TCP 实现传输安全。在如下所示的服务寄宿配置中,为寄宿的服务添加了一个基于 NetTcpBinding 的终结点。该绑定采用 Transport 安全模式,客户端凭证为 None(匿名客户端)。通过 ServiceCredentials 服务行为将创建的证书作为服务凭证。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="transportTcpBinding">
          <security mode="Transport">
            <transport clientCredentialType="None"/>
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService"
        behaviorConfiguration="serviceCertificateBehavior">
        <endpoint address="net.tcp://Jinnan-PC/calculatorservice"
          binding="netTcpBinding"
          bindingConfiguration="transportTcpBinding"
          contract="Artech.WcfServices.Service.Interface.
            ICalculator" />
        </service>
      </services>
      <behaviors>
        <serviceBehaviors>
          <behavior name="serviceCertificateBehavior">
            <serviceCredentials>
              <serviceCertificate storeLocation="LocalMachine"
                storeName="My"
                x509FindType="FindBySubjectName"
                findValue="Jinnan-PC" />
            </serviceCredentials>
          </behavior>
        </serviceBehaviors>
      </behaviors>
    </system.serviceModel>
  </configuration>
```

我们省略了简单的服务寄宿的代码,直接来看看客户端的配置和代码。如下面的配置所示,进行服务调用的终结点采用了一个与服务端匹配的 NetTcpBinding。

配置:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="transportTcpBinding">
          <security mode="Transport">
            <transport clientCredentialType="None"/>
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
    <client>
      <endpoint name="calculatorService"
        address="net.tcp://jinnan-PC/calculatorService"
        binding="netTcpBinding"
        bindingConfiguration="transportTcpBinding"
        contract="Artech.WcfServices.Service.Interface.
        ICalculator"/>
    </client>
  </system.serviceModel>
</configuration>
```

服务调用程序:

```
using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService"))
{
    ICalculator calculator = channelFactory.CreateChannel();
    Console.WriteLine("x + y = {2} when x = {0} and y = {1}", 1, 2, calculator.
Add(1, 2)); ;
}
Console.Read();
```

完成所有编程和配置工作之后,先后启动 Hosting 和 Client 这两个控制台程序,你会发现服务并不能正常地调用,而是抛出如图 7-13 所示的 SecurityNegotiationException 异常,提示服务证书不受信任。(S701)

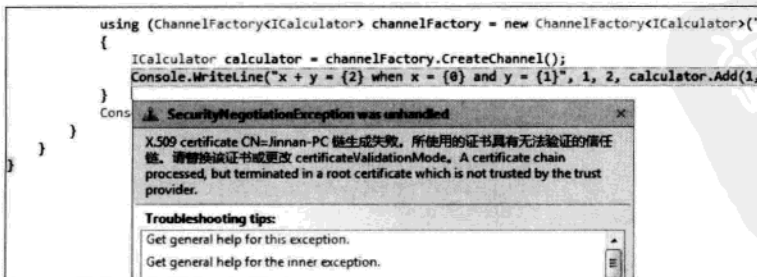


图 7-13 服务证书不受信任导致的异常

之所以会抛出这样的异常，原因在于 WCF 默认采用 ChainTrust 模式对服务证书进行验证。该认证模式要求服务证书的颁发机构必须是“受信任根证书颁发机构(Trusted Root Certification Authorities)”。为了解决这个问题，我们有如下两种“解决方案”。

- 将服务证书的颁发机构纳入受信任根证书颁发机构中。可以通过 MMC 的证书管理单元的导出/导入功能将颁发机构的证书 (C:\RootCA.cer) 导入受信任根证书颁发机构存储区中。但是不幸的是，由于 CA 证书是通过 MakeCert.exe 创建的，即使导入到受信任根证书颁发机构存储区，它也不能作为受信任的 CA。
- 通过 System.ServiceModel.Description.ClientCredentials 这个终结点行为改变默认的认证模式。

ClientCredentials 和之前提到的 ServiceCredentials 是两个相对的“行为”，我们来看如何通过 ClientCredentials 来改变客户端对服务证书的认证模式。可以通过 ChannelFactory<TChannel>的 Credentials 属性得到 ClientCredentials 对象。ClientCredential 具有一个类型为 System.ServiceModel.Security.X509CertificateRecipientClientCredential 的 ServiceCertificate 只读属性表示服务证书。

证书的认证行为定义在 X509CertificateRecipientClientCredential 的 Authentication 只读属性中。该属性的类型为 System.ServiceModel.Security.X509ServiceCertificateAuthentication。我们通过它的 CertificateValidationMode 属性设置相应的证书认证模式。关于服务证书认证模式涉及的应用编程接口反映在如下所示的代码片段中。

```
public abstract class ChannelFactory
{
    //其他成员
    public ClientCredentials Credentials { get; }
}
public class ClientCredentials : SecurityCredentialsManager, IEndpointBehavior
{
    //其他成员
    public X509CertificateRecipientClientCredential ServiceCertificate { get; }
}
public sealed class X509CertificateRecipientClientCredential
{
    //其他成员
    public X509ServiceCertificateAuthentication Authentication { get; }
}
public class X509ServiceCertificateAuthentication
{
    //其他成员
    public X509CertificateValidator CustomCertificateValidator { get; set; }
    public X509CertificateValidationMode CertificateValidationMode { get; set; }
}
```

证书认证模式通过枚举 System.ServiceModel.Security.X509CertificateValidationMode 表示，它具有 None、PeerTrust、ChainTrust、PeerOrChainTrust 和 Custom5 个选项。选择 None 意味着无须认证，而 ChainTrust 则要求证书的颁发机构必须是“受信任根证书颁发机构”存

储区, 而 PerTrust 要求证书本身 (不是 CA 证书) 存在于 “受信任的个人 (Trusted People)” 存储区。

如果这些认证模式不能满足需求, 还可以选择 Custom。在这种情况下, 需要通过继承抽象类 System.IdentityModel.Selectors.X509CertificateValidator 自定义验证规则, 具体的验证逻辑定义在 Validate 方法中。最终将自定义的 X509CertificateValidator 赋给 X509ServiceCertificateAuthentication 的 CustomCertificateValidator 属性。X509CertificateValidationMode 和 X509CertificateValidator 的定义如下。

```
public enum X509CertificateValidationMode
{
    None,
    PeerTrust,
    ChainTrust,
    PeerOrChainTrust,
    Custom
}

public abstract class X509CertificateValidator
{
    //其他成员
    public abstract void Validate(X509Certificate2 certificate);
}
```

对于本例来说, 我们创建的证书既不在 “受信任根证书颁发机构” 存储区, 也不在 “受信任的个人” 存储区。可以通过如下的代码选择 None 模式以避免异常的发生。

```
using (ChannelFactory<ICalculator> channelFactory = new
    ChannelFactory<ICalculator>("calculatorService"))
{
    channelFactory.Credentials.ServiceCertificate.Authentication.CertificateValidationMode = X509CertificateValidationMode.None;
    ICalculator calculator = channelFactory.CreateChannel();
    Console.WriteLine("x + y = {2} when x = {0} and y = {1}", 1, 2, calculator.Add(1, 2));
}
Console.Read();
```

也可以通过配置的方式来对 ClientCredentials 这个终结点行为进行相应的设置。通过上面这段程序对服务证书验证模式的设置与下面的这段配置在功能上是等效的。(S702)

```
<system.serviceModel>
...
<client>
  <endpoint behaviorConfiguration="IgreSvcCertValidation" .../>
</client>
<behaviors>
  <endpointBehaviors>
    <behavior name="IgreSvcCertValidation">
      <clientCredentials>
        <serviceCertificate>
          <authentication certificateValidationMode="None"/>
        </serviceCertificate>
      </clientCredentials>
    </behavior>
  </endpointBehaviors>
</behaviors>
```

```

    </behavior>
  </endpointBehaviors>
</behaviors>
</system.serviceModel>

```

(3) HTTPS (自我寄宿)

现在尝试使用 WS2007HttpBinding 实现基于 HTTPS 的 Transport 安全。对于基于 HTTP 的绑定, Transport 安全模式的实现方式又根据寄宿方式的不同而具有一定的差异, 下面首先来介绍自我寄宿的方式。(S703)

无论对于 HTTPS 还是 SSL via TCP, 服务端都需要绑定一个证书。对于后者, 是通过 ServiceCredentials 这个服务行为来进行证书设置的。但是对于 HTTPS, 需要通过相应的命令行工具将一个 X.509 证书绑定到相应的 HTTPS 端口。而这个工具的选择依赖于机器的操作系统, 对于 Windows XP 和 Windows 2003, 该命令行工具为 httpcfg.exe, 而对于之后的版本, 则是 netsh.exe。

httpcfg.exe 和 netsh.exe 均是通过证书的指纹 (thumbprint) 来关联具体证书的。我们之前已经通过 MakeCert.exe 命令行创建了一个以机器名作为主题名称的证书。可以通过 MMC 的证书管理单元来查看证书的指纹, 并将其复制到写字板上 (比如指纹为: afe0818b2bdb63aef9f1e96ea2b8d8b0f68bd3d9), 然后根据操作系统的不同分别执行 httpcfg.exe 和 netsh.exe 命令行, 为端口 3721 进行 SSL 证书的设置, 你可以将任意有效的 GUID 作为 appid 的参数。

```

httpcfg set ssl -i 0.0.0.0:3721 -h afe0818b2bdb63aef9f1e96ea2b8d8b0f68bd3d9
OR
netsh http add sslcert ipport=0.0.0.0:3721
certhash= afe0818b2bdb63aef9f1e96ea2b8d8b0f68bd3d9appid={CFA5621F-CD55-400
9-AD7E-51EDDAEC5786}

```

接下来我们将服务寄宿配置替换成如下的形式。从配置中可以看出, 寄宿服务的唯一终结点采用了被设置成 Transport 模式, 客户端凭证类型为 None 的 WS2007HttpBinding。地址被设置成 https://Jinan-PC:3721/calculatorService, 3721 正是上面我们进行 SSL 证书设置的端口。

```

<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="transportWS2007HttpBinding">
          <security mode="Transport">
            <transport clientCredentialType="None"/>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService">
        <endpoint address="https://Jinnan-PC:3721/calculatorService"
          binding="ws2007HttpBinding"

```

```

        bindingConfiguration="transportWS2007HttpBinding"
        contract="Artech.WcfServices.Service.Interface.
        ICalculator" />
    </service>
</services>
</system.serviceModel>
</configuration>

```

客户端的配置和编程同样进行相应的修改, 具有与服务端相同的绑定设置。具体的配置和服务调用程序如下所示。

```

<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="transportWS2007HttpBinding">
          <security mode="Transport">
            <transport clientCredentialType="None"/>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <client>
      <endpoint name="calculatorService"
        address="https://Jinnan-PC:3721/calculatorservice"
        binding="ws2007HttpBinding"
        bindingConfiguration="transportWS2007HttpBinding"
        contract="Artech.WcfServices.Service.Interface.
        ICalculator" />
    </client>
  </system.serviceModel>
</configuration>

```

然后运行程序, 会发现如图 7-14 所示的 SecurityNegotiationException 异常在服务调用过程中被抛出, 并提示“无法为 SSL/TLS 安全通道与颁发机构 ‘jinnan-pc:3721’ 建立信任关系”。导致异常抛出依然是证书不受信任所致, 因为 HTTPS 在默认的情况下依然采用 ChainTrust 认证模式。

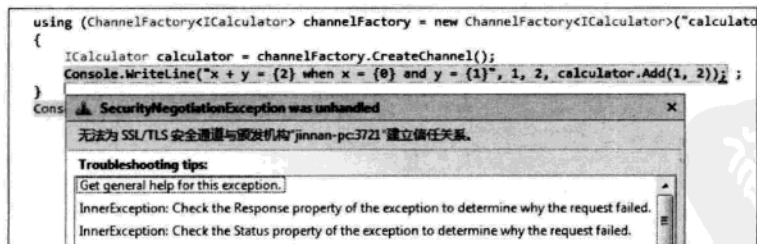


图 7-14 SSL 证书不受信任导致的异常

如果证书是官方机构颁发的, 我们可以将 CA 证书导入到“受信任根证书颁发机构”存储区来解决这个问题。但是对于我们这个测试程序来说, 就是改变客户端对 SSL 证书的认证方式。不过这与采用 NetTcpBinding 时通过终结点行为 ClientCredentials 来设置服务证书认

证模式的方式有所不同，我们需要通过注册 `System.Net.ServicePointManager` 的静态 `ServerCertificateValidationCallback` 回调自定义证书认证方式。`ServerCertificateValidationCallback` 回调在 `ServicePointManager` 中的定义如下面的代码片段所示。

```
public class ServicePointManager
{
    //其他成员
    public static RemoteCertificateValidationCallback
        ServerCertificateValidationCallback { get; set; }
}
public delegate bool RemoteCertificateValidationCallback(object sender, X509
Certificate certificate, X509Chain chain, SslPolicyErrors sslPolicyErrors);
```

在下面给出的代码片段中，我通过在进行服务调用之前注册 `ServerCertificateValidationCallback` 回调的方式来忽略对 SSL 证书的认证，就可以抑制 `SecurityNegotiationException` 异常的抛出。（S704）

```
ServicePointManager.ServerCertificateValidationCallback+=
    (sender,certificate, chain, sslPolicyErrors)=>true;
using (ChannelFactory<ICalculator> channelFactory = new
    ChannelFactory<ICalculator>("calculatorService"))
{
    ICalculator calculator = channelFactory.CreateChannel();
    Console.WriteLine("x + y = {2} when x = {0} and y = {1}", 1, 2,
        calculator.Add(1, 2)); ;
}
```

对于 SSL 证书的认证，还有一点需要说明。在默认的情况下，客户端除了采用 `ChainTrust` 模式对证书进行认证之外，还具有一个认证规则，那就是要求终结点地址的 DNS 和证书的主体名称相匹配。以我们创建的这个程序为例，现在将客户端配置文件中的终结点地址从 `https://Jinnan-PC:3721/calculatorservice` 替换成 `https://localhost:3721/calculatorservice`，在进行服务调用的时候会抛出如图 7-14 所示的 `SecurityNegotiationException` 异常。

（3）HTTPS（IIS/WAS 寄宿）

采用 Transport 安全模式的服务需要寄宿在一个 SSL 站点下。如果你的 IIS 中没有一个 SSL 站点，需要手工创建。下面先来演示如何在你的 IIS 中创建一个 SSL 站点，这里以 IIS 7.5 为例。

当开启了 IIS 管理器之后，通过点击左侧边栏的“应用程序池（Application Pool）”节点打开应用程序池列表界面。然后右击弹出上下文菜单，并选择“添加应用程序池（Add Application Pool）”菜单项，打开添加应用程序池对话框。设置添加的应用程序池的名称（比如 `DefaultSSLAppPool`），选择 .NET 版本（.NET Framework 4.0.30319）和托管管道模式（Integrated）。确认后，新的应用程序池被创建出来。

接下来需要创建使用这个应用程序池的 SSL 站点。右击“站点（Sites）”节点，选择“添加 Web 站点（Add Web Site）”菜单项，进入添加 Web 站点对话框。在该对话框中设置新建站点的名称（比如 `Default SSL Web Site`），选择刚刚创建的应用程序池（`DefaultSSLAppPool`），

并为站点设置一个本地的物理路径。在绑定类型列表中选择 https, 保持 IP 地址和端口的默认值。在 SSL 证书下拉框中会看到我们之前创建的证书 (Jinnan-PC), 选择它作为站点的 SSL 证书。单击确认按钮后, 新的站点被创建出来了。

SSL 站点被成功创建之后, 需要在该站点中创建一个应用程序, 取名为 WcfServices, 并将物理地址映射为解决方案中 Services 项目的根目录。然后需要对 Services 项目的“生成 (Build)”进行相应的设置, 将编译后的目标目录从默认的“\bin\debug”改成“\bin”, 以确保项目编译后的程序集被生成到 bin 目录下。

IIS 寄宿需要为服务创建相应的.svc 文件, 此时直接在 Services 项目中添加一个文本文件, 取名为 CalculatorService.svc, 并为其添加如下的内容。

```
<%@ ServiceHost Service="Artech.WcfServices.Services.CalculatorService" %>
```

在 Services 项目中添加一个 Web.config 文件, 定义如下一段服务寄宿的配置。在这段配置中, 除了无须指定终结点地址之外, 其他所有的配置与通过自我寄宿方式别无二致。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="transportWS2007HttpBinding">
          <security mode="Transport">
            <transport clientCredentialType="None"/>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService">
        <endpoint binding="ws2007HttpBinding"
          bindingConfiguration="transportWS2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.
            ICalculator"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

只需要在 Client 项目中的配置中修改一下终结点地址 (https://Jinnan-PC/WcfServices/CalculatorService.svc) 就能直接执行程序。

在采用 TLS/SSL 实现 Transport 安全的情况下, 客户端对服务证书实施认证。但是在默认情况下, 这种认证仅确保服务证书的合法性 (通过数字签名确保证书确实是由声明的 CA 颁发的) 和可信性 (证书或者 CA 证书存储于相应的可信赖存储区)。而 WCF 提供服务证书并不限于此, 客户端对服务认证的模式应该是这样的: 服务端预先知道了服务的身份, 在进行服务调用之前, 服务端需要提供相应的凭证用于辅助客户端确认调用的服务具有预先确定的身份。对于这样的服务认证模式, 具有两个重要的概念, 即服务凭证和服务身份。

3. 服务凭证 (Service Credential)

认证就是通过对被认证方提供的凭证进行检验以确定对方身份的一个过程, 服务认证和客户端认证并没有本质的区别。但服务认证确实有一点和客户端认证不同, 那就是客户端在对服务进行认证之前就预先确定了服务应当具有的身份。而在真正进行服务调用的时候, 客户端要求服务提供相应的凭证与预先确定的身份进行比较, 从而确定当前正在调用的服务正是自己希望调用的那一个。

客户端具有多种形式的凭证类型, 但是服务凭证只具有 Windows 凭证和 X.509 证书两种类型。Windows 认证具有 Kerberos 和 NTLM 两种具体的实现, 但是只有 Kerberos 支持双向认证。因此, 只有在基于域 (Domain) 的网络环境中, 基于 Windows 认证的服务认证才是可行的。而在工作组 (Work Group) 环境中, 推荐使用基于证书的服务认证。

服务认证方式的选择决定于客户端认证采用的方式。如果采用 Windows 认证的方式对客户端实施认证, 服务认证同样采用 Windows 认证。基于 X.509 证书的认证在非 Windows 客户端认证下被采用。如果客户端凭证类型为 Windows, 那么 WCF 采用执行服务寄宿进程的 Windows 账号对应的 Windows 凭证作为服务凭证。如果其他非 Windows 凭证作为客户端凭证, 则必须为服务显式地指定一个 X.509 证书作为服务凭证。

在 WCF 的应用编程接口中, 具有一个重要的服务行为 `ServiceCredentials`。这个类并不简单地像它的名称所表示的那样用于进行服务凭证的设置, 实际上需要在服务端执行的很多认证、授权行为都是通过 `ServiceCredentials` 来实现的。

如果服务采用基于 X.509 证书作为服务凭证, 客户端对服务的认证过程实际上分为两个阶段。第一个阶段是验证证书的合法性, 在默认的情况下会采用 `ChainTrust` 认证模式, 不过可以通过终结点行为 `ClientCredentials` 来设置不同的认证模式。然后通过比较服务证书和事先确立的服务身份信息信息进行对照, 进而确定服务是否是客户端试图访问的服务。

4. 服务身份 (Service Identity)

我们知道终结点是 WCF 最为核心的概念, 终结点通过类型 `ServiceEndpoint` 表示。终结点具有 ABC 三要素分别表示地址、绑定和契约, 其中地址通过 `EndpointAddress` 表示。如果你对 `EndpointAddress` 有一定的了解, 应该清楚该类具有如下一个只读的类型为 `System.ServiceModel.EndpointIdentity` 的 `Identity` 的属性。

```
public class ServiceEndpoint
{
    //其他成员
    public EndpointAddress Address { get; set; }
}
public class EndpointAddress
{
    //其他成员
    public EndpointIdentity Identity { get; }
```

我们通常所说的“调用某个服务”实际上应该是“调用服务的某个终结点”。服务身份实际上也应该是“终结点身份”。通过 `ServiceEndpoint` 对象表示的终结点的身份通过 `Address` 的 `Identity` 属性来表示，而该属性的类型就是具有如下定义的 `EndpointIdentity`。

```
public abstract class EndpointIdentity
{
    //其他成员
    public static EndpointIdentity CreateSpnIdentity(string spnName);
    public static EndpointIdentity CreateUpnIdentity(string upnName);
    public static EndpointIdentity CreateRsaIdentity(X509Certificate2 certificate);
    public static EndpointIdentity CreateRsaIdentity(string publicKey);
    public static EndpointIdentity CreateX509CertificateIdentity(X509
    Certificate2
    certificate);
    public static EndpointIdentity CreateDnsIdentity(string dnsName);

    public Claim IdentityClaim { get; }
}
public class SpnEndpointIdentity : EndpointIdentity
{
    //省略成员
}
public class UpnEndpointIdentity : EndpointIdentity
{
    //省略成员
}
public class DnsEndpointIdentity : EndpointIdentity
{
    //省略成员
}
public class RsaEndpointIdentity : EndpointIdentity
{
    //省略成员
}
public class X509CertificateEndpointIdentity : EndpointIdentity
{
    //省略成员
}
```

服务身份声明通过属性 `IdentityClaim` 表示，这些信息是为最终的认证服务的。从上面的代码可以看出，`EndpointIdentity` 实际上是一个抽象类，它具有如下几个常用的子类（`SpnEndpointIdentity`、`UpnEndpointIdentity`、`X509CertificateEndpointIdentity`、`RsaEndpointIdentity`、`DnsEndpointIdentity`），分别表示不同的服务身份类型。这些具体的 `EndpointIdentity` 可以通过对应的静态方法 `CreateXxxIdentity` 创建。

`SpnEndpointIdentity` 和 `UpnEndpointIdentity` 是 Windows 认证下服务身份的两种表现形式。前者被称为服务主体名（SPN，Service Principal Name），后者被称为用户主体名（UPN，User Principal Name）。

如果你对 Kerberos 有一定的了解，相信一定不会对 SPN 感到陌生。对于一个运行在域环境中某台机器上的服务，它能被访问它的客户端认证的先决条件是客户端能够唯一标识该

服务, 而 SPN 就可以看做是这个标识符。在默认情况下, 如果服务寄宿进程在机器账号 (或者系统账号, 比如 LocalService、LocalSystem 或者 NetworkService 等) 下, 则服务身份采用 SPN 表示。如果执行服务寄宿进程的是一个域用户账户, 则采用 UPN 表示服务身份。

WCF 中的 SPN 和 UPN 的格式如下。如果客户端预先指定 SPN/UPN 表示服务身份, 它通过执行服务寄宿进程账号对应的 Windows 凭证和 SPN/UPN 进行比较, 从而确定服务运行在预先设定的机器或者某个域用户账号下。

```
SPN: Host/(<<HostName>> (Host/artech-win7-x64)
UPN: <<DomainName>>/<<UserName>> (Microsoft/BillGates) 或者
      <<UserName>>@<<DomainName>> (BillGates@Microsoft)
```

如果采用 X.509 证书作为服务凭证, 服务身份可以通过 X509CertificateEndpointIdentity 和 RsaEndpointIdentity 表示。而 X509CertificateEndpointIdentity 既可以直接采用 X.509 证书中的指纹作为服务身份标识, 也可以采用存储区中某个证书的引用来表示。而 RsaEndpointIdentity 则将 X.509 证书的 RSA 密钥作为服务身份标识。如果客户端预先指定了相应的 X509CertificateEndpointIdentity/RsaEndpointIdentity 作为服务身份, 它会通过将作为服务凭证的 X.509 证书与此进行比较, 进而确定服务是相应证书的真正拥有者。

DnsEndpointIdentity 是基于 DNS 名称的服务身份。如果采用 X.509 证书作为服务凭证, 并且这个证书的主题名称是一个 DNS 名称, 客户端可以采用 DnsEndpointIdentity 来对服务证书进行认证。在基于 SPN 的 Windows 认证下, 如果 SPN 基于一个 DNS, 客户端也可以采用 DnsEndpointIdentity 认证服务。

服务端和客户端的终结点都可以设置这个表示服务身份标识的 EndpointIdentity。不过设置的 EndpointIdentity 对于服务端和客户端终结点具有不同的作用。服务端终结点设置的 EndpointIdentity 用于元数据发布, 客户端终结点设置 EndpointIdentity 最终用于对服务的认证。

一般情况下, 服务终结点的 EndpointIdentity 无须指定, 因为 WCF 会根据绑定采用的客户端凭证类型和寄宿进程运行的 Windows 账号生成相应的 EndpointIdentity。终结点的 EndpointIdentity 最终会成为元数据的一部分被写入服务的 WSDL 中。

比如我们采用 IIS 的方式寄宿服务, 终结点采用 Transport 模式的 WS2007HttpBinding, EndpointIdentity 对应在 WSDL 部分的内容将会如下面的 XML 片段所示。由于 IIS (IIS 6 或之后版本) 在 Network Service 账号下执行, 所以默认会使用 SPN 作为服务身份标识 (SPN 中的 Jinnan-Win7-X64 为机器名称)。

```
<wsdl:definitions name="CalculatorService" targetNamespace="http://tempuri.org/">
  ...
  <wsdl:service name="CalculatorService">
    <wsdl:port name="WS2007HttpBinding_ICalculator"
              binding="tns:WS2007HttpBinding_ICalculator">
      <soap12:address
        location="https://jinnan-win7-x86/WcfServices/CalculatorService.svc"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

```

<wsa10:EndpointReference>
  <wsa10:Address>
    https://jinnan-win7-x64/WcfServices/CalculatorService.svc
  </wsa10:Address>
  <Identity>
    <Spn>host/Jinnan-Win7-X64</Spn>
  </Identity>
</wsa10:EndpointReference>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>

```

客户端通过添加服务引用或者直接使用 SvcUtil.exe 导入元数据生成客户端代码和配置的时候, WSDL 中的服务身份标识会自动被写入配置中。上述 6 种不同形式的 EndpointIdentity 在配置中的表示如下面的 XML 片段所示。

```

<system.serviceModel>
  <client>
    <endpoint address="http://jinnan-win7-x86/calculatorservice1"
      binding="ws2007HttpBinding" contract="Artech.WcfServices.Contracts.ICalculator">
      <identity>
        <userPrincipalName value="jinnan@contoso.com"/>
        <servicePrincipalName value="host/jinnan-win7-x86"/>
        <certificate encodedValue="f332bf17db3abb8f9a9a2694ba2c75da701bef0f"/>
        <certificateReference storeLocation="LocalMachine"
          storeName="My"
          x509FindType="FindBySubjectName"
          findValue="jinnan-win7-x86"/>
        <rsa value="sdhjgr...djakjhg"/>
        <dns value="jinnan-win7-x86"/>
      </identity>
    </endpoint>
  </client>
</system.serviceModel>

```

5. 服务身份认证与服务凭证协商

用于调用服务的客户端终结点最终都关联着一个 EndpointIdentity, 它代表了客户端希望的服务身份。客户端在正式向服务发送功能性消息之前, 会根据服务端提供的服务凭证和这个 EndpointIdentity 对服务实施认证。如果服务凭证与客户端持有的服务身份相一致, 则认证成功并开始后续的消息交换, 否则双方之间的交互到此为止。

在默认情况下, 客户端和服务端有一个“协商 (Negotiation)”的过程。客户端通过此协商过程从服务端获取服务凭证, 所以我们将这个协商机制称为“服务凭证协商 (Service Credentials Negotiation)”。对于 Transport 安全模式, 服务凭证协商过程是必需的。但是对于 Message 安全模式, 可以通过相应的设置避免服务凭证协商。

如果服务凭证不能通过协商的方式传递给客户端, 那么必然要通过另外的方式预先提供给它。对于 Windows 认证, 需要客户端和服务端必须处于同一域中。而对基于 X.509 证书的服务凭证, 需要预先将其安装到客户端。抑制服务凭证协商会因避免证书的传递而对安全性有所增强, 但是也会因为需要额外的证书递交机制而带来额外的负担。

对于所有支持 Message 模式的 WS 的绑定 (WSHttpBinding、WS2007HttpBinding 和 WSDualHttpBinding) 来说, 开启和关闭服务凭证协商可以通过设置 MessageSecurityOverHttp 类型的 NegotiateServiceCredential 属性来实现。

```
public class MessageSecurityOverHttp
{
    //其他成员
    public bool NegotiateServiceCredential {get; set; }
}
```

可以按照如下的配置对 WSHttpBinding、WS2007HttpBinding 和 WSDualHttpBinding 这三种绑定是否支持服务凭证协商进行控制。在绑定配置节下的<security>/<message>节点中, 可以找到开启或者关闭服务凭证协商的 negotiateServiceCredential 配置属性。

```
<system.serviceModel>
  <bindings>
    <ws2007HttpBinding>
      <binding name="transportWS2007HttpBinding">
        <security mode="Message">
          <message negotiateServiceCredential="false".../>
        </security>
      </binding>
    </ws2007HttpBinding>
  </bindings>
  ...
</system.serviceModel>
```

7.2.4 客户端认证

客户端认证采用的方式决定于客户端凭证的类型, 接下来的内容只涉及 Windows、用户名和 X.509 证书三种典型客户凭证类型的认证。

1. Windows 认证

从编程的角度来讲, Windows 认证是最为简单的认证方式。在这种认证方式下, 客户端进程运行的 Window 账号对应的 Windows 凭证被自动作为调用服务的客户端凭证, 所以无须显式指定具体的 Windows 凭证。

如果需要以另一个 Windows 账号的名义调用服务, 客户端就需要通过指定 Windows 账号和密码的方式显式地进行客户端 Windows 凭证的设置。Windows 凭证通过具有如下定义的 System.ServiceModel.Security.WindowsClientCredential 类型表示, 真正的凭证最终保存在类型为 NetworkCredential 的 ClientCredential 属性中。通过该属性可以指定 Windows 凭证的域名、用户名和密码。

```
public sealed class WindowsClientCredential
{
    //其他成员
```

```

    public bool AllowNtlm { get; set; }
    public NetworkCredential ClientCredential { get; set; }
}
public class NetworkCredential : ICredentials, ICredentialsByHost
{
    //其他成员
    public string Domain { get; set; }
    public string UserName { get; set; }
    public string Password { get; set; }
    public SecureString SecurePassword { get; set; }
}

```

从上面给出的代码可以看到, `NetworkCredential` 除了一个以 `String` 类型表示的 `Password` 属性之外, 还有另外一个相关类型为 `System.Security.SecureString` 的 `SecurePassword` 属性。

`String` 类型具有恒定性 (Immutability), 一旦被创建, 它将在整个进程生命周期内一直存在。如果某个 `String` 对象含有像密码这样的敏感信息, 会因应用程序无法从计算机内存中删除这些数据, 而存在信息在使用后可能被泄露的危险。`SecureString` 对象与 `String` 对象的相似之处在于它也具有文本值。但 `SecureString` 对象的值是自动加密的, 在应用程序将它标记为只读之前可以进行修改, 并且可由应用程序或 .NET Framework 垃圾回收器将其从计算机内存中删除。

当进行服务调用的时候, 不管你是直接采用 `ChannelFactory<TChannel>` 创建服务代理的方式, 还是通过导入元数据生成客户端代理的方式, 设置 Windows 凭证都很容易。`ChannelFactory<TChannel>` 的基类 `ChannelFactory` 和 `ClientBase<TChannel>` 中都定义了一个只读的 `ClientCredentials` 属性, 该属性的类型为 `ClientCredentials`。对于类型 `ClientCredentials`, 我们应该不会感到陌生, 因为在前面的实例演示中我们通过它实现了对服务证书认证模式的改变。表示 Windows 凭证的 `WindowsClientCredential` 对象作为只读属性 `Windows` 定义在 `ClientCredentials` 中, 相关类型定义在如下的代码片段中。

```

public class ChannelFactory<TChannel> : ChannelFactory
{
    //省略成员
}
public abstract class ChannelFactory
{
    //其他成员
    public ClientCredentials Credentials { get; }
}
public abstract class ClientBase<TChannel>
{
    //其他成员
    public ClientCredentials ClientCredentials { get; }
}
public class ClientCredentials : SecurityCredentialsManager, IEndpointBehavior
{
    //其他成员
    public WindowsClientCredential Windows { get; }
}

```

下面给出的代码片段演示了当采用通过 `ChannelFactory<TChannel>` 创建的服务代理进行服务调用时如何进行 Windows 凭证的设置。

```
using (ChannelFactory<ICalculator> channelFactory = new
    ChannelFactory<ICalculator>("calculatorService"))
{
    NetworkCredential credential =
        channelFactory.Credentials.Windows.ClientCredential;
    credential.Domain = "DomainName";
    credential.UserName = "UserName";
    credential.Password = "Password";

    ICalculator calculator = channelFactory.CreateChannel();
    double result = calculator.Add(1, 2);
    //...
}
```

`WindowsClientCredential` 具有一个布尔类型的属性 `AllowNtlm`。这个属性实际上涉及 Windows 认证协议的问题。WCF 集成的 Windows 认证基于 SSPI (Security Support Provider Interface), 这是一套标准的安全编程接口, 而具体安全功能的实现定义在相应的 SSP (Security Support Provider) 中。

Windows 提供了 Kerberos、NTLMSSP 和 SPNEGO 三种典型的 SSP。前两种分别基于我们熟悉的 Kerberos 和 NTLM, 但是 SPNEGO 才是默认的选项。SPNEGO 的全名为 “Simple and Protected GSSAPI Negotiation Mechanism”, 而 GSSAPI (Generic Security Services Application Program Interface) 是互联网工程任务组 (IETF) 指定的安全应用编程接口。SPNEGO, 顾名思义, 就是通过协商 (Negotiation) 确定一种适合的 GSS API。SPNEGO 在 Windows 下的协商机制是这样的: 首选 Kerberos, 如果不可用则退而求其次, 选用 NTLM。

不论是安全性还是互操作性 (实际上 Kerberos 本身就是一种标准), Kerberos 都要优于 NTLM, 但是 Kerberos 仅限于基于 AD 的域环境中使用。如果强制要求只采用 Kerberos 认证, 可以通过将 `WindowsClientCredential` 的 `AllowNtlm` 属性设成 `False` 来实现。

2. 用户名/密码认证

对于基于 Internet 的应用, 用户名和密码认证方式是最为常用的, 而 WCF 提供了不同模式的用户名认证方式。基于用户名/密码的用户凭证通过类型 `System.ServiceModel.Security.UserNamePasswordClientCredential` 表示。而在 `ClientCredentials` 中, 只读属性 `UserName` 表示这样一个用户凭证。可以按照 Windows 凭证的方式为 `ChannelFactory<TChannel>` 或者 `ClientBase<TChannel>` 设置用户名/密码凭证。

```
public class ClientCredentials
{
    //其他成员
    public UserNamePasswordClientCredential UserName { get; }
}
public sealed class UserNamePasswordClientCredential
```

```
{
    //其他成员
    public string Password {get; set; }
    public string UserName { get; set; }
}
```

用户名/密码凭证在客户端的设置很容易,但是我们关心的是服务端采用怎样的机制来验证这个凭证。WCF 提供了如下三种方式来验证凭证中的用户名是否和密码相符。

- **Windows:** 将用户名和密码映射为 Windows 账号和密码,采用 Windows 认证。
- **MembershipProvider:** 利用配置的 ASP.NET MembershipProvider 验证用户名和密码。
- **自定义:** 通过继承抽象类 UsernamePasswordValidator, 自定义用户名/密码验证器进行验证。

WCF 通过枚举具有如下定义的 System.ServiceModel.Security.UserNamePasswordValidationMode 枚举表示上述三种用户名/密码认证模式,其中 Windows 是默认选项。

```
public enum UserNamePasswordValidationMode
{
    Windows,
    MembershipProvider,
    Custom
}
```

上述三种认证模式是通过 ServiceCredentials 这一服务行为进行设置的。从下面的定义可以看出, ServiceCredentials 定义了类型为 System.ServiceModel.Security.UserNamePasswordServiceCredential 的只读属性 UserNameAuthentication 进行基于用户名/密码认证的相关设置。UserNamePasswordServiceCredential 的 UserNamePasswordValidationMode 属性表示采用的认证模式。如果选择了 MembershipProvider 模式,则需要通过 MembershipProvider 属性设置具体实现认证的 MembershipProvider 对象。如果选择了 Custom,则需要通过 CustomUserNamePasswordValidator 属性指定自定义的 UserNamePasswordValidator 对象。

```
public class ServiceCredentials: SecurityCredentialsManager,
IServiceBehavior
{
    //其他成员
    public UserNamePasswordServiceCredential UserNameAuthentication { get; }
}
public sealed class UserNamePasswordServiceCredential
{
    //其他成员
    public UserNamePasswordValidator CustomUserNamePasswordValidator { get;
set; }
    public MembershipProvider MembershipProvider { get; set; }
    public UserNamePasswordValidationMode UserNamePasswordValidationMode
    { get; set; }
}
```

接下来通过实例演示的方式来看如何通过 MembershipProvider 进行基于用户名/密码认证,而对于自定义 UserNamePasswordValidator 的实例演示可以在 7.3 节中找到。

3. 实例演示：通过 MembershipProvider 进行用户名/密码的认证（S706）

Membership 是 ASP.NET 中的一个重要的模块，旨在进行基于用户名/密码的认证和账号管理。Membership 采用策略设计模式，所有的 API 通过静态 Membership 类暴露出来，而相应的功能实现在具体的 MembershipProvider 中。所有的 MembershipProvider 继承自同一个抽象类 System.Web.Security.MembershipProvider。ASP.NET 提供了 SqlMembershipProvider 和 ActiveDirectoryMembershipProvider 两种提供者，前者将用户存储于 SQL Server 数据库中，而后者则直接建立在 AD 之上。本实例采用 SqlMembershipProvider，在前面的实例演示中，我们创建了以计算服务为场景的解决方案，现在我们直接沿用它。

我们的首要任务是创建用于存储账户信息的 SQL Server 数据库，为此我们可以在本地 SQL Server 创建一个空的数据库（假设起名为 AspNetDb）。然后在该数据库中创建 SqlMembershipProvider 所需的数据表和相应的存储过程。这些数据库对象的创建，需要借助于 aspnet_regsql.exe 这个工具。只需要以命令行的方式执行如下 aspnet_regsql.exe（无须任何参数），相应的向导就会出现。在向导弹出的前两个窗体中保持默认设置，直接单击“下一步”后，会出现一个数据库选择窗体。此时需要选择我们刚刚创建的数据库，单击“确认”后，相关的数据库对象就会创建出来。

这些创建出来的数据表可以同时服务于多个应用，所以每一个表中都具有一个名称为 ApplicationId 的字段来明确该条记录对应的应用。所有应用记录维护在 aspnet_Applications 这样一个表中。现在我们需要通过执行下面一段 SQL 脚本在该表中添加一条表示本实例应用的记录。我们将应用取名为 MembershipAuthenticationDemo。

```
INSERT INTO [aspnet_Applications]
    ([ApplicationName]
    , [LoweredApplicationName]
    , [ApplicationId]
    , [Description])
VALUES
    (
        'MembershipAuthenticationDemo'
        , 'membershipauthenticationdemo'
        , NEWID()
        , ''
    )
```

现在数据库方面已经准备就绪，接着来完成编程和配置方面的工作。我们采用自我寄宿的方式，由于 Membership 隶属于 ASP.NET，所以需要添加 System.Web.dll 的引用，如果采用的是 .NET Framework 4.0（本例所示的配置也是基于该版本），还需额外添加对 System.Web.ApplicationServices.dll 的引用。下面是服务寄宿程序的配置。

```
<configuration>
  <connectionStrings>
    <add name="AspNetDb"
          connectionString="Server=.; Database=AspNetDb; Uid=sa;
          Pwd=password"/>
  </connectionStrings>
```

```

<system.web>
  <membership defaultProvider="myProvider">
    <providers>
      <add name="myProvider"
        type="System.Web.Security.SqlMembershipProvider, System.Web"
        connectionStringName="AspNetDb"
        applicationName="MembershipAuthenticationDemo"
        requiresQuestionAndAnswer="false" />
    </providers>
  </membership>
</system.web>
<system.serviceModel>
  <bindings>
    <ws2007HttpBinding>
      <binding name="userNameCredentialBinding">
        <security mode="Message">
          <message clientCredentialType="UserName" />
        </security>
      </binding>
    </ws2007HttpBinding>
  </bindings>
  <services>
    <service name="Artech.WcfService.Services.CalculatorService"
      behaviorConfiguration="membershipAuthentication">
      <endpoint address="http://127.0.0.1/calculatorservice"
        binding="ws2007HttpBinding"
        bindingConfiguration="userNameCredentialBinding"
        contract="Artech.WcfServices.Service.Interface.ICalculator" />
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="membershipAuthentication">
        <serviceCredentials>
          <serviceCertificateStoreLocation="LocalMachine"
            storeName="My"
            x509FindType="FindBySubjectName"
            findValue="Jinnan-PC" />
          <userNameAuthentication
            userNamePasswordValidationMode="MembershipProvider"
            membershipProviderName="myProvider" />
        </serviceCredentials>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
</configuration>

```

考虑到有些人可能对 ASP.NET 下的 Membership 相关配置不太了解, 在这里对上述这段配置进行简单的说明。

- 配置名称为 AspNetDb 的连接字符串连接的是我们刚刚创建的数据库。
- 表示 Membership 配置节的 <system.web>/<membership> 节点下配置了一个名称为 myProvider 的 SqlMembershipProvider。上面配置的连接字符串名称 AspNetDb 配置在 connectionStringName 属性中, 意味着该 SqlMembershipProvider 会将我们创建的数据库

作为用户账号存储。该 `SqlMembershipProvider` 被设置为默认的 `MembershipProvider` (`defaultProvider="myProvider"`)。

- 服务终结点采用的是 `WS2007HttpBinding`，采用 `Message` 安全模式，客户端凭证类型被设置为 `UserName`。
- 服务应用了一个配置名称为 `membershipAuthentication` 的服务行为，该行为中通过 `<serviceCertificate>` 节点设置了服务证书。在表示用户名/密码认证配置的 `<userNameAuthentication>` 节点中，将认证模式设置成 `MembershipProvider`，而 `membershipProviderName` 属性的值 `myProvider` 为我们在 `<system.web>/<membership>` 中设置的 `MembershipProvider` 的名称。

到目前为止，在我们创建的数据库中并没有用户账户记录。为了演示认证的效果，必须创建相关用户账户记录。为了省事我们直接将相关的代码写在了服务寄宿的代码中。如下面的代码片段所示，在对服务进行寄宿之前，我们通过调用 `Membership` 的静态方法 `CreateUser` 创建了一个用户名、密码和 E-mail 分别为 `Zhansan`、`Pass@word` 和 `zhanshan@gmail.com` 的账号。

```
if (Membership.FindUsersByName("Zhansan").Count == 0)
{
    Membership.CreateUser("Zhansan", "Pass@word", "zhanshan@gmail.com");
}
using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    host.Open();
    Console.Read();
}
```

接下来需要对客户端的配置进行相应的调整，整个配置内容如下面的 XML 片段所示。对于这段配置有一点需要注意：终结点应用了一个名称为 `peerTrustSvcCertValidation` 的行为，该行为中将服务证书认证模式设置成 `PeerTrust`，所以需要通过 MMC 证书管理单元的导出/导入功能将表示服务凭证的证书导入到“受信任人 (Trusted People)”存储区。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="userNameCredentialBinding">
          <security mode="Message">
            <message clientCredentialType="UserName"/>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <client>
      <endpoint name="calculatorService"
        behaviorConfiguration="peerTrustSvcCertValidation"
        address="http://127.0.0.1/calculatorService"
        binding="ws2007HttpBinding"
        bindingConfiguration="userNameCredentialBinding"
        contract="CalculatorService" />
    </client>
  </system.serviceModel>
</configuration>
```

```

        contract="Artech.WcfServices.Service.Interface.ICalculator">
    <identity>
        <certificateReference      storeLocation="LocalMachine"
                                storeName = "My"
                                x509FindType="FindBySubjectName"
                                findValue="Jinnan-PC"/>
    </identity>
</endpoint>
</client>
<behaviors>
    <endpointBehaviors>
        <behavior name="peerTrustSvcCertValidation">
            <clientCredentials>
                <serviceCertificate>
                    <authentication certificateValidationMode="PeerTrust"/>
                </serviceCertificate>
            </clientCredentials>
        </behavior>
    </endpointBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

最后我们来编写如下一段客户端进行服务调用的程序。在下面的代码中,我进行了两次服务调用。但是创建服务代理对象的 `ChannelFactory<ICalculator>` 被设置了不同的用户名凭证。其中第一个是正确的用户名和密码,第二个却指定了一个根本不存在的用户名。从输出结果可以看出,第一次服务调用成功,第二次失败。

```

using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService"))
{
    UserNamePasswordClientCredential credential =
        channelFactory.Credentials.UserName;
    credential.UserName      = "Zhansan";
    credential.Password      = "Pass@word";
    ICalculator calculator   = channelFactory.CreateChannel();
    calculator.Add(1, 2);
    Console.WriteLine("服务调用成功...");
}
using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService"))
{
    UserNamePasswordClientCredential credential =
        channelFactory.Credentials.UserName;
    credential.UserName      = "lisi";
    credential.Password      = "Pass@word";
    ICalculator calculator   = channelFactory.CreateChannel();
    try
    {
        calculator.Add(1, 2);
    }
    catch
    {
        Console.WriteLine("服务调用失败...");
    }
}

```

输出结果:

```
服务调用成功...
服务调用失败...
```

4. 证书认证

最后介绍客户端凭证类型为 X.509 证书时服务端采用的认证, 简称为证书认证。照例先看看基于 X.509 证书的客户端如何定义。

(1) 客户端证书凭证的设置

在服务认证一节中, 我们知道了基于 X.509 证书的服务凭证通过 `X509CertificateRecipientServiceCredential` 类型表示。与之对应地, 客户端凭证对应的类型是 `X509CertificateInitiatorClientCredential`。如下面的定义所示, 在终结点行为 `ClientCredentials` 中具有一个只读的 `ClientCertificate` 属性, 其类型就是 `X509CertificateInitiatorClientCredential`。该类型实际上是对一个 `X509Certificate2` 类型对象的封装, 可以通过两个 `SetCertificate` 方法重载以证书引用的方式指定某个具体的 X.509 证书作为客户端的凭证。

```
public class ClientCredentials : SecurityCredentialsManager,
    IEndpointBehavior
{
    //其他成员
    public X509CertificateInitiatorClientCredential ClientCertificate { get; }
}
public sealed class X509CertificateInitiatorClientCredential
{
    //其他成员
    public void SetCertificate(string subjectName, StoreLocation storeLocation,
        StoreName storeName);
    public void SetCertificate(StoreLocation storeLocation, StoreName storeName,
        X509FindType findType, object findValue);
    public X509Certificate2 Certificate { get; set; }
}
```

在下面给出的服务调用代码中, 我们通过编程的方式为 `ChannelFactory<TChannel>` 设置了一个基于 X.509 证书的客户端凭证。

```
using (ChannelFactory<ICalculator> channelFactory = new
    ChannelFactory<ICalculator>("calculatorService"))
{
    channelFactory.Credentials.ClientCertificate.SetCertificate(
        StoreLocation.LocalMachine,
        StoreName.TrustedPeople,
        X509FindType.FindBySubjectName,
        " Foo");
    ICalculator calculator = channelFactory.CreateChannel();
    double result = calculator.Add(1, 2);
    //...
}
```

终结点行为 `ClientCredentials` 同样为客户端 (默认) 证书的设置定义相应的配置。在下

面给出的 XML 片段中, 我们通过配置的方式为终结点凭证指定了一个 X.509 证书。这个证书将作为服务代理对象的默认客户凭证, 该凭证可以通过编程进行动态更改。

```
<system.serviceModel>
...
<behaviors>
  <endpointBehaviors>
    <behavior name="defaultClientCert">
      <clientCredentials>
        <clientCertificate storeLocation="LocalMachine"
          storeName="My"
          x509FindType="FindBySubjectName"
          findValue="Foo"/>
      </clientCredentials>
    </behavior>
  </endpointBehaviors>
</behaviors>
</system.serviceModel>
```

(2) 服务端对证书的验证

对于服务认证, 服务在寄宿的时候指定某个 X.509 证书作为服务的凭证。客户端在默认的情况下会以 ChainTrust 模式对服务证书进行认证, 也可以通过 ClientCredentials 这个终结点行为指定不同的认证模式。对于客户端认证, 服务端对客户证书的认证也采用相同的策略, 即默认认证模式为 ChainTrust, 但借助 ServiceCredentials 这个服务行为来指定不同的认证模式。

服务行为 ServiceCredentials 中具有一个只读的 ClientCertificate 属性, 类型为 X509CertificateInitiatorServiceCredential。X509CertificateInitiatorServiceCredential 定义了类型为 X509ClientCertificateAuthentication 的只读属性 Authentication。我们可以通过 CertificateValidationMode 属性设置不同的认证模式 (None、PeerTrust、ChainTrust、PeerOrChainTrust、Custom)。当选择 Custom 模式的时候, 需要通过 CustomCertificateValidator 属性设置一个自定义的 X509CertificateValidator。

```
public class ServiceCredentials : SecurityCredentialsManager,
IServiceBehavior
{
    //其他成员
    public X509CertificateInitiatorServiceCredential ClientCertificate { get; }
}
public sealed class X509CertificateInitiatorServiceCredential
{
    //其他成员
    public X509ClientCertificateAuthentication Authentication { get; }
    public X509Certificate2 Certificate { get; set; }
}
public class X509ClientCertificateAuthentication
{
    //其他成员
    public X509CertificateValidationMode CertificateValidationMode { get;
set; }
    public X509CertificateValidator CustomCertificateValidator { get; set; }
}
```

下面的代码演示了在对服务进行自我寄宿的情况下,如何将客户端证书进行认证的模式设置成 `PeerOrChainTrust`。

```
using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    ServiceCredentials serviceCredentials =
        host.Description.Behaviors.Find<ServiceCredentials>();
    if (null == serviceCredentials)
    {
        serviceCredentials = new ServiceCredentials();
        serviceCredentials.ClientCertificate.Authentication.CertificateValidationMode =
            X509CertificateValidationMode.PeerOrChainTrust;
        host.Description.Behaviors.Add(serviceCredentials);
    }
    host.Open();
    //...
}
```

上面这段代码中设置的服务行为可以通过下面一段配置来表示。

```
<system.serviceModel>
...
<behaviors>
  <serviceBehaviors>
    <behavior name="setCertAuthentication">
      <serviceCredentials>
        <clientCertificate>
          <authentication certificateValidationMode="PeerOrChainTrust"/>
        </clientCertificate>
      </serviceCredentials>
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
```

(3) 证书与 Windows 账号映射

不论是 IIS 还是 AD,都提供了让你将一个 X.509 证书映射为 Windows 账号的特性。由于篇幅所限,这里不对此展开介绍。关于证书和 Windows 账号认证的话题,可以参考 [http://technet.microsoft.com/zh-cn/library/cc736706\(WS.10\).aspx](http://technet.microsoft.com/zh-cn/library/cc736706(WS.10).aspx)。更多针对 AD 的映射可以参考 [http://technet.microsoft.com/en-us/library/cc758484\(WS.10\).aspx](http://technet.microsoft.com/en-us/library/cc758484(WS.10).aspx)。

对于 WCF 来说,用于实现针对客户端证书认证的 `X509ClientCertificateAuthentication` 通过如下所示的 `MapClientCertificateToWindowsAccount` 属性开启或者关闭映射。

```
public class X509ClientCertificateAuthentication
{
    // Fields
    public bool MapClientCertificateToWindowsAccount { get; set; }
}
```

我们可以通过 `X509ClientCertificateAuthentication` 对应配置节的同名配置元素 `mapClientCertificateToWindowsAccount` 来对这个开关进行配置。

```

<system.serviceModel>
  ...
  <behaviors>
    <serviceBehaviors>
      <behavior name="setCertAuthentication">
        <serviceCredentials>
          <clientCertificate>
            <authentication mapClientCertificateToWindowsAccount = "true".../>
          </clientCertificate>
        </serviceCredentials>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>

```

从授权的角度来讲, 将一个 X.509 证书映射为某个 Windows 账号意味着权限的提升。因为被认证后的用户具有了相应 Windows 账号的安全令牌, 具有了访问相应 Windows 资源的权限。因此域策略要求 X.509 证书在映射之前符合其策略。“SChannel”安全数据包强制执行此要求。

对于 .NET Framework 3.5 及之后的版本, WCF 确保了证书在映射到 Windows 账户之前符合域策略。而对于之前版本的 WCF, 却没有这个要求。在这种情况下, 如果启用了映射, 而 X.509 证书不满足域策略, 可能会失败。

7.2.5 ServiceCredentials V.S. ClientCredentials

WCF 的整个认证过程 (服务认证和客户端认证) 都是在信道层 (Channel Layer) 进行的。也就是说在通过绑定构建的信道栈中, 会有一个信道来实现认证和消息保护。但是认证却不仅仅涉及绑定和信道栈, 实际上更多地涉及两个重要的行为, 即 ServiceCredentials 和 ClientCredentials。

由于 WCF 的认证是双向的, 客户端和服务在整个认证过程中处于对等的关系, 因此我们可以将 ServiceCredentials 和 ClientCredentials 看成是两个对等类型。前者为服务认证提供服务凭证, 并控制客户端认证的行为 (比如用户名/密码认证模式和客户端证书认证模式的选择)。后者则为客户端认证提供客户端凭证, 并控制服务认证的行为。先来看看 ServiceCredentials 的定义。

```

public class ServiceCredentials : SecurityCredentialsManager,
  IServiceBehavior
{
    //其他成员
    public override SecurityTokenManager CreateSecurityTokenManager()
    {
        return new ServiceCredentialsSecurityTokenManager(this.Clone());
    }
}
public abstract class SecurityCredentialsManager
{
    //其他成员

```

```

    public abstract SecurityTokenManager CreateSecurityTokenManager();
}
public abstract class SecurityTokenManager
{
    //其他成员
    public abstract SecurityTokenAuthenticator CreateSecurityTokenAuthenticator(
        SecurityTokenRequirement tokenRequirement,
        out SecurityTokenResolver outOfBandTokenResolver);
    public abstract SecurityTokenProvider CreateSecurityTokenProvider(
        SecurityTokenRequirement tokenRequirement);
    public abstract SecurityTokenSerializer CreateSecurityTokenSerializer(
        SecurityTokenVersion version);
}

```

从上面给出的代码可以看出 `ServiceCredentials` 继承了抽象类型 `SecurityCredentialsManager`。该类定义了唯一的抽象方法 `CreateSecurityTokenManager` 用于返回一个 `SecurityTokenManager` 对象。`SecurityTokenManager` 在整个认证体系中是一个非常重要的对象，因为用户凭证在真正用于认证的时候都需要转变成相应的安全令牌。从名称上就可以看出，`SecurityTokenManager` 是“安全令牌管理器”。

`SecurityTokenManager` 也是一个抽象类，它定义了三个 `CreateXxx` 方法分别用于创建三种类型的对象，即 `SecurityTokenAuthenticator`、`SecurityTokenProvider` 和 `SecurityTokenSerializer`。`SecurityTokenSerializer` 用于对完成对安全令牌的序列化/反序列化（或者读/写）工作。作为安全令牌提供者的 `SecurityTokenProvider` 用于实现对令牌的获取、取消和续订等操作。最终针对安全令牌的认证实现在 `SecurityTokenAuthenticator` 中。实际上这个三个对象完成这个认证过程的绝大部分的工作。

在 `ServiceCredentials` 中被重写的 `CreateSecurityTokenManager` 方法中返回的是一个 `ServiceCredentialsSecurityTokenManager` 对象，该对象会根据客户端和服务端凭证类型和其他相关设置创建相应的基于某种凭证的 `SecurityTokenAuthenticator` 和 `SecurityTokenProvider`。比如：

- `UserNameSecurityTokenAuthenticator/UserNameSecurityTokenProvider`
- `WindowsSecurityTokenAuthenticator/ KerberosSecurityTokenProvider`
- `X509SecurityTokenAuthenticator/X509SecurityTokenProvider`

`ClientCredentials` 具有与 `ServiceCredentials` 对等的定义，它的基类也是 `SecurityCredentialsManager`。在重写的 `CreateSecurityTokenManager` 方法中返回的是一个 `ClientCredentialsSecurityTokenManager` 对象。

```

public class ClientCredentials : SecurityCredentialsManager,
IEndpointBehavior
{
    //其他成员
    public override SecurityTokenManager CreateSecurityTokenManager()
    {
        return new ClientCredentialsSecurityTokenManager(this.Clone());
    }
}

```

认证虽然是在信道层进行的,但是整个认证逻辑实现在了 `ServiceCredentials` 和 `ClientCredentials` 之中。那么两者是如何结合起来的呢?

在提起 `ServiceCredentials` 和 `ClientCredentials` 的时候,我一般会说“服务行为 `ServiceCredentials`”和“终结点行为 `ClientCredentials`”。WCF 之所以将二者定义成“行为 (Behavior)”,是为了解决一个问题:将自己传递到信道层。

在服务行为接口 `IServiceBehavior` 和终结点行为接口 `IEndpointBehavior` 均定义了 `AddBindingParameters` 方法,该方法使我们可以将某个对象以绑定参数的形式传递到信道层,信道可以通过绑定上下文获取到该对象。而 `ServiceCredentials` 和 `ClientCredentials` 分别实现 `IServiceBehavior` 和 `IEndpointBehavior` 接口的目的就在于实现 `AddBindingParameters` 方法,将自身传递到信道层。`AddBindingParameters` 方法在 `ServiceCredentials` 和 `ClientCredentials` 中的实现如下面的代码所示。

```
public class ServiceCredentials : SecurityCredentialsManager,
    IServiceBehavior
{
    //其他成员
    void IServiceBehavior.AddBindingParameters(ServiceDescription description,
        ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
        BindingParameterCollection parameters)
    {
        //其他代码
        bindingParameters.Add(this);
    }
}

public class ClientCredentials : SecurityCredentialsManager, IEndpointBehavior
{
    //其他成员
    void IEndpointBehavior.AddBindingParameters(ServiceEndpoint serviceEndpoint,
        BindingParameterCollection bindingParameters)
    {
        //其他代码
        bindingParameters.Add(this);
    }
}
```

服务行为的 `AddBindingParameters` 的调用是在 `ServiceHost` 开启时被调用的,所以 `ServiceHost` 在开启的时候会将 `ServiceCredentials` 设置成只读。在 `ServiceHost` 开启之后对 `ServiceCredentials` 进行的设置是不仅无效的,在大部分情况下还会抛出一个 `InvalidOperationException` 异常。客户端终结点行为是在 `ChannelFactory<TChannel>` 被开启时被调用的, `ClientCredentials` 在 `ChannelFactory<TChannel>` 开启的时候也会被设置成只读,在此情况下对 `ClientCredentials` 所做的设置是无效的,或者直接抛出 `InvalidOperationException` 异常。

比如说,对于如下两段客户端代码,我试图进行两次基于不同客户端凭证的服务调用。但是用于这两种服务调用的服务代理对象都是通过一个相同的 `ChannelFactory<TChannel>` 创建的。由于第一次服务调用会将 `ChannelFactory<TChannel>` 自动开启,并将客户端凭证设置为只读,因此当对客户端凭证进行重复设置的时候,会抛出一个 `InvalidOperationException`

异常，并会提示客户端凭证是只读的。

X.509 证书:

```
using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calulatorservice"))
{
    channelFactory.Credentials.ClientCertificate.SetCertificate(
        StoreLocation.LocalMachine, StoreName.My, X509FindType.FindBySubjectName,
        "Foo");
    ICalculator calculator = channelFactory.CreateChannel();
    var result = calculator.Add(1, 2);
    //...
    channelFactory.Credentials.ClientCertificate.SetCertificate(
        StoreLocation.LocalMachine, StoreName.My, X509FindType.FindBySubjectName,
        "Bar");
    calculator = channelFactory.CreateChannel();
    result = calculator.Add(1, 2);
    //...
}
```

用户名/密码:

```
using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calulatorservice3"))
{
    channelFactory.Credentials.UserName.UserName = "Foo";
    channelFactory.Credentials.UserName.Password = "password";
    ICalculator calculator = channelFactory.CreateChannel();
    var result = calculator.Add(1, 2);
    //...
    channelFactory.Credentials.UserName.UserName = "Foo";
    channelFactory.Credentials.UserName.Password = "password";
    calculator = channelFactory.CreateChannel();
    result = calculator.Add(1, 2);
    //...
}
```

有一点需要注意，`InvalidOperationException` 异常仅仅会在重新设置 X.509 证书和用户名/密码凭证的时候抛出来。如果客户端选用 Windows 凭证，当编写如下的代码对 Windows 凭证的用户名和密码进行重新设置的时候，并没有任何异常会抛出来。但是第二次对 Windows 凭证的设置实际上是无效的，两次服务调用采用的客户端凭证是相同的。

```
using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calulatorservice1"))
{
    channelFactory.Credentials.Windows.ClientCredential.UserName = "Foo";
    channelFactory.Credentials.Windows.ClientCredential.Password = "password";
    ICalculator calculator = channelFactory.CreateChannel();
    var result = calculator.Add(1, 2);
    //...
    channelFactory.Credentials.Windows.ClientCredential.UserName = "Bar";
    channelFactory.Credentials.Windows.ClientCredential.Password = "password";
    calculator = channelFactory.CreateChannel();
    result = calculator.Add(1, 2);
    //...
}
```

7.3 消息保护 (Message Protection)

前面已经对认证进行了详细的介绍。现在关注安全传输的另外两个要素，即消息的一致性和机密性，两者又统称为消息保护 (Message Protection)。

7.3.1 消息的保护级别

消息的保护级别指的是对整个消息或者消息的某个部分实施安全保护采用的级别。按照级别的由低到高，WCF 支持如下三种不同的保护级别。在 WCF 的应用编程接口中，消息保护级别通过如下定义的 `System.Net.Security.ProtectionLevel` 枚举表示。

- **None**: 不采用任何措施来保护消息的一致性和机密性；
- **Sign**: 通过对整个消息或者消息的某个部分进行数字签名以确保消息的一致性；
- **EncryptAndSign**: 通过对整个消息或者消息的某个部分同时进行签名和加密确保消息的一致性和机密性。

```
public enum ProtectionLevel
{
    None,
    Sign,
    EncryptAndSign
}
```

1. 消息保护级别的定义

消息的保护涉及签名和（或者）加密，而与签名与加密相对的是签名检验和解密。要确保消息保护机制的正常进行，客户端和服务双方需要首先在保护级别上达成一致，双方按照这个约定完成属于各自的工作。从这个意义上讲，消息保护级别属于契约的一部分，所以基于消息保护级别的编程体现在契约的定义中。

我们在定义服务契约的时候，可以通过 `ServiceContractAttribute` 特性的 `ProtectionLevel` 属性为整个服务契约设置保护级别。也可以通过 `OperationContractAttribute` 特性的 `ProtectionLevel` 属性为某个具体的操作设置保护级别。`ProtectionLevel` 属性在这两个特性中的定义如下。

```
public sealed class ServiceContractAttribute : Attribute
{
    //其他成员
    public ProtectionLevel ProtectionLevel { get; set; }
    public bool HasProtectionLevel { get; }
}
public sealed class OperationContractAttribute : Attribute
```

```

{
    //其他成员
    public ProtectionLevel ProtectionLevel {get; set; }
    public bool HasProtectionLevel{ get; }
}

```

通过 `ServiceContractAttribute` 和 `OperationContractAttribute` 特性设置的消息保护级别作用在正常的功能性请求消息和回复消息中。而对于出现异常时返回给客户端的错误消息 (`Fault Message`), 我们依然需要加以保护。基于错误消息的保护级别可以通过 `FaultContractAttribute` 特性的 `ProtectionLevel` 进行设置。

```

public sealed class FaultContractAttribute : Attribute
{
    //其他成员
    public ProtectionLevel ProtectionLevel {get; set; }
    public bool HasProtectionLevel{ get; }
}

```

上述的两种方式定义的消息保护级别都是基于整个消息的, 有时候我们仅需要对消息中包含敏感信息的某个部分进行签名或者加密, 那么就需要通过消息契约的方式定义整个消息的结构了。我们通过在实体类上直接应用 `MessageContractAttribute` 特性来定义消息契约, 而分别通过应用 `MessageHeaderAttribute` 和 `MessageBodyMemberAttribute` 特性将目标元素映射为消息报头成员和消息主体成员。

从如下代码中可以看出, 这些特性都具有一个 `ProtectionLevel` 属性。对于 `MessageHeaderAttribute` 和 `MessageBodyMemberAttribute` 特性来说, 这个属性是通过从它们共同的基类 `MessageContractMemberAttribute` 继承得来的。通过 `MessageContractAttribute` 特性设置的保护级别应用于整个消息, 而通过 `MessageContractMemberAttribute` 特性设置的保护级别则是基于对应的消息内容成员。

```

public sealed class MessageContractAttribute : Attribute
{
    //其他成员
    public ProtectionLevel ProtectionLevel {get; set; }
    public bool HasProtectionLevel{ get; }
}
public abstract class MessageContractMemberAttribute : Attribute
{
    //其他成员
    public ProtectionLevel ProtectionLevel {get; set; }
    public bool HasProtectionLevel{ get; }
}
public class MessageHeaderAttribute : MessageContractMemberAttribute
{
    //省略成员
}
public class MessageBodyMemberAttribute : MessageContractMemberAttribute
{
    //省略成员
}

```

2. 消息保护级别的作用范围

通过上面的介绍我们知道了可以通过一系列基于契约(服务契约、错误契约和消息契约)的特性来定义消息的保护级别。如果我们在这些特性中设置了不同的保护级别,它们之间具有怎样的优先级? WCF 又采用怎样的策略来决定最终的消息保护级别呢?

定义消息保护级别的 6 个特性分别位于如图 7-15 所示层次结构的 4 个层次中。低层次可以继承离它最近的高层次的消息保护级别。举个具体的例子,如果通过 `ServiceContractAttribute` 特性在服务契约级别将保护级别设置为 `Sign`,该服务契约所有的操作、操作的错误契约,以及操作使用到的消息契约的默认保护级别都变成 `Sign`。而服务操作可以通过 `OperationContractAttribute` 特性将保护级别设置成 `EncryptAndSign`,那么不仅仅是该操作,就连基于该操作的错误契约和消息契约对应的保护级别也都变成 `EncryptAndSign`。

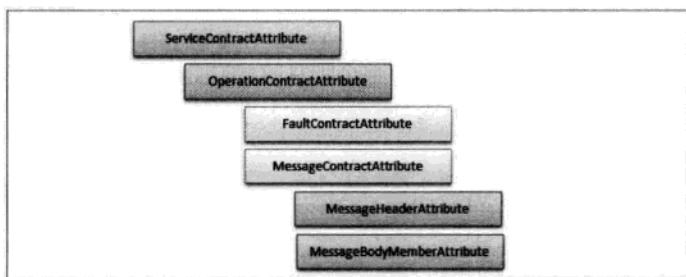


图 7-15 定义消息保护级别不同方式的优先级层次

3. 绑定采用怎样的消息保护级别

前面我们着重介绍了如何在契约上定义消息的保护级别,接下来我们将关注点放在绑定上面。主要关注两个问题:

- 在默认的情况下绑定采用怎样的保护级别?
- 绑定的保护级别可以自定义吗?

对于第一个问题,为了让读者有一个深刻的印象,我不直接告诉你答案,而是要通过编程的方式自己去获取这个答案。在这里我们需要用到一个名为 `System.ServiceModel.Channels.ISecurityCapabilities` 的接口。`ISecurityCapabilities` 定义了一些简单的属性成员用于检测绑定具有怎样的安全相关的属性,其中就包括消息的保护级别。如下面的代码片段所示,`ISecurityCapabilities` 具有 `SupportedRequestProtectionLevel` 和 `SupportedResponseProtectionLevel` 两个属性,分别表示对应的绑定对于请求消息和回复消息采用怎样的保护级别。

```
public interface ISecurityCapabilities
{
    //其他成员
    ProtectionLevel SupportedRequestProtectionLevel { get; }
    ProtectionLevel SupportedResponseProtectionLevel { get; }
}
```

现在就来检测基于某种安全模式下的绑定在默认情况下采用怎样的消息保护级别。为了使我们的程序显得简洁,我写了如下一个针对 `Binding` 类型的扩展方法 `PrintProtectionLevel`, 用于输出绑定对请求和回复消息采用的保护级别。

```
public static class BindingExtension
{
    public static void PrintProtectionLevel(this Binding binding, string
securityMode)
    {
        var bindingParameters = new BindingParameterCollection();
        var requestProtectionLevel = binding.GetProperty<ISecurityCapabi-
lities>(bindingParameters).SupportedRequestProtectionLevel;
        var responseProtectionLevel = binding.GetProperty<ISecurityCapabi-
lities>(bindingParameters).SupportedResponseProtectionLevel;
        Console.WriteLine("{0, -25}{1, -20}{2,-20}", securityMode, request-
ProtectionLevel, responseProtectionLevel);
    }
}
```

现在通过下面的代码检测 `BasicHttpBinding` 针对 4 种不同的安全级别默认采用怎样的消息保护级别。从输出结果可以很清楚地看到,除了 `TransportCredentialOnly` 之外, `BasicHttpBinding` 都采用 `EncryptAndSign` 保护级别。

```
Console.WriteLine("{0, -25}{1, -20}{2,-20}", "", "Request", "Response");
var binding = new BasicHttpBinding(BasicHttpSecurityMode.Transport);
binding.PrintProtectionLevel("Transport");

binding = new BasicHttpBinding(BasicHttpSecurityMode.Message);
binding.Security.Message.ClientCredentialType =
BasicHttpMessageCredentialType.Certificate;
binding.PrintProtectionLevel("Message");

binding = new
BasicHttpBinding(BasicHttpSecurityMode.TransportWithMessageCredential);
binding.PrintProtectionLevel("Mixed");

binding = new
BasicHttpBinding(BasicHttpSecurityMode.TransportCredentialOnly);
binding.PrintProtectionLevel("TransportCredentialOnly");
```

输出结果:

	Request	Response
Transport	EncryptAndSign	EncryptAndSign
Message	EncryptAndSign	EncryptAndSign
Mixed	EncryptAndSign	EncryptAndSign
TransportCredentialOnly	None	None

如果将上面的测试程序用于其他的绑定 (`WSHttpBinding`/`WS2007DualHttpBinding`、`WSDualHttpBinding`、`NetTcpBinding`、`NetNamedPipeBinding` 及 `Message` 和 `Both` 模式下的 `NetMsmqBinding`), 你会发现现在安全被开启的情况下, 这些绑定默认都采用最高的消息保护级别 `EncryptAndSign`。

但是我们编写的扩展方法不能用于 `Transport` 模式下的 `NetMsmqBinding`。不过在表示

NetMsmqBinding 基于 Transport 安全的类型 MsmqTransportSecurity 中具有一个 MsmqProtectionLevel 属性返回采用的消息保护级别。从应用在该属性上的 DefaultValueAttribute 特性的定义中, 可以直接看出 NetMsmqBinding 在 Transport 模式下默认采用的消息保护级别为 Sign。

```
public sealed class MsmqTransportSecurity
{
    //其他成员
    [DefaultValue(1)]
    public ProtectionLevel MsmqProtectionLevel { get; set; }
}
```

前面讨论了对于常用的绑定针对相应的安全模式默认采用的消息保护级别, 接下来我们讨论的话题是: 这些默认的保护级别可以自定义吗? 答案是“部分可以”。只可以修改三个基于局域网的绑定针对 Transport 安全模式下的消息保护级别。

对于 NetMsmqBinding, 可以通过 MsmqTransportSecurity 的 MsmqProtectionLevel 进行设置。而用于设置 NetTcpBinding 和 NetNamedPipeBinding 基于 Transport 安全的 TcpTransportSecurity 和 NamedPipeTransportSecurity 类型中, 都具有 ProtectionLevel 属性用于进行消息保护级别的显式设置。

从应用在该属性上的 DefaultValueAttribute 特性中可以看出默认值为 EncryptAndSign。可以通过编程或者配置的方式来指定 NetTcpBinding、NetNamedPipeBinding 和 NetMsmqBinding 在 Transport 安全模式下的消息保护级别。

```
public sealed class TcpTransportSecurity
{
    //其他成员
    [DefaultValue(2)]
    public ProtectionLevel ProtectionLevel { get; set; }
}
public sealed class NamedPipeTransportSecurity
{
    //其他成员
    [DefaultValue(2)]
    public ProtectionLevel ProtectionLevel { get; set; }
}
```

4. 契约消息保护级别 VS 绑定的消息保护级别

定义在契约上的消息保护级别实际上为 WCF 实施消息保护设置了一个“最低标准”。整个消息保护机制 (不论是签名还是加密), 都是在信道层实现的。而信道层最终是通过绑定来实现的, 绑定的属性决定了信道层处理消息的能力。而绑定安全方面的属性自然就决定了最终的信道层是否有能力对消息实施签名和加密。

一方面, 以契约形式定义的消息保护级别帮助信道层决定应该对传入的消息采取哪个级别的保护机制。另一方面, 如果绑定所能提供的消息保护能力不能达到这个最低标准, 就会抛出异常。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract(ProtectionLevel = ProtectionLevel.EncryptAndSign)]
    double Add(double x, double y);
}
```

我们通过上面的代码将服务契约 ICalculator 的 Add 操作的保护级别设置成 EncryptAndSign。

```
<system.serviceModel>
  <bindings>
    <ws2007HttpBinding>
      <binding name="bindingWithNoneSecurityMode">
        <security mode="None"/>
      </binding>
    </ws2007HttpBinding>
  </bindings>
  ...
</system.serviceModel>
```

但是我们通过上面的配置将终结点使用到的 WS2007HttpBinding 的安全模式设置成 None。那么在对服务进行寄宿的时候,就会抛出如图 7-16 所示的 InvalidOperationException 异常,提示“必须保护请求消息……”。

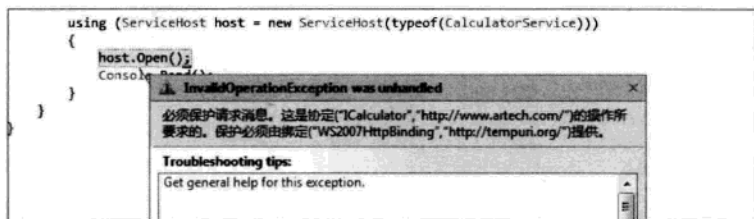


图 7-16 绑定安全设置与消息保护级别不符导致的异常

在这里有一个很多人会忽视的要点。表示消息保护级别的 ProtectionLevel 类型是一个枚举类型,所以它肯定有一个默认值。这个默认值就是 None,也就是说当我们没有显式地指定契约采用哪个保护级别的时候,默认值就是 None。但是这种情况和你显式将保护级别设置为 None 的效果是完全不一致的。因为前者真正采用的保护级别(当绑定安全被开启)实际上是 EncryptAndSign,后者才是 None。那么 WCF 如何来区分这两种情况呢?

如果你足够细心,应该会发现上面介绍的定义消息保护级别的特性中,除了具有一个可读可写的 ProtectionLevel 属性之外,还具有一个只读的 HasProtectionLevel 属性,该属性表示你是否对消息保护级别进行了“显式”的设置。

可以通过一个简单的实验来演示 HasProtectionLevel 的作用。我定义了如下两个服务契约 IServiceContract1 和 IServiceContract2,前者没有对 ProtectionLevel 进行相应的设置,后者被显式地设置为 None。

```
[ServiceContract]
public interface IServiceContract1
{
    [OperationContract]
    void DoSomething();
}

[ServiceContract(ProtectionLevel = ProtectionLevel.None)]
public interface IServiceContract2
{
    [OperationContract]
    void DoSomething();
}
```

然后编写了如下的代码。基于上面两个接口类型生成相应的 `ContractDescription` 对象，然后将它们的 `ProtectionLevel` 和 `HasProtectionLevel` 属性输出。从最终的输出结果可以很清楚地看到两种情况下 `ProtectionLevel` 属性值都是 `None`，但是只有在显式地设置了 `ProtectionLevel` 的情况下，`HasProtectionLevel` 属性才会返回 `True`。WCF 就是根据 `ContractDescription` 的这两个属性决定最终采用怎样的消息保护级别的。

```
ContractDescription contract1 =
ContractDescription.GetContract(typeof(IServiceContract1));
ContractDescription contract2 =
ContractDescription.GetContract(typeof(IServiceContract2));

Console.WriteLine("{0,-10}{1,-20}{2,-20}", "", "ProtectionLevel",
"HasProtectionLevel");

Console.WriteLine("{0,-10}{1,-20}{2,-20}", "contract1",
contract1.ProtectionLevel, contract1.HasProtectionLevel);
Console.WriteLine("{0,-10}{1,-20}{2,-20}", "contract2",
contract2.ProtectionLevel, contract2.HasProtectionLevel);
```

输出结果:

	ProtectionLevel	HasProtectionLevel
contract1	None	False
contract2	None	True

关于消息保护级别与绑定的关系，还有一点需要着重强调。虽然我们可以对同一个服务契约下的操作设置不同的保护级别，但是在 WSDL 中需要基于 WS-Addressing 中的寻址机制来识别基于操作的保护级别。在使用的绑定不支持 WS-Addressing 的情况下（比如 `BasicHttpBinding`），它会选择所有操作中等级最高的那个作为所有操作的保护级别。比如说对于如下定义的服务契约 `ICalculator`，在使用 `BasicHttpBinding` 的情况下，两个操作采用的保护级别都是 `EncryptAndSign`。

```
[ServiceContract]
public interface ICalculator
{
    [OperationContract(ProtectionLevel = ProtectionLevel.Sign)]
    double Add(double x, double y);
    [OperationContract(ProtectionLevel = ProtectionLevel.EncryptAndSign)]
    double Subtract(double x, double y);
}
```


这实际上会为你的应用带来一个很隐晦的问题。为了将这个问题阐述得更加清楚，我通过一个例子来说明。还是沿用我们的计算服务的例子，下面是我们再熟悉不过的服务契约的定义，Add 操作的保护级别被设置成 Sign。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract(ProtectionLevel = ProtectionLevel.Sign)]
    double Add(double x, double y);
}
```

但是这个服务契约并没有被客户端共享，而客户端服务契约中定义了一个额外的操作 Substract，该操作的保护级别并未做显式设置。

```
[ServiceContract(Namespace = "http://www.artech.com/")]
public interface ICalculator
{
    [OperationContract(ProtectionLevel = ProtectionLevel.Sign)]
    double Add(double x, double y);
    [OperationContract]
    double Substract(double x, double y);
}
```

现在选择 BasicHttpBinding 作为终结点的绑定，并将安全模式设置成 Message。当客户端调用 Add 操作的时候。会抛出如图 7-17 所示的 MessageSecurityException 异常，提示“主签名必须加密”。但是当你将客户端 Substract 删除或者将 Substract 操作的消息保护级别也设置成 Sign 时，这个异常将不会出现。

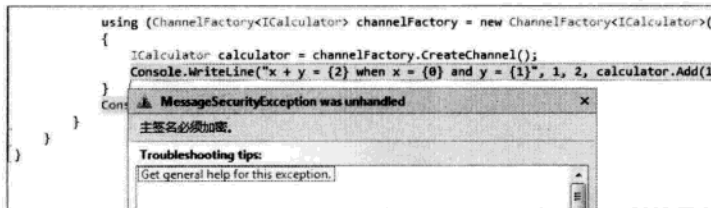


图 7-17 在不支持 WS-Addressing 情况下服务契约不一致导致的异常

出现这样的异常的原因在于：对不支持 WS-Addressing 的 BasicHttpBinding 来说，会选择所有操作中等级最高的那个作为所有操作的保护级别。对于客户端来说，由于 Substract 没有对保护级别进行显式设置，默认采用最高等级的 EncryptAndSign。但是服务端的等级却是 Sign。

在这种情况下，请求消息会同时被加密和签名。请求消息被服务端接收之后，虽然它对应的等级是 Sign，但是依然能够处理该请求。这就是所谓的“消息保护级别的最低标准”原则，定义在契约中的保护级别只是确立了一个消息保护的“底线”。不能低于这个最低标准，但是可以高于它。但是服务执行正常的运算后，只会按照定义在本地契约中设置的保护级别对回复消息进行签名。客户端接收到这个仅仅被签名的回复消息，会发现等级不够，所以才

会提示“主签名必须加密”。

7.3.2 签名与加密的实现

对消息进行签名和加密分别解决了消息的一致性和机密性问题。而最终是仅采用签名还是签名与加密共用取决于契约中对消息保护级别的设置。但是具体的签名和加密在整个 WCF 框架体系中如何实现？是采用对称加密还是非对称加密？密钥如何而来？

在本章中我不断在强调这么一个要点，即整个安全传输的实施最终是在信道层实现的。信道层是根据终结点绑定创建的，而绑定从结构上是一系列绑定元素的有序集合。当绑定安全开启的时候，决定最终安全传输实现方式的必然是某一个或者多个绑定元素。了解相关绑定元素可以帮助读者从本质上理解安全传输实现原理。

为了演示方便，我写了如下一个针对 Binding 类型的扩展方法 ListAllBindingElements，该方法会将所有的绑定元素的类型打印出来。接下来，我们就通过这个扩展方法应用了哪些常见的绑定，看看最终决定安全传输的是哪些绑定元素。

```
public static class BindingExtension
{
    public static void ListAllBindingElements(this Binding binding)
    {
        int i = 0;
        foreach (var bindingElement in binding.CreateBindingElements())
        {
            Console.WriteLine("\t{0}.{1}", ++i, bindingElement.GetType().FullName);
        }
    }
}
```

1. BasicHttpBinding

我们先来看看三种典型安全模式 (Transport、Message 和 Mixed) 下的 BasicHttpBinding 具体由哪些绑定元素构成，为此我编写了如下的程序。

```
BasicHttpBinding binding = new
BasicHttpBinding(BasicHttpSecurityMode.Transport);
Console.WriteLine("Transport:");
binding.ListAllBindingElements();

binding = new BasicHttpBinding(BasicHttpSecurityMode.Message);
binding.Security.Message.ClientCredentialType =
BasicHttpMessageCredentialType.Certificate;
Console.WriteLine("Message:");
binding.ListAllBindingElements();

binding = new
BasicHttpBinding(BasicHttpSecurityMode.TransportWithMessageCredential);
Console.WriteLine("Mixed:");
binding.ListAllBindingElements();
```

输出结果:

```

Transport:
    1.System.ServiceModel.Channels.TextMessageEncodingBindingElement
    2.System.ServiceModel.Channels.HttpsTransportBindingElement
Message:
    1.System.ServiceModel.Channels.AsymmetricSecurityBindingElement
    2.System.ServiceModel.Channels.TextMessageEncodingBindingElement
    3.System.ServiceModel.Channels.HttpTransportBindingElement
Mixed:
    1.System.ServiceModel.Channels.TransportSecurityBindingElement
    2.System.ServiceModel.Channels.TextMessageEncodingBindingElement
    3.System.ServiceModel.Channels.HttpsTransportBindingElement

```

下面来具体分析一下最终在不同安全模式下输出的绑定元素列表。在 Mixe 安全模式下对服务的验证、消息签名和加密都是基于 Transport 安全, Message 安全仅用于对客户端的认证。所以对于 Transport 和 Mixed 模式, 消息保护都是通过 HttpsTransportBindingElement 来实现的。从名称就可以看出, 这是一个基于 HTTPS 的传输绑定元素, 这也再次印证了 BasicHttpBinding 通过 HTTPS 实现 Transport 安全模式的说法。

在 Message 安全模式的三个绑定元素中, 很明显和安全传输相关的是 AsymmetricSecurityBindingElement。从名称我们就知道, 该绑定元素通过非对称加密的方式提供签名和加密的实现。对于请求消息来说, 发送方使用自己的私钥对消息进行签名, 使用接收方的公钥对消息进行加密。接收方采用发送方的公钥验证签名, 用自己的私钥对消息进行解密。这也是为什么在选择了 Message 安全模式的情况下, 基于用户名/密码的客户端凭证不被支持的真正原因。

2. WSHttpBinding、WS2007HttpBinding 和 WSDualHttpBinding

我们按照相同的方式来分析基于 WS 的绑定。由于 WSHttpBinding 和 WS2007HttpBinding 仅在实现 WS-* 协议上有所不同, 整个设计基本相同, 所以我们仅分析 WS2007HttpBinding 和 WSDualHttpBinding。首先来分析 WS2007HttpBinding。对前面的分析程序略加修改, 将绑定类型替换成 WS2007HttpBinding。

```

WS2007HttpBinding binding = new WS2007HttpBinding(SecurityMode.Transport);
Console.WriteLine("Transport:");
binding.ListAllBindingElements();

binding = new WS2007HttpBinding(SecurityMode.Message);
Console.WriteLine("Message:");
binding.ListAllBindingElements();

binding = new
WS2007HttpBinding(SecurityMode.TransportWithMessageCredential);
Console.WriteLine("Mixed:");
binding.ListAllBindingElements();

```

输出结果:

```

Transport:
    1.System.ServiceModel.Channels.TransactionFlowBindingElement

```

```

2.System.ServiceModel.Channels.TextMessageEncodingBindingElement
3.System.ServiceModel.Channels.HttpsTransportBindingElement
Message:
1.System.ServiceModel.Channels.TransactionFlowBindingElement
2.System.ServiceModel.Channels.SymmetricSecurityBindingElement
3.System.ServiceModel.Channels.TextMessageEncodingBindingElement
4.System.ServiceModel.Channels.HttpTransportBindingElement
Mixed:
1.System.ServiceModel.Channels.TransactionFlowBindingElement
2.System.ServiceModel.Channels.TransportSecurityBindingElement
3.System.ServiceModel.Channels.TextMessageEncodingBindingElement
4.System.ServiceModel.Channels.HttpsTransportBindingElement

```

WS2007HttpBinding 和 BasicHttpBinding 实现 Transport 安全都是基于 HTTPS, 所以 Transport 和 Mixed 安全模式都是通过 HttpsTransportBindingElement 实现了对消息的签名和加密。而对于 Message 安全模式, 最终和安全传输相关的是一个叫做 SymmetricSecurityBindingElement 的绑定元素。从名称可以猜出, SymmetricSecurityBindingElement 采用了对称加密实现了对消息的签名和加密。这就意味着, 客户端和服务在进行正式的功能性消息交换之前, 会相互协商生成一个仅限于双方知道的密钥。

接着来看用户双向通信的 WSDualHttpBinding。通过前面的接收, 我们已经知道了该绑定仅支持 Message 安全模式。同样调用 ListAllBindingElements 扩展方法列出 WSDualHttpBinding 在 Message 安全模式下的所有绑定元素。

```

WSDualHttpBinding binding = new WSDualHttpBinding(WSDualHttpSecurityMode.
Message);
Console.WriteLine("Message:");
binding.ListAllBindingElements();

```

输出结果:

```

Message:
1.System.ServiceModel.Channels.TransactionFlowBindingElement
2.System.ServiceModel.Channels.ReliableSessionBindingElement
3.System.ServiceModel.Channels.SymmetricSecurityBindingElement
4.System.ServiceModel.Channels.CompositeDuplexBindingElement
5.System.ServiceModel.Channels.OneWayBindingElement
6.System.ServiceModel.Channels.TextMessageEncodingBindingElement
7.System.ServiceModel.Channels.HttpTransportBindingElement

```

从输出结果可以看到, WSDualHttpBinding 和 WS2007HttpBinding (WSHttpBinding) 在实现 Message 安全模式方面采用相同的机制, 都是采用 SymmetricSecurityBindingElement 来实现的。最终进行的签名和加密都是采用对称加密的方式来实现的。

3. NetTcpBinding 和 NetNamedPipeBinding

我们按照之前的方式来分析另外两个主要应用于局域网环境中的两个绑定, 即 NetTcpBinding 和 NetNamedPipeBinding。这两个绑定和之前介绍的基于 HTTP/HTTPS 传输协议的绑定有所不同。不论是 BasicHttpBinding 还是 WSHttpBinding、WS2007HttpBinding 和 WSDualHttpBinding, 当绑定的安全模式确定之后, 绑定元素集合就确定了。但是对于

NetTcpBinding 和 NetNamedPipeBinding 来说, 如果采用 Transport 安全模式, 最终的绑定元素集合还和采用的认证方式有关。

下面是针对 NetTcpBinding 的分析程序, 对于 Transport 和 Mixed 安全模式, 我们又分两种情况: 将客户端凭证类型分别设置成 Windows 和 Certificate。

```
NetTcpBinding binding = new NetTcpBinding(SecurityMode.Transport);
binding.Security.Transport.ClientCredentialType =
    TcpClientCredentialType.Windows;
Console.WriteLine("Transport (Windows):");
binding.ListAllBindingElements();

binding = new NetTcpBinding(SecurityMode.Transport);
binding.Security.Transport.ClientCredentialType =
    TcpClientCredentialType.Certificate;
Console.WriteLine("Transport (Certificate):");
binding.ListAllBindingElements();

binding = new NetTcpBinding(SecurityMode.Message);
Console.WriteLine("Message:");
binding.ListAllBindingElements();

binding = new NetTcpBinding(SecurityMode.TransportWithMessageCredential);
binding.Security.Message.ClientCredentialType =
    MessageCredentialType.Windows;
Console.WriteLine("Mixed (Windows):");
binding.ListAllBindingElements();

binding = new NetTcpBinding(SecurityMode.TransportWithMessageCredential);
binding.Security.Message.ClientCredentialType =
    MessageCredentialType.Certificate;
Console.WriteLine("Mixed (Certificate):");
binding.ListAllBindingElements();
```

输出结果:

```
Transport (Windows):
    1.System.ServiceModel.Channels.TransactionFlowBindingElement
    2.System.ServiceModel.Channels.BinaryMessageEncodingBindingElement
    3.System.ServiceModel.Channels.WindowsStreamSecurityBindingElement
    4.System.ServiceModel.Channels.TcpTransportBindingElement
Transport (Certificate):
    1.System.ServiceModel.Channels.TransactionFlowBindingElement
    2.System.ServiceModel.Channels.BinaryMessageEncodingBindingElement
    3.System.ServiceModel.Channels.SslStreamSecurityBindingElement
    4.System.ServiceModel.Channels.TcpTransportBindingElement
Message:
    1.System.ServiceModel.Channels.TransactionFlowBindingElement
    2.System.ServiceModel.Channels.SymmetricSecurityBindingElement
    3.System.ServiceModel.Channels.BinaryMessageEncodingBindingElement
    4.System.ServiceModel.Channels.TcpTransportBindingElement
Mixed (Windows):
    1.System.ServiceModel.Channels.TransactionFlowBindingElement
    2.System.ServiceModel.Channels.TransportSecurityBindingElement
    3.System.ServiceModel.Channels.BinaryMessageEncodingBindingElement
    4.System.ServiceModel.Channels.SslStreamSecurityBindingElement
    5.System.ServiceModel.Channels.TcpTransportBindingElement
```

```
Mixed(Certificate):
    1.System.ServiceModel.Channels.TransactionFlowBindingElement
    2.System.ServiceModel.Channels.TransportSecurityBindingElement
    3.System.ServiceModel.Channels.BinaryMessageEncodingBindingElement
    4.System.ServiceModel.Channels.SslStreamSecurityBindingElement
    5.System.ServiceModel.Channels.TcpTransportBindingElement
```

从输出结果可以看出这样的问题: NetTcpBinding 同样采用 SymmetricSecurityBindingElement 实现 Message 模式安全。但是对于 Transport 安全模式的实现却还决定于客户端凭证类型 (或者说认证方式)。在 Windows 认证下, WindowsStreamSecurityBindingElement 用于实现 Transport 安全。而对于非 Windows 认证, 对应的绑定元素变成了 SslStreamSecurityBindingElement。对于实现 Transport 安全的两个绑定元素 WindowsStreamSecurityBindingElement 和 SslStreamSecurityBindingElement, 它们具有相同的基类, 即 StreamUpgradeBindingElement。

```
public abstract class StreamUpgradeBindingElement : BindingElement
{
    //省略成员
}
public class WindowsStreamSecurityBindingElement :
    StreamUpgradeBindingElement
{
    //省略成员
}
public class SslStreamSecurityBindingElement : StreamUpgradeBindingElement
{
    //省略成员
}
```

StreamUpgradeBindingElement 实现消息保护的机制被称为 Stream Upgrade。在这种机制下, 经过编码转化成的二进制流在进入传输层之前, 会被拦截。拦截得到的二进制流经过签名和加密后再被重新流入传输层发送。

还有一点需要特别指出的是, StreamUpgradeBindingElement 并不会创建相应的信道, 而是将功能实现的对象作为绑定参数传入信道层, 传输信道再将其取出并完成相应的签名和加密工作。目前为止, 只有两种面向连接的网络协议的传输信道支持 Stream Upgrade, 即 TCP 和命名管道。至于两种具体的 StreamUpgradeBindingElement, SslStreamSecurityBindingElement 采用 TLS/SSL 协议, WindowsStreamSecurityBindingElement 则基于 Windows 安全协议。

程序的输出结果还反映了另一个现象, 客户凭证对最终使用的绑定协议的影响仅限于 Transport 安全模式。对于 Mixed 模式, 不论采用怎样的客户凭证, 最终实现 Transport 安全的绑定元素总是 SslStreamSecurityBindingElement。也就是说 Mixed 模式下的 NetTcpBinding 总是采用 TLS/SSL 实现 Transport 安全。换句话说, 如果你使用 Mixed 模式下的 NetTcpBinding, 那么必须为服务指定一个 X.509 证书作为服务的凭证。

由于 NetNamedPipeBinding 只支持 Transport 安全模式, 并且在该安全模式下默认使用 Windows 认证, 因此最终实现 Transport 安全的总是 WindowsStreamSecurityBindingElement。

```
NetNamedPipeBinding binding = new
NetNamedPipeBinding(NetNamedPipeSecurityMode.Transport);
binding.ListAllBindingElements();
```

输出结果:

```
1.System.ServiceModel.Channels.TransactionFlowBindingElement
2.System.ServiceModel.Channels.BinaryMessageEncodingBindingElement
3.System.ServiceModel.Channels.WindowsStreamSecurityBindingElement
4.System.ServiceModel.Channels.NamedPipeTransportBindingElement
```

4. NetMsmqBinding

接着来分析 NetMsmqBinding, 直接将我们定义的 ListAllBindingElements 扩展方法应用在三个具有不同安全模式 (Transport、Message 和 Both) 的 NetMsmqBinding 对象上。

```
NetMsmqBinding binding = new NetMsmqBinding(NetMsmqSecurityMode.Transport);
Console.WriteLine("Transport:");
binding.ListAllBindingElements();
```

```
. binding = new NetMsmqBinding(NetMsmqSecurityMode.Message);
Console.WriteLine("Message:");
binding.ListAllBindingElements();
```

```
binding = new NetMsmqBinding(NetMsmqSecurityMode.Both);
Console.WriteLine("Both:");
binding.ListAllBindingElements();
```

输出结果:

Transport:

```
1.System.ServiceModel.Channels.BinaryMessageEncodingBindingElement
2.System.ServiceModel.Channels.MsmqTransportBindingElement
```

Message:

```
1.System.ServiceModel.Channels.SymmetricSecurityBindingElement
2.System.ServiceModel.Channels.BinaryMessageEncodingBindingElement
3.System.ServiceModel.Channels.MsmqTransportBindingElement
```

Both:

```
1.System.ServiceModel.Channels.SymmetricSecurityBindingElement
2.System.ServiceModel.Channels.BinaryMessageEncodingBindingElement
3.System.ServiceModel.Channels.MsmqTransportBindingElement
```

输出结果反映了这样一个结论: 基于 NetMsmqBinding 的 Transport 安全通过 MsmqTransportBindingElement 在传输信道中完成了, 而 Message 安全还是通过 SymmetricSecurityBindingElement 采用对称签名和加密实现的。

5. 总结

前面我们从横向比较各种常见的绑定在不同安全模式下具有怎样的绑定元素列表。由于绑定元素是认识安全传输实现本质的核心, 所以现在抛开不同绑定类型的差异, 直接看看 Transport 和 Message 这两种不同的安全模式最终都是由哪些具体的绑定元素实现的。

Transport 安全对应以下 4 种绑定元素: HttpsTransportBindingElement、WindowsStream

SecurityBindingElement、SslStreamSecurityBindingElement 和 MsmqTransportBindingElement。其中三个基于互联网的绑定 (BasicHttpBinding、WSHttpBinding 和 WS2007HttpBinding) 因为都是采用 HTTPS 实现的, 所以最终落实到 HttpsTransportBindingElement 上。两种基于局域网的绑定, NetTcpBinding 和 NetNamePipeBinding, 采用 Stream Upgrade 的机制实现 Transport 安全, 具体来说又落实到 WindowsStreamSecurityBindingElement 和 SslStreamSecurityBindingElement 两个绑定元素上。

Message 安全对应如下三种绑定元素: AsymmetricSecurityBindingElement、SymmetricSecurityBindingElement 和 TransportSecurityBindingElement。其中 TransportSecurityBindingElement 使用在 Mixed 安全模式下。对于 Message 模式, 除了 BasicHttpBinding 使用 AsymmetricSecurityBindingElement 外, 其余都是使用 SymmetricSecurityBindingElement。

7.3.3 安全会话 (Secure Sessions)

对于基于 WS-Security 协议簇的 Message 安全模式, 消息的签名和加密都是采用对称加密的方式。而使用的密钥在正式进行消息交换之前通过协商产生。那么这种为了生成安全密钥而进行的协商是针对每次基于服务调用的消息交换, 还是基于某个服务代理对象呢? 换句话说, 如果我通过一个相同的服务代理进行 N 次服务调用, 上述的协商过程会进行 1 次还是 N 次?

此外, 客户端在处理来自同一个客户端的 N 个服务调用请求时, 会有一个客户端认证的过程。那么, 这个认证是仅仅进行 1 次还是 N 次? 这就是 WCF 的安全会话需要解决的问题。

1. WS-Secure Conversation 与安全会话

在 7.1 节中我们简单地介绍了 WS-Security 协议簇之一的 WS-Secure Conversation。该协议旨在客户端和 Web 服务之间建立一个安全上下文, 以避免认证的重复进行。这个上下文会维持一个安全密钥对传输的消息进行签名和加密。

建立安全上下文的根本目的在于提高性能, 不论是安全认证 (比如说采用基于 MembershipProvider 的用户名/密码认证, 需要访问维护用户账号的数据库), 还是为了生成安全密钥的消息交换, 都可能是一个耗时的过程。如果客户端在一个较短的时间内会频繁地调用服务, 通过事先创建一个安全上下文可以避免重复的认证和安全密钥的协商过程, 这无疑会极大地提高服务调用的性能。

WCF 通过安全会话提供对 WS-Secure Conversation 的支持, 而这个会话就代表了 WS-Secure Conversation 中的安全上下文。所有支持基于 WS-Security 的 Message 安全模式的系统绑定 (BasicHttpBinding 支持的所谓 Message 安全模式和 WS-Security 无关) 都提供了对安全会话的支持。

对于 NetTcpBinding, 由于 TCP 本身基于连接 (连接可以看成是通信双方之间的会话)

的特性, 因此安全会话机制始终是开启的。而对于 WSDualHttpBinding, 由于需要通过会话的机制维护两个 HTTP 通道之间的关系, 因此安全会话机制也始终是开启的。

只有 WSHttpBinding 和 WS2007HttpBinding 才能显式地开启和关闭安全会话。如下面的代码片段所示, 安全会话的开关通过 NonDualMessageSecurityOverHttp 的 EstablishSecurityContext 属性表示。该属性的默认值是 True, 即在默认的情况下开启了安全会话。

```
public sealed class NonDualMessageSecurityOverHttp : MessageSecurityOverHttp
{
    //其他成员
    public bool EstablishSecurityContext { get; set; }
}
```

可以通过编程或者配置的方式开启或者关闭 WSHttpBinding 和 WS2007HttpBinding 的安全会话。安全会话的开关通过绑定配置节下<security>/<message>的 establishSecurityContext 属性表示。在下面的配置中, 我们定义了关闭安全会话的 WS2007HttpBinding。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="disableSecureSessions">
          <security>
            <message establishSecurityContext="false"/>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

2. 实例演示: 验证安全会话是否可以避免重复认证

为了验证安全会话是否能够真正地避免重复认证, 我们进行一个简单的实例演示。为了确定 WCF 是否真正实施了认证, 通过创建 UserNamePasswordValidator 按照我们自定义的方式进行基于用户名/密码的认证。这个自定义的 UserNamePasswordValidator 就是具有如下定义的 SimpleUserNamePasswordValidator。顺便提一下, 抽象类 UserNamePasswordValidator 定义在程序集 System.IdentityModel.dll 中。

```
using System;
using System.Collections.Generic;
using System.IdentityModel.Selectors;
using System.IdentityModel.Tokens;
namespace Artech.WcfServices.Service
{
    public class SimpleUserNamePasswordValidator : UserNamePasswordValidator
    {
        public IDictionary<string, string> UserNamePasswords { get; private set; }
        public SimpleUserNamePasswordValidator()
        {

```

```

        this.UserNamePasswords = new Dictionary<string, string>();
        this.UserNamePasswords.Add("foo", "Password");
        this.UserNamePasswords.Add("bar", "Password");
        this.UserNamePasswords.Add("baz", "Password");
    }
    public override void Validate(string userName, string password)
    {
        Console.WriteLine("开始认证客户端...");
        bool authenticated = true;
        if(!this.UserNamePasswords.ContainsKey(userName.ToLower()))
        {
            authenticated = false;
        }
        if (authenticated)
        {
            if (this.UserNamePasswords[userName.ToLower()] != password)
            {
                authenticated = false;
            }
        }
        if (!authenticated)
        {
            throw new SecurityTokenValidationException(
                "用户名不存在, 或者用户名与密码不符");
        }
    }
}

```

我们的认证很简单, 就是定义了一个包含三个用户账号 (Foo、Bar 和 Baz) 的列表, 它们对应的密码均为 Password。如果用户提供的用户名和密码与这个列表相匹配, 则认证成功。我们在 Validate 方法中通过在控制台上打印一段文字以证明服务端确实在进行客户端认证工作。

依然采用之前计算服务的例子, 为了能够让读者清楚地看到认证是基于服务调用还是基于服务代理, 我们在 CalculatorService 的 Add 方法执行之前也打印出一段文字。

```

public class CalculatorService: ICalculator
{
    double ICalculator.Add(double x, double y)
    {
        Console.WriteLine("Add 操作开始执行...");
        return x + y;
    }
}

```

我们自定义的 UserNamePasswordValidator 通过如下的配置应用到 WCF 的认证体系, 并通过将绑定配置节下 <security>/<message> 的 establishSecurityContext 属性设置为 False 从而将安全会话关闭。当然在客户端的配置中也需要按照如下的方式对绑定的安全会话进行相应的设置。

服务端配置:

```

<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="customizedAuthentication">
          <serviceCredentials>
            <serviceCertificate storeLocation="LocalMachine"
                                storeName="My"
                                x509FindType="FindBySubjectName"
                                findValue="Jinnan-PC"/>
            <userNameAuthentication
userNamePasswordValidationMode="Custom"
customUserNamePasswordValidatorType="Artech.WcfServices.Service.SimpleUser
NamePasswordValidator, Artech.WcfServices.Service" />
          </serviceCredentials>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <bindings>
      <ws2007HttpBinding>
        <binding name="userNameCredentialBinding">
          <security>
            <message clientCredentialType="UserName"
establishSecurityContext="false"/>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService"
              behaviorConfiguration="customizedAuthentication">
        <endpoint address="http://127.0.0.1:3721/calculatorservice"
                  binding="ws2007HttpBinding"
                  bindingConfiguration="userNameCredentialBinding"
                  contract="Artech.WcfServices.Service.Interface.
                    ICalculator"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

客户端配置:

```

<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="userNameCredentialBinding">
          <security mode="Message">
            <message clientCredentialType="UserName"
establishSecurityContext="false"/>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <client>
      <endpoint name="calculatorService"
                behaviorConfiguration="ignoreServiceCertificate"
                address="http://127.0.0.1:3721/calculatorservice"

```

```

        binding="ws2007HttpBinding"
        bindingConfiguration="userNameCredentialBinding"
        contract="Artech.WcfServices.Service.Interface.ICalculator">
    <identity>
        <certificateReference
            storeLocation="LocalMachine"
            storeName="My"
            x509FindType="FindBySubjectName"
            findValue="Jinnan-PC"/>
    </identity>
</endpoint>
</client>
<behaviors>
    <endpointBehaviors>
        <behavior name="ignoreServiceCertificate">
            <clientCredentials>
                <serviceCertificate>
                    <authentication certificateValidationMode="None"/>
                </serviceCertificate>
            </clientCredentials>
        </behavior>
    </endpointBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

最后我们按照如下的方式进行服务调用,即通过创建出来的同一个服务代理先后三次进行服务调用。从服务端的输出结果可以清楚地看到,在关闭安全会话的情况下,认证在每次服务调用中都会进行。(S707)

```

using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService"))
{
    channelFactory.Credentials.UserName.UserName = "Foo";
    channelFactory.Credentials.UserName.Password = "Password";
    ICalculator calculator = channelFactory.CreateChannel();
    calculator.Add(1, 2);
    calculator.Add(1, 2);
    calculator.Add(1, 2);
}

```

输出结果:

```

开始认证客户端...
Add 操作开始执行...
开始认证客户端...
Add 操作开始执行...
开始认证客户端...
Add 操作开始执行...

```

现在通过如下的配置为服务端和客户端绑定开启安全会话。再次运行程序将看到不一样的输出结果。从中不难看出,真正的认证工作只进行了一次而已。(S708)

服务端配置:

```

<configuration>
    <system.serviceModel>

```

```

...
<bindings>
  <ws2007HttpBinding>
    <binding name="userNameCredentialBinding">
      <security >
        <message      establishSecurityContext="true" .../>
      </security>
    </binding>
  </ws2007HttpBinding>
</bindings>
<system.serviceModel>
</configuration>

```

客户端配置:

```

<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="userNameCredentialBinding">
          <security mode="Message">
            <message      establishSecurityContext="true" .../>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>

```

输出结果:

```

开始认证客户端...
Add 操作开始执行...
Add 操作开始执行...
Add 操作开始执行...

```



第 8 章 授权与审核

(Authorization and Auditing)

授权旨在通过权限控制让用户只能执行被允许的功能，访问被许可的资源，它将被授权的实体（功能性操作或者资源）和一组权限集进行关联，通过判断被认证的用户是否具有相匹配的权限来确定该用户针对该实体是否被授权。安全审核旨在对认证和授权事件进行日志记录。既可以在认证/授权成功时进行日志记录，也可以针对基于认证/授权失败事件进行记录。

资源
PDG

我们为应用建立安全保障体系的一个重要目的在于通过权限控制让用户只能执行被允许的功能，访问被许可的资源，这就是本章的主题——授权 (Authorization)。授权的原理其实很简单，即将被授权的实体 (功能性操作或者资源) 和一组权限集进行关联，通过判断被认证的用户是否具有相匹配的权限来确定该用户针对该实体是否被授权。

对于 WCF 服务来说，一个服务具有若干操作。这些操作由于提供的功能或者内部访问的资源不同，需要进行不同的权限设置。借助于 .NET 安全相关的应用编程接口，我们可以通过声明的方式将某个服务操作与调用该操作应当具有的权限集进行关联。当调用某个服务操作的用户被成功认证后，它具有的权限集被获取出来并绑定到当前安全上下文。

WCF 框架本身在试图执行目标操作之前，可以根据当前安全上下文确定该用户是否有权限执行该服务操作。由于当前安全上下文中已经具有了当前访问者的权限集，也可以通过代码的方式来定义授权策略。我们将这两种不同的授权编程方式称为声明式编程 (Declarative Programming) 和命令式编程 (Imperative Programming)。

8.1 身份 (Identity) 与安全主体 (Principal)

在安全领域，认证和授权是两个重要的主题。认证是第一道屏障，守护着整个应用或服务的第一道大门。当访问者叩门请求进入的时候，认证体系通过验证对方提供的凭证确定其真实身份。作为看门人的认证体系，只有在证实了访问者的真实身份的情况下才会为其打开城门，否则将其拒之门外。

当访问者入门之后，并不意味着他可以为所欲为。为了让适合的人干适合的事，就需要为具体的人设置具体的权限，并根据这些权限设置决定试图调用的操作对该访问者是否是安全的。授权的执行是假定已经通过认证确定了访问者的真实身份，因为用于进行授权采用的权限集是基于这个确定的身份的。在真正进入对 WCF 授权的具体介绍之前，有必要来了解一下这个“身份”的问题。

8.1.1 身份

在 .NET 的安全应用编程接口中，身份是通过具有如下定义的 `System.Security.Principal.IIdentity` 接口来表示的。该接口定义了三个只读属性，其中 `Name` 表示身份的名称，`IsAuthenticated` 表示是否经过认证。而 `AuthenticationType` 属性则代表身份认证所采用的类型。

```
public interface IIdentity
{
    string AuthenticationType { get; }
    bool IsAuthenticated { get; }
    string Name { get; }
}
```

通过 `IIdentity` 表示的身份是基于某种认证类型的,不同类型的认证往往对应于不同的身份类型。以 ASP.NET 认证为例,如果采用 Forms 认证,那么认证后的身份通过一个 `System.Web.Security.FormsIdentity` 对象表示。而 Windows Live Passport 认证对应的具体身份类型则是 `System.Web.Security.PassportIdentity`。在这里着重介绍一下 `WindowsIdentity`、`GenericIdentity` 和 `X509Identity` 三种身份类型。

1. WindowsIdentity

`WindowsIdentity` 定义在 `System.Security.Principal` 命名空间下,表示一个基于 Windows 认证的身份。一个采用 `WindowsIdentity` 定义的 Windows 身份具有如下一系列的属性。

```
public class WindowsIdentity : IIdentity,...
{
    //其他成员
    public virtual string Name { get; }
    public string AuthenticationType { get; }
    public virtual bool IsAuthenticated { get; }

    public IdentityReferenceCollection Groups { get; }
    public virtual bool IsAnonymous { get; }
    public virtual bool IsGuest { get; }
    public virtual bool IsSystem { get; }
}
```

对于用于表示认证类型的 `AuthenticationType` 属性来说,在工作组模式下返回 NTLM。对于域模式,如果操作系统是 Vista 或者以后的版本,该属性返回 Negotiate,表示采用 SPNEGO 认证协议。而对于之前的 Windows 版本,则该属性值为 Kerberos。

`Groups` 属性返回 `WindowsIdentity` 对应的 Windows 账号所在的用户组 (User Group)。`IsGuest` 用于判断 Windows 账号是否存在于 Guest 用户组中。`IsSystem` 属性则表示 Windows 账号是否是一个系统账号。

如果你对 ASP.NET 的安全有一定的了解,应该知道我们可以对 IIS 进行相应的配置使 ASP.NET 应用支持匿名用户。也就是说,用户无须提供具体的用户凭证,而是以匿名的方式登录到 ASP.NET 站点中。对于匿名登录,IIS 实际上会采用一个预先指定的 Windows 账号进行登录。而在这里,`IsAnonymous` 属性就表示该 `WindowsIdentity` 对应的 Windows 账号是否是匿名账号。

`WindowsIdentity` 定义了如下一个静态的 `GetAnonymous` 方法用于返回一个表示匿名身份的 `WindowsIdentity` 对象。但是这仅仅是一个空的 `WindowsIdentity` 对象而已,并不对应某个确定的 Windows 账号。

```
public class WindowsIdentity : IIdentity,...
{
    //其他成员
    public static WindowsIdentity GetAnonymous()
}
```

任何一个具体的 Windows 进程总是运行在一个确定的安全身份下。如果手工启动一个 .exe 文件, 被开启的进程会运行在基于当前登录账号的身份下。也可以通过“Run As”的方式选择一个不同于当前登录账号的身份去运行某个 .exe 文件。而对于很多的 Windows 服务, 它们大多运行在某个系统账号下。比如我们熟悉的 IIS (IIS6 或者之后的版本) 在默认的情况下就运行在 Network Service 这个系统账号下面。

当一个线程在这个进程中被创建并启动的时候, 进程的安全身份会自动附加到线程上。WindowsIdentity 为我们提供了如下一个 GetCurrent 静态方法返回基于当前线程 / 进程的 WindowsIdentity。

```
public class WindowsIdentity : IIdentity,...
{
    //其他成员
    public static WindowsIdentity GetCurrent();
}
```

2. GenericIdentity

对于这些常用的认证类型 (比如 Windows 认证、Forms 认证和 Windows Live Passport 认证等), 都具有对应的安全身份类型。如果采用自定义的认证方式, 是否意味着也需要定义一个实现了 IIdentity 接口的类型呢? 实际上是不需要的, 我们可以直接使用 System.Security.Principal.GenericIdentity 这个类型。

正如名称所体现的一样, GenericIdentity 为我们定义了一个一般性的安全身份。GenericIdentity 的定义非常简单, 仅仅实现了定义在 IIdentity 接口的三个只读属性而已。我们可以通过指定用户名或者用户名与认证类型来创建一个 GenericIdentity 对象。下面的代码片段体现了 GenericIdentity 的整个定义。

```
public class GenericIdentity : IIdentity
{
    public GenericIdentity(string name);
    public GenericIdentity(string name, string type);

    public virtual string AuthenticationType { get; }
    public virtual bool IsAuthenticated { get; }
    public virtual string Name { get; }
}
```

由于 GenericIdentity 的 IsAuthenticated 属性是只读的, 因此不能通过构造函数对其进行初始化。那么如何确定一个通过 GenericIdentity 对象表示的安全身份是否已经通过认证了呢? 其实 GenericIdentity 是否是匿名取决于用户名是否为空。如果用户名不为空, 则 IsAuthenticated 返回 True, 否则返回 False。下面给出的代码可以验证这一点。

```
var anonymousIdentity = new GenericIdentity("");
var authenticatedIdentity = new GenericIdentity("Foo");
Debug.Assert(anonymousIdentity.IsAuthenticated == false);
Debug.Assert(authenticatedIdentity.IsAuthenticated == true);
```

3. X509Identity

WCF 具有三种典型的认证方式, 即 Windows 认证、用户名/密码认证和证书认证。认证的方式决定了安全身份的类型, `System.IdentityModel.Claims.X509Identity` 代表了客户端凭证为 X.509 证书时认证后的身份。`X509Identity` 定义在程序集 `System.IdentityModel.dll` 中。从下面给出的定义可以看出 `X509Identity` 仅仅是一个内部 (Internal) 类型。

```
internal class X509Identity : GenericIdentity, IDisposable
{
    //其他成员
    public X509Identity(X500DistinguishedName x500DistinguishedName);
    public X509Identity(X509Certificate2 certificate);

    public X509Identity Clone();
    public void Dispose();
    public override string Name { get; }
}
```

`X509Identity` 直接继承自 `GenericIdentity`。我们可以通过传入一个 `X509Certificate2` 对象或者以 `X500DistinguishedName` 对象表示的证书的标识名称来创建 `X509Identity`。`X509Identity` 重写了 `GenericIdentity` 的 `Name` 属性, 最终作为名称返回的是采用如下格式返回的证书的主题名称和指纹的组合。`X509Identity` 对象的 `AuthenticationType` 属性为“X509”。

```
<<主题名称>>; <<指纹>> (分号之后具有一个空格, 比如:
CN=Foo; 12BA3675C89BD7FE00E3F7E92A620749FB9E6D89)。
```

4. 服务安全上下文中的身份

在服务安全开启的情况下, 服务端在经过认证之后会创建一个上下文用于存储基于当前服务调用相关的安全相关的信息, 其中就包含了代表被认证客户端的安全身份。这个上下文就是通过具有如下定义的 `System.ServiceModel.ServiceSecurityContext` 类型表示的服务安全上下文。

```
public class ServiceSecurityContext
{
    //其他成员
    public static ServiceSecurityContext Current { get; }
    public IIdentity PrimaryIdentity { get; }
    public WindowsIdentity WindowsIdentity { get; }

    public bool IsAnonymous { get; }
    public static ServiceSecurityContext Anonymous { get; }
}
```

可以通过两种方式获取当前的 `ServiceSecurityContext`, 一种是通过 `ServiceSecurityContext` 的静态只读属性 `Current`, 另一种则是通过当前 `OperationContext` 的 `ServiceSecurityContext` 属性。实际上通过这两种方式得到的是同一个 `ServiceSecurityContext`。`ServiceSecurityContext` 对象的同一性可以通过下面的代码来验证。

```
var securityContext1 = OperationContext.Current.ServiceSecurityContext;  
var securityContext2 = ServiceSecurityContext.Current;  
Debug.Assert(object.ReferenceEquals(securityContext1, securityContext2));
```

`ServiceSecurityContext` 具有两个代表客户端身份的属性, 即 `PrimaryIdentity` 和 `WindowsIdentity`。对于 Windows 认证, 这两个属性返回同一个 `WindowsIdentity` 对象。不过需要注意的是, 这里所说的 Windows 认证实际上包括如下三种情况。

- 客户端凭证为 Windows 凭证;
- 客户端凭证为用户名/密码凭证, 并采用 Windows 认证模式;
- 客户端凭证为 X.509 证书凭证, 并允许与 Windows 账号进行映射。

对于不属于上述三种情况的非 Windows 凭证, 当前 `ServiceSecurityContext` 的 `WindowsIdentity` 属性返回 `Null`, 而 `PrimaryIdentity` 属性则因客户端凭证类型和认证方式而有所区别。如果客户端凭证为用户名/密码凭证, 并采用 `Membership` 和 `Custom` 认证模式, 则在成功认证的情况下 `PrimaryIdentity` 的属性返回一个以用户名作为名称的 `GenericIdentity`。如果客户端凭证为 X.509 证书凭证 (不采用 Windows 账号映射机制), 则 `PrimaryIdentity` 的属性返回的是一个 `X509Identity`。

对于匿名客户端 (客户端凭证类型为 `None`), `PrimaryIdentity` 返回一个空用户名的 `GenericIdentity` 对象, 其 `IsAnonymous` 属性为 `True`。表 8-1 体现了成功认证后当前 `ServiceSecurityContext` 的 `PrimaryIdentity` 与客户端凭证类型及认证模式之间的关系。

表 8-1 成功认证后当前 `ServiceSecurityContext` 的 `PrimaryIdentity` 与客户端凭证类型及认证模式之间的关系

None	Windows	UserName			Certificate	
		Windows	Membership	Custom	Default	Windows Account Mapping
Generic-Identity	Windows-Identity	Windows-Identity	GenericIdentity	Generic-Identity	X509 Identity	Windows-Identity

8.1.2 安全主体

毫不夸张地说, 安全主体是整个授权机制的核心。可以简单地将安全主体看成是能够被成功实施授权的实体。一个安全主体具有两个基本的要素, 即基于某个用户的安全身份和该用户具有的权限。绝大部分的授权都是围绕着“角色”进行的, 将一组相关的权限集和一个角色绑定, 然后分配给某个用户。所以在基于角色授权环境下, 可以简单地将安全主体表示成“身份 + 角色”。在 .NET 基于安全的应用编程接口中, 通过 `IPrincipal` 接口表示安全主体。

1. IPrincipal、WindowsPrincipal 和 GenericPrincipal

用于表示安全主体的类型实现了具有如下定义的 System.Security.Principal.IPrincipal 接口。它具有两个成员，只读属性 Identity 表示安全主体的身份，而 IsInRole 用于判断安全主体对应的用户是否被分配了给定的角色。

```
public interface IPrincipal
{
    bool IsInRole(string role);
    IIdentity Identity { get; }
}
```

前面具体介绍了 IIdentity 接口的两个实现，即 WindowsIdentity 和 GenericIdentity。实际上 IPrincipal 也具有相类似的实现类型，即 WindowsPrincipal 和 GenericPrincipal，它们均定义在 System.Security.Principal 命名空间下。

一个安全主体具有身份与权限两个基本要素。在 Windows 安全体系下，某个用户具有的权限决定于它被添加到哪些用户组 (User Group) 中。Windows 默认为我们创建了一些用户组，比如 Administrators 和 Guests 等。也可以根据需要创建自定义用户组。从本质上讲，Windows 的用户组和我们之前谈到的角色并没有本质的区别，都是一组权限的载体。

WindowsPrincipal 的定义如下。表示安全身份的只读属性 Identity 返回一个 WindowsIdentity 对象，在构造函数中指定。在 Windows 安全体系中，一个用户组具有多种不同的标识方式，比如相对标识符 (RID, Relative Identifier)、安全标识符 (SID, Security Identifier) 和用户组名称，对于一些已定义的用户组甚至还可以通过 System.Security.Principal.WindowsBuiltInRole 枚举来表示，所以 WindowsPrincipal 具有若干重载的 IsInRole 方法。

```
public class WindowsPrincipal : IPrincipal
{
    public WindowsPrincipal(WindowsIdentity ntIdentity);
    public virtual bool IsInRole(int rid);
    public virtual bool IsInRole(SecurityIdentifier sid);
    public virtual bool IsInRole(WindowsBuiltInRole role);
    public virtual bool IsInRole(string role);
    public virtual IIdentity Identity { get; }
}
```

一个 GenericPrincipal 对象本质上就是对一个 IIdentity 对象和表示角色列表的字符串数组的封装而已。下面的代码片段体现了整个 GenericPrincipal 的定义。

```
public class GenericPrincipal : IPrincipal
{
    public GenericPrincipal(IIdentity identity, string[] roles);
    public virtual bool IsInRole(string role);
    public virtual IIdentity Identity { get; }
}
```

2. 基于安全主体的授权

一个通过接口 `IPrincipal` 表示的安全主体不仅可以表示被授权用户的身份 (通过 `Identity` 属性), 其本身就具有授权判断的能力 (通过 `IsInRole` 方法)。如果我们在访问者成功认证后根据用户的权限设置构建一个安全主体, 并将其存储在当前的上下文中, 在需要的时候就可以从该安全主体获取出来以完成对授权的实现。

实际上 Windows 授权机制就是按照这样的原理实现的, 而这个所谓的上下文被存储于当前线程的线程本地存储 (TLS: Thread Local Storage)。反映在编程上, 可以通过 `Thread` 类型的 `CurrentPrincipal` 属性来获取或者设置这个当前的安全主体。

```
public sealed class Thread
{
    //其他成员
    public static IPrincipal CurrentPrincipal { get; set; }
}
```

一旦为当前线程设置了安全主体, 在需要确定当前用户是否有权限执行某项操作的时候, 就可以通过上述的这个 `CurrentPrincipal` 属性将设置的安全主体获取出来, 通过调用 `IsInRole` 方法判断当前用户是否具有相应的权限。下面的代码体现了用户需要具有 `Administrators` 角色 (或者 Windows 用户组) 才能执行被授权的操作, 否则会抛出一个安全异常。

```
IPrincipal currentPrincipal = Thread.CurrentPrincipal;
if (currentPrincipal.IsInRole("Administrators"))
{
    //执行被授权的操作
}
else
{
    //抛出安全异常
}
```

编写具体授权逻辑的编程方式称为命令式编程 (Imperative Programming)。如果是针对某个方法的授权 (当前用户是否有权限执行定义方法的所有操作), 还可以采用一种声明式的编程方式 (Declarative Programming)。这需要用到 `System.Security.Permissions.PrincipalPermissionAttribute` 特性。

从如下代码片段给出的关于 `PrincipalPermissionAttribute` 类型的定义不难看出, 这是一个基于代码访问安全 (CAS: CodeAccessSecurity) 的特性 (继承自 `CodeAccessSecurityAttribute`)。如果在某个方法上应用了该特性, 授权将被以检验代码访问安全的方式来执行。`PrincipalPermissionAttribute` 的 `Authenticated` 属性用于指定目标方法是否一定需要在认证用户环境下执行。而 `Name` 和 `Role` 表示执行目标方法所允许的用户名和角色。

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Class,
AllowMultiple=true)]
public sealed class PrincipalPermissionAttribute :
```

```

CodeAccessSecurityAttribute
{
    //其他成员
    public PrincipalPermissionAttribute(SecurityAction action);

    public bool Authenticated { get; set; }
    public string Name { get; set; }
    public string Role { get; set; }
}

```

从应用在 `PrincipalPermissionAttribute` 上的 `AttributeUsageAttribute` 可以看出该特性可以应用到类型和方法级别, 并且可以在同一个目标元素上应用多个特性。如果在同一个方法上应用了不止一个 `PrincipalPermissionAttribute` 特性, 那么只要定义在任何一个特性上的授权策略通过检验, 就意味着目标方法被授权了。

在下面的程序中, 我们创建了 4 个应用了 `PrincipalPermissionAttribute` 特性的测试方法。其中 `TestMethod1` 和 `TestMethod2` 上设置了不同的用户名 `Foo` 和 `Bar`。而 `TestMethod3` 和 `TestMethod4` 则设置了不同的角色, 前者设置了单一的角色 `Administrators`, 后者则设置了两个角色 `Administrators` 和 `Guests`。

4 个方法在 `Try/Catch` 中执行, 在方法执行之前一个 `GenericPrincipal` 对象被创建并设置成当前线程的安全主体。该 `GenericPrincipal` 安全身份是一个用户名为 `Foo` 的 `GenericIdentity`, 并且具有唯一的角色 `Guests`。通过最终的输出可以看出, 系统自动为我们完成的授权正式采用了定义于应用在目标方法上的 `PrincipalPermissionAttribute` 特性中的授权策略。

```

static void Main(string[] args)
{
    GenericIdentity identity = new GenericIdentity("Foo");
    Thread.CurrentPrincipal = new GenericPrincipal(identity, new string[]
    { "Guests" });
    Invoke(() => TestMethod1());
    Invoke(() => TestMethod2());
    Invoke(() => TestMethod3());
    Invoke(() => TestMethod4());
}

public static void Invoke(Action action)
{
    try
    {
        action();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

[PrincipalPermission(SecurityAction.Demand, Name = "Foo")]
public static void TestMethod1()
{
    Console.WriteLine("TestMethod1 方法被成功执行。");
}

[PrincipalPermission(SecurityAction.Demand, Name = "Bar")]
public static void TestMethod2()

```

```

{
    Console.WriteLine("TestMethod2 方法被成功执行。");
}
[PrincipalPermission(SecurityAction.Demand, Role="Administrators")]
public static void TestMethod3()
{
    Console.WriteLine("TestMethod3 方法被成功执行。");
}
[PrincipalPermission(SecurityAction.Demand, Role = "Administrators")]
[PrincipalPermission(SecurityAction.Demand, Role = "Guests")]
public static void TestMethod4()
{
    Console.WriteLine("TestMethod4 方法被成功执行。");
}

```

输出结果:

```

TestMethod1 方法被成功执行。
对主体权限的请求失败。
对主体权限的请求失败。
TestMethod4 方法被成功执行。

```

虽然从应用在 `PrincipalPermissionAttribute` 的 `AttributeUsageAttribute` 特性定义上看, `PrincipalPermissionAttribute` 可同时应用在类和方法上。但是当我们采用这个特性以声明的方式进行 WCF 服务授权的时候,却只能将 `PrincipalPermissionAttribute` 应用在服务操作方法上。

3. WCF 服务授权

如果在实施授权的时候,当前线程的安全主体能够被正确设置,就可以正确地完成授权。对于 WCF 的服务授权,如果正确的安全主体能够在服务操作被执行之前被正确设置到当前线程,借助于这个安全主体,不但可以采用命令式编程的方式将授权逻辑写在相应的操作中,也可以采用声明式编程的方式将授权策略定义在应用在服务操作方法上的 `PrincipalPermissionAttribute` 特性中。

安全主体具有身份与权限两个基本的要素。身份在客户端经过认证之后已经确立下来,现在需要解决的问题就是如何获取被认证用户的权限。为了解决这个问题, WCF 为我们提供了不同的方案,我们把这些方案称为不同的“安全主体权限模式 (Principal Permission Mode)”。WCF 支持如下三种安全主体权限模式。

- Windows 用户组: 将经过认证的用户映射为同名的 Windows 账号,将该账号所在的用户组作为权限集;
- ASP.NET Roles 提供程序: 通过 ASP.NET 角色管理机制借助于某个 `RoleProvider` 获取基于当前认证用户的角色列表,并将其作为权限集;
- 自定义权限模式: 自定义权限解析和安全主体创建机制。

上述三种模式通过具有如下定义的 `System.ServiceModel.Description.PrincipalPermissionMode` 枚举表示。

```
public enum PrincipalPermissionMode
{
    None,
    UseWindowsGroups,
    UseAspNetRoles,
    Custom
}
```

采用的安全主体权限模式决定了最终生成的安全主体的类型。之前我们介绍了 `WindowsPrincipal` 和 `GenericPrincipal`，而 `UseAspNetRoles` 模式对应的是另一种安全主体类型：`RoleProviderPrincipal`。

4. RoleProviderPrincipal

`RoleProviderPrincipal` 定义在 `System.ServiceModel.Security` 命名空间下，是基于 ASP.NET `RoleProvider` 授权模式下产生的安全主体。和 `X509Identity` 一样，`RoleProviderPrincipal` 也是定义在 `System.ServiceModel.dll` 程序集中的一个内部类型而已。下面的代码片段体现了 `RoleProviderPrincipal` 的定义。

```
internal sealed class RoleProviderPrincipal : IPrincipal
{
    public RoleProviderPrincipal(object roleProvider,
        ServiceSecurityContext securityContext);
    public bool IsInRole(string role);
    public IIdentity Identity { get; }
}
```

5. ServiceAuthorizationBehavior

WCF 的服务端框架根据当前分发运行时 (`DispatchRuntime`) 的 `PrincipalPermissionMode` 属性判断具体采用哪种安全主体权限模式。如果采用 `UseAspNetRoles` 模式，则通过 `RoleProvider` 属性得到用于获取角色列表的 `RoleProvider`。`PrincipalPermissionMode` 和 `RoleProvider` 在 `DispatchRuntime` 中的定义如下。

```
public sealed class DispatchRuntime
{
    //其他成员
    public PrincipalPermissionMode PrincipalPermissionMode { get; set; }
    public RoleProvider RoleProvider { get; set; }
}
```

分发运行时这两个属性是通过 `System.ServiceModel.Description.ServiceAuthorizationBehavior` 服务行为进行设置的。从下面的代码片段中可以看到 `PrincipalPermissionMode` 和 `RoleProvider` 两属性定义在 `ServiceAuthorizationBehavior` 中。定义在 `ServiceAuthorizationBehavior` 中的授权相关的设置最终通过 `ApplyDispatchBehavior` 方法被应用到所有终结点分发器 (`EndpointDispatcher`) 的分发运行时。


```

public sealed class ServiceAuthorizationBehavior : IServiceBehavior
{
    //其他成员
    void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription description,
        ServiceHostBase serviceHostBase);

    public PrincipalPermissionMode PrincipalPermissionMode { get; set; }
    public RoleProvider RoleProvider { get; set; }
}

```

8.2 Windows 用户组授权

Windows 用户组安全主体权限模式就是利用 Windows 安全系统将对应的 Windows 账号所在的用户组作为该用户权限集的授权方式。虽然认证和授权密不可分，但是对于认证和授权在 WCF 安全体系中的实现来说，它们却是相对独立的。认证属于安全传输的范畴，是在信道层实现的，而授权则是在服务模型层实现的。但是对于基于 Windows 用户组的授权来说，最终体现出来的授权行为却决定于采用的认证方式。

8.2.1 Windows 用户组授权与认证的关系

无论是对于 WCF 提供的两种原生的安全主体权限模式（基于 Windows 用户组和基于 ASP.NET Roles 提供程序），还是自定义安全主体权限模式，最终都体现在创建相应的安全主体，并将其附加到当前线程上。

对于 Windows 用户组模式来说，不论采用何种客户端凭证类型及认证模式，最终建立的安全主体都是一个身份类型为 `WindowsIdentity` 的 `WindowsPrincipal` 对象。但是它的身份能否正确地反映被认证后的用户，以及其本身能否正确反映该认证用户的权限，就和认证行为有密切的关系。

当你选择了 Windows 用户组安全主体权限模式后，只有在采用 Windows 认证的情况下最终生成的安全主体才能正确地反映被认证的用户。这里的 Windows 认证包括如下三种情况：

- 客户端凭证为 Windows 凭证；
- 客户端凭证为用户名/密码凭证，并采用 Windows 认证模式；
- 客户端凭证为 X.509 证书凭证，并允许与 Windows 账号进行映射。

在其他情况下，最终被创建的是一个“空”的 `WindowsPrincipal`。这个空的 `WindowsPrincipal` 不仅体现在具有一个“空”的权限集，而且其内部的 `WindowsIdentity` 也为“空”。在非 Windows 认证的情况下，即使存在一个与认证用户一致的 Windows 账号，WCF 授权系统也不会基于该 Windows 账号来创建最终的 `WindowsPrincipal`。

举个例子, 假设服务寄宿端所在的域中具有一个用户叫做“张三”, 并且存在于当前机器的管理员 (Administrators) 用户组中。现在我们对某个服务操作进行授权, 要求必须具有管理员权限才能被调用。在进行服务寄宿的时候, 终结点的绑定采用用户名/密码作为客户端凭证, 并选择 MembershipProvider 认证模式。在认证成功的情况下, 被授权的服务操作也是不能被正常调用的。

8.2.2 Windows 用户组授权编程

基于授权的编程基本上都是围绕着服务行为 ServiceAuthorizationBehavior 进行的。既然 ServiceAuthorizationBehavior 是一个服务行为, 我们只需要通过编程或者配置的方式将该服务行为添加到寄宿服务的行为列表中就可以了。可以按照下面的编程方式让寄宿的服务采用基于 Windows 用户组授权模式。

```
using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    ServiceAuthorizationBehavior behavior =
        host.Description.Behaviors.Find<ServiceAuthorizationBehavior>();
    if (null == behavior)
    {
        behavior = new ServiceAuthorizationBehavior();
        host.Description.Behaviors.Add(behavior);
    }
    behavior.PrincipalPermissionMode = PrincipalPermissionMode.UseWindowsGroups;
    host.Open();
    //...
}
```

也可以通过 ServiceHost 的只读属性 Authorization 得到这个 ServiceAuthorizationBehavior 行为对象。如下面的代码片段所示, 该属性实际上定义在 ServiceHost 的基类 ServiceHostBase 中。

```
public abstract class ServiceHostBase
{
    //其他成员
    public ServiceAuthorizationBehavior Authorization { get; }
}
```

在读取该属性的时候, 如果当前服务描述中的服务行为列表中找不到 ServiceAuthorizationBehavior 行为, 系统会自动创建一个自动 ServiceAuthorizationBehavior 对象并添加到服务行为列表中。所以上面的这段服务寄宿代码实际上和下面这段代码是完全等效的。

```
using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    Host.Authorization.PrincipalPermissionMode =
        PrincipalPermissionMode.UseWindowsGroups;
    host.Open();
    //...
}
```

依然推荐采用配置的方式进行授权模式的设置。下面一段配置和上面的代码在作用上也是等效的。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Services.CalculatorService"
        behaviorConfiguration="UseWindowsGroupsAuthorization">
        ...
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="UseWindowsGroupsAuthorization">
          <serviceAuthorization principalPermissionMode="UseWindowsGroups"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

为了让读者对基于 Windows 用户组的授权具有深刻的认识，接下来通过一个简单的实例来讲解在真正的应用中如何使用该授权模式。

8.2.3 实例演示：基于 Windows 用户组的声明式授权（S801）

对于接下来演示的实例，将采用 Windows 认证和授权。至于授权的最终实现，我们采用的是在服务方法上面应用 `PrincipalPermissionAttribute` 特性方式的声明式授权。

步骤一：创建测试账号

在创建实例解决方案之前先完成相应的准备工作，即创建两个测试用的 Windows 账号。假设两个账号的名称分别为 Foo 和 Bar，密码为 Password。然后将账号 Foo 添加到管理员（Administrators）用户组中。

步骤二：创建服务契约和服务

依然沿用我们再熟悉不过的计算服务的例子，解决方案依然按照如图 8-1 所示的结构来设计。整个解决方式包括三个项目。对于这样的结构我们已经了解得够多了，在这里没有必要再赘述了。

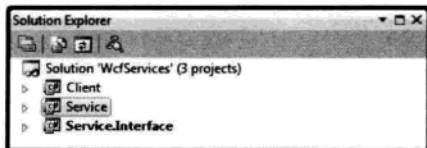


图 8-1 Windows 用户组授权实例解决方案

我们分别在 `Service.Interface` 和 `Service` 项目中定义服务契约接口和服务类型。下面是契约接口 `ICalculator` 和服务 `CalculatorService` 的定义。而在 `CalculatorService` 类的 `Add` 方法中应用了 `PrincipalPermissionAttribute` 特性, 并将 `Roles` 属性设置成了 `Administrators`, 意味着该服务操作只能被管理员用户组中的用户调用。

ICalculator:

```
using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface ICalculator
    {
        [OperationContract]
        double Add(double x, double y);
    }
}
```

CalculatorService:

```
using System.Security.Permissions;
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Service
{
    public class CalculatorService : ICalculator
    {
        [PrincipalPermission(SecurityAction.Demand, Role = "Administrators")]
        public double Add(double x, double y)
        {
            return x + y;
        }
    }
}
```

步骤三: 寄宿服务

现在通过控制台程序对上面创建的服务进行寄宿, 下面给出的是整个寄宿程序的配置。从该配置中可以看到, 服务唯一的终结点采用的绑定类型为 `WS2007HttpBinding`。而在默认的情况下, `WS2007HttpBinding` 采用 `Message` 安全模式和 `Windows` 认证方式。基于 `UseWindowsGroups` 安全主体权限模式的 `ServiceAuthorization` 服务行为被应用到了该服务上。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService"
        behaviorConfiguration="useWindowsGroupsAuthorization">
        <endpoint address="http://127.0.0.1:3721/calculatorservice"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.ICalculator"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

```

<behaviors>
  <serviceBehaviors>
    <behavior name="useWindowsGroupsAuthorization">
      <serviceAuthorization principalPermissionMode="UseWindowsGroups"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

而服务寄宿的程序依然简洁如故，仅仅包括针对寄宿服务类型的 `ServiceHost` 的创建和开启而已。

```

using System.ServiceModel;
using Artech.WcfServices.Service;
using System;
namespace Artech.WcfServices.Service
{
    public class Program
    {
        static void Main(string[] args)
        {
            using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
            {
                host.Open();
                Console.Read();
            }
        }
    }
}

```

步骤四：创建客户端程序

我们将服务调用的客户程序定义在 `Client` 项目中。整个实例演示的目的在于确认针对服务操作 `Add` 的授权是根据 Windows 用户组进行的，我们只需要关注被授权的服务操作是否被成功调用。为此我写了如下一个简单的辅助性的方法 `Invoke`。如果服务操作被成功执行，则输出“服务调用成功”，如果抛出异常则输出“服务调用失败”。

```

static void Invoke(ICalculator calculator)
{
    try
    {
        calculator.Add(1, 2);
        Console.WriteLine("服务调用成功...");
    }
    catch
    {
        Console.WriteLine("服务调用失败...");
    }
}

```

下面演示了完整的客户端程序和相应的配置。整个程序体现了两次针对相同服务操作的调用，而两次服务调用采用的客户端凭证分别是基于之前创建的两个 Windows 账号 `Foo` 和 `Bar`。

客户端程序:

```
ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService");
NetworkCredential credential = channelFactory.Credentials.Windows.Client
Credential;
credential.UserName = "Foo";
credential.Password = "Password";
ICalculator calculator = channelFactory.CreateChannel();
Invoke(calculator);

channelFactory = new ChannelFactory<ICalculator>("calculatorService");
credential = channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Bar";
credential.Password = "Password";
calculator = channelFactory.CreateChannel();
Invoke(calculator);
```

配置:

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="calculatorService"
        address="http://127.0.0.1:3721/calculatorService"
        binding="ws2007HttpBinding"
        contract="Artech.WcfServices.Service.Interface.ICalculator"/>
    </client>
  </system.serviceModel>
</configuration>
```

由于调用的服务操作需要具有管理员权限才能调用,所以以 Foo 名义进行调用是没有问题的。但是对于账号 Bar,由于权限不足,调用将会失败。而客户端输出的结果反映了这一点。

服务调用成功...

服务调用失败...

对于这个实例来说,服务操作只有具有管理员权限方能被正常调用。虽然我们创建的 Windows 账号 Foo 在管理员用户组中,但是如果使用 Windows Vista、Windows Server 2008 和 Windows 7 这三种操作系统,在 UAC 开启的情况下,即使以管理员身份运行我们的演示程序, Foo 也不具有管理员权限。所以需要关闭 UAC 才能得到正确的执行结果,否则两次调用都会输出“服务调用失败...”。

8.2.4 身份模拟 (Impersonation)

由于服务操作是在寄宿进程中执行的,在默认的情况下,服务操作是否具有足够的权限访问某个资源(比如文件)决定于执行寄宿进程的 Windows 账号的权限设置,而与作为客户端的 Windows 账号无关。在很多情况下,我们希望服务操作执行在基于客户端的安全上下文中,以解决执行服务的账号权限不足的问题。这就涉及身份模拟 (Impersonation) 的话题。

1. 从访问令牌 (Access Token) 说起

当我们以某个 Windows 账号的名义成功登录 Windows 的时候, 操作系统会创建一个访问令牌 (Access Token)。访问令牌不仅仅包括登录用户身份相关信息, 还包括该用户具有的权限信息 (比如 ACL)。当我们开启某个进程的时候, 该访问令牌会自动附加到该进程上, 作为其安全上下文重要的组成部分。Windows 下的访问令牌主要具有如下两种形式。

- **主令牌 (Primary Token):** 每一个进程都具有一个唯一的主令牌, 进程被主令牌被开启。
- **模拟令牌 (Impersonation Token):** 在默认的情况下, 当线程被开启的时候, 所在进程的主令牌会自动附加到当前线程上, 作为线程的安全上下文。而线程可以运行在另一个非主令牌 of 的访问令牌下执行, 这个令牌被称为模拟令牌。指定线程的模拟令牌的过程被称为模拟。

我们可以调用 Win32 函数 LogonUser, 通过输入 Windows 账号、密码、域名及认证相关信息创建一个访问令牌。LogonUser 被调用的时候, 会试图进行基于本机的登录操作。访问令牌会在认证成功认证的情况下被创建并返回。

LogonUser 函数的定义如下, 其输入参数依次代表的含义分别是用户名、域名 (可选参数)、密码 (明文)、登录类型和登录提供者。创建的访问令牌以输出参数 (phToken) 的形式返回。

```

BOOL LogonUser(
    __in     LPTSTR    lpszUsername,
    __in_opt LPTSTR    lpszDomain,
    __in     LPTSTR    lpszPassword,
    __in     DWORD     dwLogonType,
    __in     DWORD     dwLogonProvider,
    __out    PHANDLE   phToken
);

```

当我们为进行模拟而试图创建一个模拟令牌的时候, 需要用到另外一个重要的 Win32 函数 DuplicateToken。该函数通过一个现有的令牌来“复制”一个新的令牌。DuplicateToken 函数的定义如下面的代码片段所示。

```

BOOL WINAPI DuplicateToken(
    __in     HANDLE ExistingTokenHandle,
    __in     SECURITY_IMPERSONATION_LEVEL ImpersonationLevel,
    __out    PHANDLE DuplicateTokenHandle
);

```

从上面给出的 DuplicateToken 函数的定义可以看出, 除了传入现有的访问令牌作为输入参数之外, 还具有一个表示模拟级别的 ImpersonationLevel 的参数。模拟级别是一个非常重要的概念, 它表示被复制的模拟令牌可以被使用的程度和范围。模拟等级通过一个具有如下定义的枚举 SECURITY_IMPERSONATION_LEVEL 表示。

```

typedef enum _SECURITY_IMPERSONATION_LEVEL {
    SecurityAnonymous,
    SecurityIdentification,

```

```

SecurityImpersonation,
SecurityDelegation
} SECURITY_IMPERSONATION_LEVEL, *PSECURITY_IMPERSONATION_LEVEL;

```

四个枚举项 (SecurityAnonymous、SecurityIdentification、SecurityImpersonation 和 SecurityDelegation) 分别代表如下 4 种模拟的等级。

- 匿名 (Anonymous): 无法获取有关客户端的标识信息, 且无法模拟客户端;
- 识别 (Identification): 可以获取有关客户端的识别信息 (如安全标识符和特权), 但是无法模拟客户端;
- 模拟 (Impersonation): 可以在本地模拟客户端的安全上下文, 但无法在远程系统上模拟客户端;
- 委托 (Delegation): 可以在本地和远程系统上模拟客户端的安全上下文。

对于模拟级别, 在托管代码的应用编程接口中也具有一个匹配的枚举 System.Security.Principal.TokenImpersonationLevel。如下面的代码片段所示, 除了上述的 4 个对应于具体模拟级别的枚举项之外, TokenImpersonationLevel 还有一个额外的枚举值 None, 表示具体的模拟级别尚未指定。

```

public enum TokenImpersonationLevel
{
    None,
    Anonymous,
    Identification,
    Impersonation,
    Delegation
}

```

2. 再谈 WindowsIdentity

关于模拟在托管代码的实现, 不得不提到我们之前介绍过的类型 WindowsIdentity。从某种意义上讲, 一个 WindowsIdentity 对象可以看成是对一个访问令牌的封装, 并且直接为我们提供了模拟的功能。从下面给出的 WindowsIdentity 部分成员定义可以看到, 它具有一个 IntPtr 类型的只读属性 Token, 实际上代表的就是我们上面介绍的访问令牌。而我们可以通过直接指定这个访问令牌创建一个 WindowsIdentity 对象。ImpersonationLevel 表示访问令牌的模拟级别。

```

public class WindowsIdentity : IIdentity,...
{
    //其他成员
    public WindowsIdentity(IntPtr userToken);
    public virtual IntPtr Token { get; }
    public TokenImpersonationLevel ImpersonationLevel { get; }

    public virtual WindowsImpersonationContext Impersonate();
    public static WindowsImpersonationContext Impersonate(IntPtr userToken);
}

```


对于一个现有的 `WindowsIdentity` 对象，只要其访问令牌的模拟级别为 `Impersonation` 或者 `Delegation`，我们就能调用 `Impersonate` 方法模拟这个身份。`WindowsIdentity` 还提供了静态的 `Impersonate` 使我们可以直接根据一个访问令牌实施身份模式。

一旦 `Impersonate` 方法被调用，基于被模拟身份的安全上下文会自动附加到当前线程。`Impersonate` 方法返回的是一个具有如下定义的 `System.Security.Principal.WindowsImpersonationContext` 对象。如果需要将安全上下文恢复到模拟之前的状态，可以调用 `WindowsImpersonationContext` 的 `Undo` 方法。而 `WindowsImpersonationContext` 实现了 `IDisposable` 接口，`Undo` 方法实际上也会在 `Dispose` 方法中被调用。

```
public class WindowsImpersonationContext : IDisposable
{
    //其他成员
    public void Dispose();
    public void Undo();
}
```

一般来说，只有某些特殊的操作（比如访问一个受权限控制的文件）才需要在被模拟的身份下执行。如果这些操作执行完毕或者在执行过程中抛出异常，我们都需要恢复线程安全上下文到被模拟之前的状态。所以正确的模拟编程应该采用如下的方式。

```
WindowsImpersonationContext impersonationContext = identity.Impersonate();
try
{
    //在模拟身份下执行的操作
}
catch (Exception ex)
{
    //异常处理
}
finally
{
    impersonationContext.Undo();
}
```

或者

```
using (WindowsImpersonationContext impersonationContext =
identity.Impersonate())
{
    //在模拟身份下执行的操作
}
```

3. 实例演示：通过身份模拟的方式读取文件（S802）

为了让读者对身份模式的作用和实现有深刻的认识，我们来演示一个简单的实例。在这个实例中，我们将通过 ACL 设置一个文件的读取权限，然后演示针对不同 Windows 账号进行模拟的情况下，是否能够正常读取该文件。

在之前的实例演示中，我们创建了两个本机的 Windows 账户 Foo 和 Bar（密码为

Password)。如果这两个账号尚未创建，需要现在创建。然后在某个目录下（比如 D:盘）创建一个简单的文本文件（比如 impersonationTest.txt）。然后赋予账号 Foo 对该文件的读取权限，但拒绝账号 Bar 读取该文件。

然后我们创建一个简单的控制台应用作为演示程序。先定义如下一个用于创建 WindowsIdentity 的 CreateWindowsIdentity 静态方法。该方法通过输入用户名、密码和模拟级别创建相应的 WindowsIdentity。在 CreateWindowsIdentity 方法内部实际上是直接调用了上面介绍的两个 Win32 函数 LogonUser 和 DuplicateToken。

```
class Program
{
    //其他成员
    public const int LOGON32_PROVIDER_DEFAULT = 0;
    public const int LOGON32_LOGON_INTERACTIVE = 2;

    [DllImport("advapi32.dll", SetLastError = true)]
    public static extern bool LogonUser(string userName, string domainName,
        string password, int logonType, int logonProvider, ref IntPtr token);

    [DllImport("advapi32.dll", CharSet = CharSet.Auto, SetLastError = true)]
    public static extern bool DuplicateToken(IntPtr existingToken,
        int impersonationLevel, ref IntPtr newToken);

    public static WindowsIdentity CreateWindowsIdentity(string userName,
        string password, TokenImpersonationLevel tokenImpersonationLevel)
    {
        IntPtr token = IntPtr.Zero;
        IntPtr duplicateToken = IntPtr.Zero;
        if (LogonUser(userName, string.Empty, password,
            LOGON32_LOGON_INTERACTIVE, LOGON32_PROVIDER_DEFAULT, ref token))
        {
            int impersonationLevel;
            switch (tokenImpersonationLevel)
            {
                case TokenImpersonationLevel.Anonymous:
                    { impersonationLevel = 0; break; }
                case TokenImpersonationLevel.Impersonation:
                    { impersonationLevel = 2; break; }
                case TokenImpersonationLevel.Delegation:
                    { impersonationLevel = 3; break; }
                default:
                    { impersonationLevel = 1; break; }
            }
            if (DuplicateToken(token, impersonationLevel, ref duplicateToken))
            {
                return new WindowsIdentity(duplicateToken);
            }
            else
            {
                throw new InvalidOperationException(string.Format(
                    "创建模拟令牌失败 (错误代码: {0})", Marshal.GetLastWin32Error()));
            }
        }
        else
        {

```

```

        throw new InvalidOperationException(
            string.Format("用户登录失败 (错误代码: {0})", Marshal.GetLastWin32Error()));
    }
}

```

然后定义如下一个静态的 `ReadFile` 方法。在这个方法中, 根据传入的用户名和密码调用上述的 `CreateWindowsIdentity` 方法创建相应的 `WindowsIdentity`。然后模拟该用户进行文件的读取并在成功读取和抛出异常的情况下分别输出相应的指示性文字。在 `Main` 方法中分别传入账号 `Foo` 和 `Bar` 及相应的密码对该方法进行调用。从输出的结果可以看出, 在模拟账号 `Foo` 时文件被成功读取, 而在模拟账号 `Bar` 的时候却失败了。这和测试文件的 ACL 设置是一致的。

```

public static void ReadFile(string userName, string password)
{
    try
    {
        WindowsIdentity identity = CreateWindowsIdentity(userName, password,
            TokenImpersonationLevel.Impersonation);
        using (WindowsImpersonationContext context = identity.Impersonate())
        {
            Console.WriteLine("当前用为: {0}", WindowsIdentity.GetCurrent().Name);
            File.ReadAllText("D:\\impersonationTest.txt");
            Console.WriteLine("成功读取文件内容...");
        }
    }
    catch
    {
        Console.WriteLine("读取文件失败...");
    }
}

static void Main(string[] args)
{
    ReadFile("Foo", "Password");
    ReadFile("Bar", "Password");
}

```

输出结果 (Jinnan-PC 为我的本机的机器名):

```

当前用户为: Jinnan-PC\Foo
成功读取文件内容...
当前用户为: Jinnan-PC\Bar
读取文件失败...

```

4. 在服务操作中实现身份模拟

如果我们有一个具有模拟级别为 `Impersonation` 或者 `Delegation` 的 `WindowsIdentity`, 就可以通过调用其 `Impersonate` 对其进行身份模拟。在采用 Windows 认证的情况下, 我们可以通过当前 `ServiceSecurityContext` 的 `WindowsIdentity` 或者 `PrimaryIdentity` 获取到代表认证客户端的 `WindowsIdentity` 对象。

```
using (WindowsImpersonationContext context =
    ServiceSecurityContext.Current.WindowsIdentity.Impersonate())
{
    //在模拟上下文中执行的操作
}
```

这种在服务操作实现中通过编程的方式实现的身份模式可以将服务操作的部分逻辑在模拟的客户端身份下执行。如果希望整个操作都在模拟上下文中执行,可以采用声明式的模拟编程。只需要在需要进行模拟的服务操作方法上应用 `OperationBehaviorAttribute` 特性,并指定相应的模拟选项即可。

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute,
    IOperationBehavior
{
    //其他成员
    public ImpersonationOption Impersonation { get; set; }
}
public enum ImpersonationOption
{
    NotAllowed,
    Allowed,
    Required
}
```

通过 `OperationBehaviorAttribute` 的 `Impersonation` 属性指定模拟选项通过枚举 `System.ServiceModel.ImpersonationOption` 表示。定义在 `ImpersonationOption` 中的三个枚举项 `NotAllowed`、`Allowed` 和 `Required` 分别表示的含义如下。

- `NotAllowed`: 不允许模拟客户端身份,这是默认值。
- `Allowed`: 在身份模拟条件满足的情况下允许模拟客户端身份。
- `Required`: 强制模拟客户端身份。这要求强制采用 Windows 认证,对于非 Windows 认证会抛出异常。

如果你要求服务的所有操作均强制采用身份模拟,可以通过编程或者配置将 `ServiceAuthorizationBehavior` 服务行为的 `ImpersonateCallerForAllOperations` 属性设置成 `true`。但是在这种情况下,如果该服务有任何模拟选项被设置成 `NotAllowed` 的服务操作,服务在寄宿过程中会抛出 `InvalidationOperationException` 异常。

```
public sealed class ServiceAuthorizationBehavior : IServiceBehavior
{
    //其他成员
    public bool ImpersonateCallerForAllOperations { get; set; }
}
```

5. 设置模拟级别

在采用 Windows 认证的情况下,服务在成功认证后可以获取代表客户端身份的 `WindowsIdentity` 对象。但是服务端是否可以根据 `WindowsIdentity` 获取客户端身份信息,是

否可以模拟客户端身份访问某些安全资源，取决于该 `WindowsIdentity` 的模拟级别。

身份模拟密切地关系到被模拟身份代表的用户的安全，所以模拟级别应该通过客户端自行控制。在 WCF 安全体系中，该模拟级别是在客户端提供的 `Windows` 凭证中指定的。如下面的代码所示，表示客户端 `Windows` 凭证的 `WindowsClientCredential` 类型中，具有一个类型为 `TokenImpersonationLevel` 枚举的 `AllowedImpersonationLevel` 属性，用于指定模拟级别。该属性的默认值为 `None`，实际上代表的等级是 `Identification`。

```
public sealed class WindowsClientCredential
{
    //其他成员
    public TokenImpersonationLevel AllowedImpersonationLevel { get; set; }
}
```

8.3 ASP.NET Roles 授权

在采用 `Windows` 认证的情况下，使用基于 `Windows` 用户组安全主体权限模式是一个不错的选择。我们可以直接使用现有的用户组设置，也可以为相应的应用或服务创建单独的用户组。但是该模式对 `Windows` 认证的依赖，意味着这种模式只能用于局域网环境中。如果采用证书和 `Windows` 账号的映射，也可以适用于像 B2B 这样的外部网环境。在其他的网络环境中，基于 `Windows` 用户组的授权方式将会无能为力。

还有这样一种状况，即使是在同一个局域网环境中，也采用 `Windows` 进行客户端认证。但是我们不想创建太多的 `Windows` 用户组，而是将用户的权限信息维护在相应的数据库中，通过单独的安全系统来维护。在这种情况下，基于 `ASP.NET` 角色管理模块的授权模式是一个不错的选择。

8.3.1 ASP.NET Roles 提供程序

和 `Membership` 一样，`Roles` 也是 `ASP.NET` 的一个重要的提供程序，旨在解决对角色的维护和基于角色的授权。`ASP.NET Roles` 同样采用策略设计模式，角色的添加、删除、获取及授权功能定义在 `System.Web.Security.RoleProvider` 这个抽象类中。而 `ASP.NET` 默认提供了如下三个具体的 `RoleProvider`。如果它们还不能满足你的具体授权要求，还可以自定义 `RoleProvider`。比如说，如果你使用的数据库是 `Oracle`，可以参考 `SqlRoleProvider` 自定义一个 `OracleRoleProvider`。

- `System.Web.Security.SqlRoleProvider`：将角色和授权信息存储于 `SQL Server` 数据库预定义的表中。
- `System.Web.Security.WindowsTokenRoleProvider`：直接使用 `Windows` 用户组进行授权，但是角色（用户组）的添加和删除操作是不允许的。

- System.Web.Security.AuthorizationStoreRoleProvider: 使用 AuthorizationManager (AzMan) 库作为角色存储。

ASP.NET Roles 相关的功能基本上都可以通过调用 System.Web.Security.Roles 这个静态类的相应方法来完成。下面的代码片段列出了 Roles 的主要方法。其中 CreateRole 和 DeleteRole 用于进行角色创建和删除。RoleExists 用于确定指定的角色是否存在, 而 AddUser(s)ToRole(s) 和 RemoveUser(s)FromRole(s) 则用于建立和解除用户和角色的关系。IsUserInRole 用于确定指定的用户具有相应的角色。

```
public static class Roles
{
    //其他成员
    public static void CreateRole(string roleName);
    public static bool DeleteRole(string roleName);
    public static bool DeleteRole(string roleName, bool throwOnPopulatedRole);
    public static bool RoleExists(string roleName);

    public static void AddUsersToRole(string[] usernames, string roleName);
    public static void AddUsersToRoles(string[] usernames, string[] roleNames);
    public static void AddUserToRole(string username, string roleName);
    public static void AddUserToRoles(string username, string[] roleNames);

    public static void RemoveUserFromRole(string username, string roleName);
    public static void RemoveUserFromRoles(string username, string[] roleNames);
    public static void RemoveUsersFromRole(string[] usernames, string roleName);
    public static void RemoveUsersFromRoles(string[] usernames, string[] roleNames);

    public static bool IsUserInRole(string roleName);
    public static bool IsUserInRole(string username, string roleName);
}
```

Roles 的方法 AddUser(s)ToRole(s) 并不会进行用户账号存在与否的验证。因为用户账号的管理属于 Membership 的范畴, 而建立用户与角色的关系才是角色管理需要负责的。Membership 和 Roles 对于 ASP.NET 是相互独立的两个提供程序, 它们不具有任何依赖关系。

完全可以采用 ActiveDirectoryMembershipProvider 利用 AD 进行用户账号管理和认证, 而将角色维护在基于 SqlRoleProvider 的 SQL Server 数据表中。在这种情况下, 当我们调用 Roles 的 AddUser(s)ToRole(s) 方法的时候, 指定的用户账号在数据库中是不存在的。所以 Roles 不会进行用户存在与否的验证, 它只是负责将指定的用户名添加到相应的角色之中而已。认证和授权的这种独立性同样体现在 WCF 上。

8.3.2 ASP.NET Roles 授权与认证的无关性

Windows 用户组授权依赖于 Windows 认证, 但是如果采用了 ASP.NET Roles 安全主体权限模式, 可以采用任何非匿名客户端凭证和认证方式。ASP.NET Roles 模式真正体现了认证和授权的无关性。

在采用 ASP.NET Roles 安全主体权限模式下，最终创建并作为当前线程安全主体的是一个 `RoleProviderPrincipal` 对象。而该 `RoleProviderPrincipal` 的 `Identity` 与当前 `ServiceSecurityContext` 的 `PrimaryIdentity` 属性实际上是同一个对象。两者的统一性可以通过如下的验证程序来体现。

```

IIdentity identity1 = Thread.CurrentPrincipal.Identity;
IIdentity identity2 = ServiceSecurityContext.Current.PrimaryIdentity;
Debug.Assert(object.ReferenceEquals(identity1, identity2));

```

原则上只要被认证的用户名能够通过 ASP.NET Roles 正确获取到反映权限的角色列表，授权就能顺利进行。

- Windows 认证（包括之前提到的三种情况）：需要针对 Windows 账号（域名/用户名）进行角色分配。
- Membership 和 Custom 的用户名/密码认证：直接针对用户名角色的分配。
- 证书凭证（并不允许 Windows 账号映射）：被认证的用户名是证书主体名称和指纹的组合（<<主题名称>>; <<指纹>>），需要以此进行权限（角色）的设置。

8.3.3 ASP.NET Roles 授权编程

所有基于安全主体授权的编程都体现在 `ServiceAuthorizationBehavior` 这个服务行为上。如果要让 WCF 采用 ASP.NET Roles 进行授权，需要将 `ServiceAuthorizationBehavior` 的 `PrincipalPermissionMode` 属性设置成 `PrincipalPermissionMode.UseAspNetRoles`，并为其指定一个具体的 `RoleProvider` 即可。

至于 `RoleProvider` 的获取，可以通过 `Roles` 的 `Provider` 得到默认的 `RoleProvider`。此外 `Roles` 还具有一个类似于字典类型的 `Providers` 属性返回所有配置的 `RoleProvider` 列表，可以通过传入配置名称获取相应的 `RoleProvider`。

```

public sealed class ServiceAuthorizationBehavior : IServiceBehavior
{
    //其他成员
    public PrincipalPermissionMode PrincipalPermissionMode { get; set; }
    public RoleProvider RoleProvider { get; set; }
}
public static class Roles
{
    //其他成员
    public static RoleProvider Provider { get; }
    public static RoleProviderCollection Providers { get; }
}

```

下面给出的是一段基于自我寄宿的代码。在开启 `ServiceHost` 之前，我们为服务指定了一个 `ServiceAuthorizationBehavior` 行为，并将其安全主体权限模式设置成 `PrincipalPermissionMode.UseAspNetRoles`，它使用当前配置的默认 `RoleProvider`。

```

using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    ServiceAuthorizationBehavior behavior =
        host.Description.Behaviors.Find<ServiceAuthorizationBehavior>();
    if (null == behavior)
    {
        behavior = new ServiceAuthorizationBehavior();
        host.Description.Behaviors.Add(behavior);
    }
    behavior.PrincipalPermissionMode = PrincipalPermissionMode.UseAspNetRoles;
    behavior.RoleProvider = Roles.Provider
    host.Open();
    //...
}

```

我们还是一如既往地推荐采用配置的方式进行服务授权的设置。在下面这段配置中，我们在<system.web>/<roleManager>节点下配置了一个唯一的 SqlRoleProvider。该 SqlRoleProvider 的配置名称为 sqlRoleProvider，而目标数据库对应的连接字符串名称为 aspNetDb。ServiceAuthorizationBehavior 配置在名称为 aspNetRolesAuthorization 的服务行为配置节中，反映其安全主体权限模式的 principalPermissionMode 属性被设置成 UseAspNetRoles，而 roleProviderName 属性则正是配置的 RoleProvider 的名称。

```

<configuration>
  <connectionStrings>
    <add name="aspNetDb"
          connectionString="..."
          providerName="System.Data.SqlClient"/>
  </connectionStrings>
  <system.web>
    <roleManager enabled="true"
                  defaultProvider="SqlRoleProvider">
      <providers>
        <add name="sqlRoleProvider"
              type="System.Web.Security.SqlRoleProvider, System.Web"
              connectionStringName="aspNetDb"
              applicationName="AspRolesAuthorizationDemo"/>
      </providers>
    </roleManager>
  </system.web>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration="aspNetRolesAuthorization" ...>
        ...
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="aspNetRolesAuthorization">
          <serviceAuthorization principalPermissionMode="UseAspNetRoles"
                                roleProviderName="sqlRoleProvider"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```


8.3.4 实例演示：不同认证方式下的 ASP.NET Roles 授权

为了让读者对基于 ASP.NET Roles 的授权方式有一个全面的认识，现在来做一个实例演示。在这个实例中，将采用不同的认证方式，包括 Windows 认证和证书认证（ASP.NET Membership + Roles 为常见的组合方式，在这里不多做演示）。我们直接沿用在 8.2.3 节已经创建好的解决方案，并且依然采用声明式的授权。在服务操作方法 Add 上通过应用 `PrincipalPermissionAttribute` 特性指定其被授权的角色 Administrators。

```
public class CalculatorService : ICalculator
{
    [PrincipalPermission(SecurityAction.Demand, Role = "Administrators")]
    public double Add(double x, double y)
    {
        return x + y;
    }
}
```

我们具体采用的 RoleProvider 为 `SqlRoleProvider`，所以首先需要做的就是创建相应的数据库。ASP.NET 所有提供程序（比如 Membership、Roles、Profile 和 Site Map 等）所用的数据库的初始化工作都可以通过 `aspnet_regsql.exe` 这个工具来生成。由于在第 7 章演示 ASP.NET Membership 认证的时候已经对如何使用这个工具有过详细的说明了，因此这里就不再进行重复的介绍了。

创建了数据库之后，需要在 `aspnet_Applications` 表中插入一条记录，以表示我们即将演示的应用。可以直接执行如下的一段 SQL 脚本，在该脚本中我们为演示应用取名为 `AspRolesAuthorizationDemo`。

```
INSERT INTO [aspnet_Applications]
    ([ApplicationName]
    ,[LoweredApplicationName]
    ,[ApplicationId]
    ,[Description])
VALUES
    (
        'AspRolesAuthorizationDemo'
        , 'asprolesauthorizationdemo '
        , NEWID()
        , ''
    )
```

1. 在 Windows 认证下使用 ASP.NET Roles 授权（S803）

我们授权演示的是在客户端凭证类型为 Windows 的情况下采用 ASP.NET Roles 授权模式，需要更新一下服务端和客户端的配置。注意不要忘了根据实际情况修正连接字符串。

服务端配置：

```
<configuration>
  <connectionStrings>
```

```

<add name="aspNetDb"
      connectionString="..."
      providerName="System.Data.SqlClient"/>
</connectionStrings>
<system.web>
  <roleManager enabled="true" defaultProvider="sqlRoleProvider">
    <providers>
      <add name="sqlRoleProvider"
            type="System.Web.Security.SqlRoleProvider, System.Web"
            connectionStringName="AspNetDb"
            applicationName="AspNetRolesAuthorizationDemo"/>
    </providers>
  </roleManager>
</system.web>
<system.serviceModel>
  <services>
    <service name="Artech.WcfServices.Service.CalculatorService"
              behaviorConfiguration="useAspNetRoles">
      <endpoint address="http://127.0.0.1:3721/calculatorservice"
                binding="ws2007HttpBinding"
                contract="Artech.WcfServices.Service.Interface.ICalculator"/>
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="useAspNetRoles">
        <serviceAuthorization principalPermissionMode="UseAspNetRoles"
                              roleProviderName="sqlRoleProvider"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
</configuration>

```

客户端配置:

```

<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="calculatorService"
                address="http://127.0.0.1:3721/calculatorservice"
                binding="ws2007HttpBinding"
                contract="Artech.WcfServices.Service.Interface.ICalculator"/>
    </client>
  </system.serviceModel>
</configuration>

```

在这之前我们已经创建了两个 Windows 账号 Foo 和 Bar, 密码为 Password。如果这两个 Windows 账号尚未创建, 需要自行创建。由于我们现在是采用 ASP.NET Roles 进行授权, 因此需要通过 Roles 这个静态类为它们分配相应的权限。我直接将相应的实现写在如下所示的服务寄宿程序中。在这段代码中, 如果 Administrators 角色不存在, 先创建它, 并将其分配给用户 Jinnan-PC\Foo (Jinnan-PC 为我的机器名, 对于域账号, 用域名替换)。

```

if (!Roles.RoleExists("Administrators"))
{
    Roles.CreateRole("Administrators");
}

```

```

}
if(!Roles.IsUserInRole(@"Jinnan-PC\Foo", "Administrators"))
{
    Roles.AddUserToRole(@"Jinnan-PC\Foo", "Administrators");
}
using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    host.Open();
    Console.Read();
}

```

然后客户端分别以 Foo 和 Bar 的名义进行两次服务调用。由于 Foo 在服务启动之初就已经分配了 Administrators 角色，而 Bar 没有，因此只有第一次服务调用能够成功。而最终的执行结果也印证了这一点。

```

ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService");
NetworkCredential credential =
channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Foo";
credential.Password = "Password";
ICalculator calculator = channelFactory.CreateChannel();
Invoke(calculator);

channelFactory = new ChannelFactory<ICalculator>("calculatorService");
credential = channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Bar";
credential.Password = "Password";
calculator = channelFactory.CreateChannel();
Invoke(calculator);

```

输出结果：

```

服务调用成功...
服务调用失败...

```

2. 在 X.509 证书认证下使用 ASP.NET Roles 授权 (S804)

接下来演示客户端使用 X.509 证书的情况下如何使用 ASP.NET Roles 授权。为此需要通过如下的命令行创建三个主题名称 (CN) 分别为 Jinnan-PC (笔者的机器名称)、Foo 和 Bar 的证书。第一个作为服务证书，后两个作为客户端证书。它们都自动保存到本机 (LocalMachine) 的个人证书存储区。然后我们利用 MMC 的证书管理单元将 Foo 和 Bar 两个证书导入“受信任人 (Trusted People)”证书存储区。

```

MakeCert -n "CN=Jinnan-PC" -sr LocalMachine -ss My -pe -sky exchange
MakeCert -n "CN=Foo" -sr LocalMachine -ss My -pe -sky exchange
MakeCert -n "CN=Bar" -sr LocalMachine -ss My -pe -sky exchange

```

为了采用 X.509 证书作为客户端凭证，我们需要修改服务端和客户端的配置。在服务端配置中，不仅仅通过服务行为进行基于 ASP.NET Roles 授权相应的设置，还为服务设置了服务证书 (Jinnan-PC)，以及针对证书的认证模式 (PeerOrChainTrust)。而客户端则将服务证书的认证模式设为 None。

服务端配置:

```
<configuration>
  <connectionStrings>
    <add name="AspNetDb"
      connectionString="..."
      providerName="System.Data.SqlClient"/>
  </connectionStrings>
  <system.web>
    <roleManager enabled="true" defaultProvider="SqlRoleProvider">
      <providers>
        <add name="sqlRoleProvider"
          type="System.Web.Security.SqlRoleProvider, System.Web"
          connectionStringName="AspNetDb"
          applicationName="AspNetRolesAuthorizationDemo"/>
      </providers>
    </roleManager>
  </system.web>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="certificateCredentialBinding">
          <security mode="Message">
            <message clientCredentialType="Certificate"/>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService"
        behaviorConfiguration="useAspNetRoles">
        <endpoint address="http://127.0.0.1:3721/calculatorservice"
          binding="ws2007HttpBinding"
          bindingConfiguration="certificateCredentialBinding"
          contract="Artech.WcfServices.Service.Interface. ICalculator"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="useAspNetRoles">
          <serviceAuthorization principalPermissionMode="UseAspNetRoles"
            roleProviderName="sqlRoleProvider"/>
          <serviceCredentials>
            <serviceCertificate storeLocation="LocalMachine"
              storeName="My"
              x509FindType="FindBySubjectName"
              findValue="Jinnan-PC"/>
            <clientCertificate>
              <authentication certificateValidationMode="PeerOrChainTrust"/>
            </clientCertificate>
          </serviceCredentials>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

客户端:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="certificateCredentialBinding">
          <security mode="Message">
            <message clientCredentialType="Certificate"/>
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <client>
      <endpoint name="calculatorService"
        behaviorConfiguration="ignoreCertValidation"
        address="http://127.0.0.1:3721/calculatorservice"
        binding="ws2007HttpBinding"
        bindingConfiguration="certificateCredentialBinding"
        contract="Artech.WcfService.Service.Interface.ICalculator">
        <identity>
          <certificateReference storeLocation="LocalMachine"
            storeName="My"
            x509FindType="FindBySubjectName"
            findValue="Jinnan-PC"/>
        </identity>
      </endpoint>
    </client>
    <behaviors>
      <endpointBehaviors>
        <behavior name="ignoreCertValidation">
          <clientCredentials>
            <serviceCertificate>
              <authentication certificateValidationMode="None"/>
            </serviceCertificate>
          </clientCredentials>
        </behavior>
      </endpointBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

接下来需要做的是通过 Roles 这个静态类型对以证书表示的两个用户进行角色的分配。在客户端采用证书作为客户端凭证的情况下，用户名称的格式为“<<主题名称>>; <<指纹>>”。Foo 的主题名称为 CN=Foo，可以通过 MMC 的证书管理单元查看证书的指纹，比如指纹内容为 50819320DAAF1BAD9DE8823D3216BE9B36760C4D。那么我们只需要针对用户名“CN=Foo; 50819320DAAF1BAD9DE8823D3216BE9B36760C4D”进行授权就可以了。我们一样将角色分配实现在服务寄宿程序中。

```
if (!Roles.RoleExists("Administrators"))
{
    Roles.CreateRole("Administrators");
}
if (!Roles.IsUserInRole("CN=Foo; 50819320DAAF1BAD9DE8823D3216BE9B36760C4D",
    "Administrators"))
```

```

{
    Roles.AddUserToRole("CN=Foo; 50819320DAAF1BAD9DE8823D3216BE9B36760C4D",
        "Administrators");
}
using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
{
    host.Open();
    Console.Read();
}

```

然后客户端分别使用针对 Foo 和 Bar 的两张不同证书作为凭证进行服务调用, 相应的客户端程序如下所示。根据权限的不同, 也只有第一次服务调用能够成功。

```

ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService");
channelFactory.Credentials.ClientCertificate.SetCertificate( StoreLocation
.LocalMachine, StoreName.My, X509FindType.FindBySubjectName, "Foo");
ICalculator calculator = channelFactory.CreateChannel();
Invoke(calculator);

channelFactory = new ChannelFactory<ICalculator>("calculatorService");
channelFactory.Credentials.ClientCertificate.SetCertificate(StoreLocation.
LocalMachine, StoreName.My, X509FindType.FindBySubjectName, "Bar");
calculator = channelFactory.CreateChannel();
Invoke(calculator);

```

输出结果:

```

服务调用成功...
服务调用失败...

```

到目前为止, 我们已经从编程的角度对两种典型的安全主体权限模式进行了详细的介绍。很多人可能很想知道在 WCF 安全体系内部, 基于这两种模式的授权是如何实现的吧。为了让读者对授权有个深刻的认识, 接下来我们不采用 WCF 现有的授权编程方式, 而是通过 WCF 的扩展自行实现基于 Windows 用户组和 ASP.NET Roles 的授权模式。

8.3.5 实例演示: 通过 WCF 扩展实现授权 (S805)

针对安全主体的授权实现的原理其实很简单。只要你在服务操作执行之前能够根据被认证的用户正确设置当前的安全主体就可以了。如果了解 WCF 的整个运行时框架结构, 你会马上想到用于授权的安全主体初始化可以通过自定义 `CallContextInitializer` 来实现。如果你不了解也没关系, 下一章会专门讨论这个主题。

对于 WCF 的整个运行时框架来说, `CallContextInitializer` 是一个重要的对象。一个运行时服务操作 (`DispatchOperation`) 具有一个 `CallContextInitializer` 列表。而每一个 `CallContextInitializer` 实现 `System.ServiceModel.Dispatcher.ICallContextInitializer` 接口。如下面的代码片段所示, `ICallContextInitializer` 具有两个方法 `BeforeInvoke` 和 `AfterInvoke`。它们分别在操作方法之前和之后进行调用上下文的初始化和清理操作。我们可以通过自定义 `CallContextInitializer` 初始化当前的安全主体 (`BeforeInvoke` 方法)。

```
public interface ICallContextInitializer
{
    void AfterInvoke(object correlationState);
    object BeforeInvoke(InstanceContext instanceContext, IClientChannel channel,
        Message message);
}
```

步骤一：自定义 CallContextInitializer

我们为实现授权自定义了一个 `CallContextInitializer`，取名为 `AuthorizationCallContextInitializerBase`。如下面的代码所示，这是一个抽象类。抽象的方法 `GetPrincipal` 用于根据当前的安全上下文信息创建安全主体。该方法会在 `BeforeInvoke` 方法中被调用，返回值被设置成当前线程的安全主体。为了让服务操作执行之后当前线程的上下文恢复到执行前的状态，在 `BeforeInvoke` 方法中当前的安全主体被保存下来，并传递给 `AfterInvoke` 方法中恢复当前线程的原来的安全主体。

```
public abstract class AuthorizationCallContextInitializerBase:
    ICallContextInitializer
{
    public void AfterInvoke(object correlationState)
    {
        IPrincipal principal = correlationState as IPrincipal;
        if (null != principal)
        {
            Thread.CurrentPrincipal = principal;
        }
    }
    public object BeforeInvoke(InstanceContext instanceContext,
        IClientChannel channel, Message message)
    {
        var originalPrincipal = Thread.CurrentPrincipal;
        Thread.CurrentPrincipal =
            this.GetPrincipal(ServiceSecurityContext.Current);
        return originalPrincipal;
    }
    protected abstract IPrincipal GetPrincipal(ServiceSecurityContext
        serviceSecurityContext);
}
```

基于两种安全主体权限模式，我们创建了两个具体的 `CallContextInitializer`。第一个为具有如下定义的用于实现基于 Windows 用户组授权的 `WindowsAuthorizationCallContextInitializer`。在实现的抽象方法 `GetPrincipal` 中根据当前 `ServiceSecurityContext` 的 `WindowsIdentity` 属性创建 `WindowsPrincipal`。

```
public class WindowsAuthorizationCallContextInitializer:
    AuthorizationCallContextInitializerBase
{
    protected override IPrincipal GetPrincipal(ServiceSecurityContext
        serviceSecurityContext)
    {
        WindowsIdentity identity = serviceSecurityContext.WindowsIdentity;
        if (null == identity)
```

```

    {
        identity = WindowsIdentity.GetAnonymous();
    }
    return new WindowsPrincipal(identity);
}
}

```

而基于 ASP.NET Roles 安全主体权限模式的安全主体初始化实现在如下所示的 `AspRoleAuthorizationCallContextInitializer` 类中。它具有一个 `RoleProvider` 属性,表示用于获取当前用户角色列表的 `RoleProvider`,该属性在构造函数中被初始化。在实现的 `GetPrincipal` 抽象方法中,借助于 `RoleProvider` 获取基于当前用户的所有角色,并创建 `GenericPrincipal`。

```

public class AspRoleAuthorizationCallContextInitializer :
    AuthorizationCallContextInitializerBase
{
    public RoleProvider RoleProvider { get; private set; }
    public AspRoleAuthorizationCallContextInitializer(RoleProvider roleProvider)
    {
        this.RoleProvider = roleProvider;
    }
    protected override IPrincipal GetPrincipal(ServiceSecurityContext
        serviceSecurityContext)
    {
        var userName = serviceSecurityContext.PrimaryIdentity.Name;
        var identity = new GenericIdentity(userName);
        var roles = this.RoleProvider.GetRolesForUser(userName);
        return new GenericPrincipal(identity, roles);
    }
}

```

步骤二: 创建服务行为

用户进行安全主体初始化的两个具体的 `CallContextInitializer` 已经创建完成,现在需要做的工作就是将其应用到 WCF 的运行时框架体系之中。为此,我们创建了如下一个服务行为 `ServiceAuthorizationBehaviorAttribute`。

`ServiceAuthorizationBehaviorAttribute` 是一个自定义特性,并实现了 `IServiceBehavior` 接口。它具有 `PrincipalPermissionMode` 和 `CallContextInitializer` 两个属性。前者在构造函数中指定,我们根据该参数决定具体创建的 `CallContextInitializer` 类型 (`WindowsAuthorizationCallContextInitializer` 或 `AspRoleAuthorizationCallContextInitializer`)。而构造函数中具有一个可选的参数 `roleProviderName` 表示采用的 `RoleProvider` 配置名称。

```

[AttributeUsage( AttributeTargets.Class)]
public class ServiceAuthorizationBehaviorAttribute: Attribute, IServiceBehavior
{
    public PrincipalPermissionMode PrincipalPermissionMode { get; private set; }
    public ICallContextInitializer CallContextInitializer { get; private set; }

    public ServiceAuthorizationBehaviorAttribute(PrincipalPermissionMode
        principalPermissionMode, string roleProviderName = "")
    {
        switch (principalPermissionMode)

```



```

    {
        case PrincipalPermissionMode.UseWindowsGroups:
        {
            this.CallContextInitializer =
                new WindowsAuthorizationCallContextInitializer();
            break;
        }
        case PrincipalPermissionMode.UseAspNetRoles:
        {
            if (string.IsNullOrEmpty(roleProviderName))
            {
                this.CallContextInitializer = new
                    AspRoleAuthorizationCallContextInitializer
                        (Roles.Provider);
            }
            else
            {
                this.CallContextInitializer = new
                    AspRoleAuthorizationCallContextInitializer(Roles.Providers
                    [roleProviderName]);
            }
            break;
        }
        case PrincipalPermissionMode.Custom:
        {
            throw new ArgumentException(
                "只有 UseWindowsGroups 和 UseAspNetRoles 模式被支持! ");
        }
    }
}

public void AddBindingParameters(ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
    BindingParameterCollection bindingParameters) { }

public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase)
{
    if (null == this.CallContextInitializer)
    {
        return;
    }

    foreach (ChannelDispatcher channelDispatcher in
        serviceHostBase.ChannelDispatchers)
    {
        foreach (EndpointDispatcher endpoint in channelDispatcher.Endpoints)
        {
            foreach (DispatchOperation operation in
                endpoint.DispatchRuntime.Operations)
            {
                operation.CallContextInitializers.Add(this.CallContext
                    Initializer);
            }
        }
    }
}

public void Validate(ServiceDescription serviceDescription, ServiceHostBase
    serviceHostBase) { }

```

```
}
```

CallContextInitializer 的注册实现在 ApplyDispatchBehavior 方法中。我们遍历所有信道分发器 (ChannelDispatcher) 的每个信道分发器的所有终结点分发器 (EndpointDispatcher), 以及每个终结点分发器对应的分发运行时 (DispatchRuntime) 的所有运行时操作 (DispatchOperation)。最后将初始化的 CallContextInitializer 添加到操作的 CallContext Initializer 列表中。

步骤三: 使用服务行为进行授权

上面定义的服务行为 ServiceAuthorizationBehaviorAttribute 是一个自定义特性, 我们可以直接将其应用到服务类型上。在服务类型 CalculatorService 上应用了该特性, 并采用了 UseWindowsGroups 安全主体权限模式。

```
[ServiceAuthorizationBehavior(PrincipalPermissionMode.UseWindowsGroups)]
public class CalculatorService : ICalculator
{
    [PrincipalPermission(SecurityAction.Demand, Role = "Administrators")]
    public double Add(double x, double y)
    {
        return x + y;
    }
}
```

为了证明我们自定义的服务行为也能和 ServiceAuthorizationBehavior 一样实现正确的授权, 需要将 ServiceAuthorizationBehavior 的授权功能关闭。为此我们修正了服务端的配置, 将 ServiceAuthorizationBehavior 的 PrincipalPermissionMode 设置为 None。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService"
        behaviorConfiguration="disableAuthorization">
        <endpoint address="http://127.0.0.1:3721/calculatorservice"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.ICalculator"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="disableAuthorization">
          <serviceAuthorization principalPermissionMode="None"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

而客户端的服务调用程序中, 依然是分别以 Foo 和 Bar (Foo 具有管理员权限) 的名义进行两次服务调用。由于两个 Windows 账号权限不同, 同样只有第一个服务调用能够成功, 这反映在最终的执行结果中。

```

ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService");
NetworkCredential credential =
channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Foo";
credential.Password = "Password";
ICalculator calculator = channelFactory.CreateChannel();
Invoke(calculator);

channelFactory = new ChannelFactory<ICalculator>("calculatorService");
credential = channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Bar";
credential.Password = "Password";
calculator = channelFactory.CreateChannel();
Invoke(calculator);

```

输出结果:

```

服务调用成功...
服务调用失败...

```

8.4 自定义授权方式

如果采用 ASP.NET Roles 安全主体权限模式, 可以不受客户端凭证和认证方式的限制。而且 ASP.NET Roles 本身是一个具有很好可扩展性的角色管理和授权模块。如果现有的 RoleProvider 不能满足你的授权要求, 还可以自定义 RoleProvider。所以我们可以说 ASP.NET 能够解决绝大部分基于角色的授权。即便如此, WCF 依然通过可扩展的设计, 使你能够实现自定义的授权方式。

8.4.1 通过自定义 AuthorizationPolicy 和 ServiceAuthorizationManager 创建安全主体

自定义服务授权主要依赖于 AuthorizationPolicy 和 ServiceAuthorizationManager 这两个重要的对象。前者将用于辅助授权的策略信息存储于当前安全上下文中, 而后者根据当前的安全上下文最终决定当前的服务操作的调用是否应该被授权。

AuthorizationPolicy 实现了 System.IdentityModel.Policy.IAuthorizationPolicy 接口。如下面的代码所示, IAuthorizationPolicy 继承自 System.IdentityModel.Policy.IAuthorizationComponent 接口, 本身具有一个 ClaimSet 类型的 Issuer 属性和一个 Evaluate 方法。这里我们只需要关注 Evaluate 方法。该方法的第一个参数的类型为 System.IdentityModel.Policy.EvaluationContext, 它具有一个字典类型的只读属性 Properties。

```

public interface IAuthorizationPolicy : IAuthorizationComponent
{
    bool Evaluate(EvaluationContext evaluationContext, ref object state);
}

```

```

    ClaimSet Issuer {get; }
}
public abstract class EvaluationContext
{
    //其他成员
    public abstract IDictionary<string, object> Properties { get; }
}

```

如果需要通过自定义的方式来提供安全主体,我们只需要通过实现 `IAuthorizationPolicy` 接口创建自定义的 `AuthorizationPolicy`,并在 `Evaluate` 方法中按照如下的方式将创建安全主体对象添加到 `EvaluationContext` 的 `Properties` 字典中即可。在该字典中用于存放安全主体的条目对应的键值为“Principal”。

```

Public class CustomAuthorizationPolicy:IAuthorizationPolicy
{
    //其他成员
    bool Evaluate(EvaluationContext evaluationContext, ref object state)
    {
        //其他操作
        evaluationContext.Properties["Principal"] = customPrincipal;
        return true;
    }
}

```

那么自定义的 `AuthorizationPolicy` 通过怎样的方式被应用到 WCF 的授权运行时呢?这还是要借助于我们已经很熟悉的服务行为 `ServiceAuthorizationBehavior`。如下面给出的代码片段所示, `ServiceAuthorizationBehavior` 具有一个类型为 `ReadOnlyCollection<IAuthorizationPolicy>` 的 `ExternalAuthorizationPolicies` 属性,表示自定义 `AuthorizationPolicy` 的集合。

```

public sealed class ServiceAuthorizationBehavior : IServiceBehavior
{
    //其他成员
    public ReadOnlyCollection<IAuthorizationPolicy> ExternalAuthorizationPolicies
    { get; set; }
}

```

可以通过编程的方式将自定义的 `AuthorizationPolicy` 添加到 `ServiceAuthorizationBehavior` 的 `ExternalAuthorizationPolicies` 集合中,也可以通过配置指定自定义 `AuthorizationPolicy` 的类型。`ServiceAuthorizationBehavior` 的 `ExternalAuthorizationPolicies` 集合对应的配置节点为 `<serviceAuthorization>/<authorizationPolicies>`。我们可以按照如下所示的方式添加自定义的 `AuthorizationPolicy` 类型。

```

<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="useCustomAuthorization">
          <serviceAuthorization principalPermissionMode="Custom">
            <authorizationPolicies >
              <add policyType="AuthorizationPolicyType1" />
              <add policyType="AuthorizationPolicyType2" />
              ...
            </authorizationPolicies >
          </serviceAuthorization>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

```

        </authorizationPolicies>
    </serviceAuthorization>
    <serviceDebug includeExceptionDetailInFaults="true"/>
</behavior>
</serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

在 `ServiceAuthorizationBehavior` 选择 `Custom` 安全主体权限模式的情况下，除了自定义 `AuthorizationPolicy`，还可以通过自定义 `ServiceAuthorizationManager` 来提供当前的安全主体。下面给出了 `ServiceAuthorizationManager` 的定义，它具有两个 `CheckAccess` 方法用于实现授权。方法的返回值表示当前请求的服务操作是否被授权。实际上最终的授权判断实现在受保护方法 `CheckAccessCore` 中，并且在 `ServiceAuthorizationManager` 中该方法直接返回 `True`。

```

public class ServiceAuthorizationManager
{
    //其他成员
    public virtual bool CheckAccess(OperationContext operationContext);
    public virtual bool CheckAccess(OperationContext operationContext, ref Message
        message);
    protected virtual bool CheckAccessCore(OperationContext operationContext);
}

```

在 `ServiceAuthorizationBehavior` 的 `PrincipalPermissionMode` 被设置成 `Custom` 的情况下，被设置的当前安全主体实际上是通过当前服务安全上下文（`ServiceSecurityContext`）获取的。`ServiceSecurityContext` 具有一个表示授权信息的 `AuthorizationContext` 对象。和 `EvaluationContext` 一样，`AuthorizationContext` 也具有一个字典类型的 `Properties` 属性。实际上，通过 `AuthorizationPolicy` 添加到 `EvaluationContext` 中的属性，最终都会被转移到当前 `AuthorizationContext` 的 `Properties` 属性中。

```

public class ServiceSecurityContext
{
    //其他成员
    public AuthorizationContext AuthorizationContext { get; }
}
public abstract class AuthorizationContext : IAuthorizationComponent
{
    //其他成员
    public abstract IDictionary<string, object> Properties { get; }
}

```

所以只要我们能够在 `WCF` 从当前 `AuthorizationContext` 获取安全主体之前对其进行初始化，整个基于安全主体的授权体系就能正常运作。这个工作可以通过自定义 `ServiceAuthorizationManager` 来实现。一般来讲，我们只需通过继承 `ServiceAuthorizationManager`，重写虚方法 `CheckAccessCore` 并按照如下的方式进行安全主体的初始化即可。

```

public class CustomServiceAuthorizationManager : ServiceAuthorizationManager
{
    protected override bool CheckAccessCore(OperationContext operationContext)
    {

```

```

        //其他操作
        AuthorizationContext authorizationContext =
            operationContext.ServiceSecurityContext.AuthorizationContext;
        authorizationContext.Properties["Principal"] = customPrincipal;
        return true;
    }
}

```

自定义的 `ServiceAuthorizationManager` 最终还是通过 `ServiceAuthorizationBehavior` 这个服务行为应用到 WCF 授权框架体系中。如下面给出的代码片段所示，在 `ServiceAuthorizationBehavior` 中依然具有相应属性定义。而在 `ServiceAuthorizationBehavior` 的配置节中，`ServiceAuthorizationManager` 对应的配置属性为 `serviceAuthorizationManager`，可以通过该配置属性设置自定义 `ServiceAuthorizationManager` 的类型。

```

public sealed class ServiceAuthorizationBehavior: IServiceBehavior
{
    //其他成员
    public ServiceAuthorizationManager ServiceAuthorizationManager { get; set; }
}

```

到目前为止，我们介绍的授权策略都是围绕着安全主体进行的，基本上都是基于角色的授权。虽然角色是定义权限最为常用的形式，但是它解决不了授权的所有问题。有时候授权需要通过一个复杂的表达式来表示，而且其中会涉及诸多元素，比如身份、角色和组织等。

之所以说这么多，主要是为说明一个问题：授权策略有时候需要手工定制。而作为一种基于可扩展性的通信框架，WCF 在授权方面提供了扩展点，使我们可以根据实际需要定制相应的授权策略。WCF 创建了一个基于“声明”的授权系统，为了让读者对该系统的内部原理有全面的了解，我们不妨先来讨论一下这里指的“声明”是如何定义的。

8.4.2 Claim 和 ClaimSet

声明描述了与系统中某个实体关联的能力，该实体通常为该系统中的某个用户。通过对受保护资源所需的声明和与试图进行访问的实体关联的声明进行比较，便可确定该资源的访问权限。声明是针对目标对象的权限表达式。权限可以是读取、写入或拥有。目标对象可以是数据库、文件、邮箱或资产。声明还具有声明类型。

在 WCF 安全应用编程接口中，声明通过具有如下定义的类型 `System.IdentityModel.Claims.Claim` 表示。一个通过 `Claim` 对象表示的声明具有如下三要素，即声明类型 (`ClaimType`)、声明关联的资源 (`Resource`) 和声明代表的权限类型 (`Right`)。

```

public class Claim
{
    //其他成员
    public string ClaimType { get; }
    public object Resource { get; }
    public string Right { get; }
}

```

通过 `Right` 属性表示的权限类型一般通过一个统一资源标识符 (URI) 来表示。静态类 `System.IdentityModel.Claims.Rights` 定义了如下两个预定义的权限类型, 即 `Identity` 和 `PossessProperty`。前者表示声明用于身份标识, 后者则表示声明关联的实体具有的属性。

```
public static class Rights
{
    public static string Identity { get; }
    public static string PossessProperty { get; }
}
```

8.4.3 自定义授权实现原理剖析

借助于 WCF 的扩展, 我们通过自定义 `AuthorizationPolicy` 和 `ServiceAuthorizationManager` 来让 WCF 按照我们自定义的授权策略进行访问控制。这两个对象是如何参与到 WCF 的授权执行流程中的呢?

自定义的 `AuthorizationPolicy` 和 `ServiceAuthorizationManager` 通过服务行为 `ServiceAuthorizationBehavior` 成为 WCF 运行时的一部分。在 `ServiceAuthorizationBehavior` 的 `ApplyDispatchBehavior` 方法被调用的时候, 定义在 `ExternalAuthorizationPolicies` 属性中的 `AuthorizationPolicy` 列表和 `ServiceAuthorizationManager` 被赋予所有终结点的分发运行时。在 `DispatchRuntime` 类型中, 具有两个同名的属性。

```
public sealed class DispatchRuntime
{
    //其他成员
    public ReadOnlyCollection<IAuthorizationPolicy> ExternalAuthorizationPolicies
    { get; set; }
    public ServiceAuthorizationManager ServiceAuthorizationManager { get; set; }
}
```

整个自定义授权先从 `AuthorizationPolicy` 开始。WCF 先创建一个 `EvaluationContext` 对象。我们之前只提到过 `EvaluationContext` 用于表示属性的 `Properties`, 实际上它的核心是通过 `ClaimSets` 属性表示的声明集 (`ClaimSet`) 的集合。下面给出的 `EvaluationContext` 的整个公有成员的定义, 除了 `Properties` 和 `ClaimSets`, `EvaluationContext` 还具有一个额外的属性 `Generation` 表示声明集被添加到 `ClaimSets` 集合的次数。而声明集的添加通过方法 `AddClaimSet` 实现。`EvaluationContext` 具有有效性, 失效的时间可以通过方法 `RecordExpirationTime` 来记录。

```
public abstract class EvaluationContext
{
    public abstract void AddClaimSet(IAuthorizationPolicy policy, ClaimSet claimSet);
    public abstract void RecordExpirationTime(DateTime expirationTime);

    public abstract ReadOnlyCollection<ClaimSet> ClaimSets { get; }
    public abstract int Generation { get; }
    public abstract IDictionary<string, object> Properties { get; }
}
```

`EvaluationContext` 是一个抽象类型, 实际被创建的是一个被称为 `DefaultEvaluationContext` 的内部类型的对象。当 `EvaluationContext` 被初始化之后, WCF 会遍历定义在当前分发运行时 `ExternalAuthorizationPolicies` 属性中的所有 `AuthorizationPolicy`, 然后依次调用它们的 `Evaluate` 方法。而传入的参数就是之前初始化的这个 `EvaluationContext` 对象。一般我们通过自定义 `AuthorizationPolicy` 的 `Evaluate` 方法将基于自定义授权策略相关的声明集添加到 `EvaluationContext` 的 `ClaimSets` 中。

在所有的自定义 `AuthorizationPolicy` 的 `Evaluate` 方法被调用之后, 最终的 `EvaluationContext` 对象用于初始化当前的授权上下文 (`AuthorizationContext`)。下面给出了 `AuthorizationContext` 的所有公共属性的定义。一般来说, 除了 `Id`, 其余三个属性直接来源于 `EvaluationContext`。 `EvaluationContext` 的 `ClaimSets` 和 `Properties` 作为 `AuthorizationContext` 的 `ClaimSets` 和 `Properties`, 而 `EvaluationContext` 通过 `RecordExpirationTime` 记录的过期时间反映在 `AuthorizationContext` 的 `ExpirationTime` 上。

```
public abstract class AuthorizationContext : IAuthorizationComponent
{
    public abstract string Id { get; }

    public abstract ReadOnlyCollection<ClaimSet> ClaimSets { get; }
    public abstract DateTime ExpirationTime { get; }
    public abstract IDictionary<string, object> Properties { get; }
}
```

和 `EvaluationContext` 一样, `AuthorizationContext` 也是一个抽象类, 默认被创建的是一个名称为 `DefaultAuthorizationContext` 的内部类型对象。通过 `EvaluationContext` 创建的 `AuthorizationContext` 最终服务于自定义的 `ServiceAuthorizationManager` 以实现最终授权的判断。WCF 通过调用 `ServiceAuthorizationManager` 的 `CheckAccess` 方法决定当前操作是否被授权访问。

8.4.4 实例演示: 通过自定义 `AuthorizationPolicy` 和 `ServiceAuthorizationManager` 实现授权 (S806)

为了让读者对自定义授权有深刻的理解, 我们来进行一个相应的实例演示。这个实例依然采用简单的计算服务的例子。我们在服务契约 `ICalculator` 接口上定义如下4个分别表示加、减、乘、除的运算操作。服务类型 `CalculatorService` 也进行相应的修正。

服务契约:

```
using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface ICalculator
    {
```

```

[OperationContract(Action = "http://www.artech.com/calculator/add")]
double Add(double x, double y);
[OperationContract(Action = "http://www.artech.com/calculator/subtract")]
double Subtract(double x, double y);
[OperationContract(Action = "http://www.artech.com/calculator/multiply")]
double Multiply(double x, double y);
[OperationContract(Action = "http://www.artech.com/calculator/divide")]
double Divide(double x, double y);
}
}

```

服务类型:

```

using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Service
{
    public class CalculatorService : ICalculator
    {
        public double Add(double x, double y)
        {
            return x + y;
        }
        public double Subtract(double x, double y)
        {
            return x - y;
        }
        public double Multiply(double x, double y)
        {
            return x * y;
        }
        public double Divide(double x, double y)
        {
            return x / y;
        }
    }
}

```

现在的授权策略是这样的:操作 Add 和 Subtract 仅对用户 Foo 开放,而 Multiply 和 Divide 操作仅对用户 Bar 开放。虽然这个简单的授权完全可以通过在相应的服务操作方法上应用 `PrincipalPermissionAttribute` 并指定 `Name` 属性来实现,但是我们要尝试通过自定义 `AuthorizationPolicy` 和 `ServiceAuthorizationManager` 来实现这样的授权策略。

1. 自定义 AuthorizationPolicy

我们将自定义的 `AuthorizationPolicy` 创建在 `Service` 项目中。由于 `IAuthorizationPolicy` 定义在 `System.IdentityModel.dll` 程序集中,因此先为项目添加该程序集的引用。我们直接将自定义的 `AuthorizationPolicy` 命名为 `SimpleAuthorizationPolicy`,下面是整个 `SimpleAuthorizationPolicy` 的定义。

```

using System;
using System.Collections.Generic;
using System.IdentityModel.Claims;
using System.IdentityModel.Policy;
using System.Linq;

```

```

namespace Artech.WcfServices.Service
{
    public class SimpleAuthorizationPolicy : IAuthorizationPolicy
    {
        const string ActionOfAdd = "http://www.artech.com/calculator/add";
        const string ActionOfSubtract = "http://www.artech.com/calculator/subtract";
        const string ActionOfMultiply = "http://www.artech.com/calculator/multiply";
        const string ActionOfDivide = "http://www.artech.com/calculator/divide";

        internal const string ClaimType4AllowedOperation =
            "http://www.artech.com/allowed";

        public SimpleAuthorizationPolicy()
        {
            this.Id = Guid.NewGuid().ToString();
        }

        public bool Evaluate(EvaluationContext evaluationContext, ref object state)
        {
            if (null == state)
            {
                state = false;
            }
            bool hasAddedClaims = (bool)state;
            if (hasAddedClaims)
            {
                return true; ;
            }
            IList<Claim> claims = new List<Claim>();
            foreach (ClaimSet claimSet in evaluationContext.ClaimSets)
            {
                foreach (Claim claim in claimSet.FindClaims(ClaimTypes.Name,
                    Rights.PossessProperty))
                {
                    string userName = (string)claim.Resource;
                    if (userName.Contains('\'))
                    {
                        userName = userName.Split('\')[1];
                        if (string.Compare("Foo", userName, true) == 0)
                        {
                            claims.Add(new Claim(ClaimType4AllowedOperation,
                                ActionOfAdd, Rights.PossessProperty));
                            claims.Add(new Claim(ClaimType4AllowedOperation,
                                ActionOfSubtract, Rights.PossessProperty));
                        }
                        if (string.Compare("Bar", userName, true) == 0)
                        {
                            claims.Add(new Claim(ClaimType4AllowedOperation,
                                ActionOfMultiply, Rights.PossessProperty));
                            claims.Add(new Claim(ClaimType4AllowedOperation,
                                ActionOfDivide, Rights.PossessProperty));
                        }
                    }
                }
            }
            evaluationContext.AddClaimSet(this, new DefaultClaimSet(this.Issuer,
                claims));
            state = true;
            return true;
        }
    }
}

```

```

public ClaimSet Issuer
{
    get { return ClaimSet.System; }
}
public string Id { get; private set; }
}
}

```

Evaluate 方法的主要逻辑是这样的：通过 EvaluationContext 现有的声明集获取当前的用户名（声明类型和声明权限分别为 ClaimTypes.Name 和 Rights.PossessProperty），并根据用户名（Foo 或者 Bar）创建与被授权服务操作关联的声明。其中声明的三要素（类型、权限和资源）分别为“http://www.artech.com/allowed”、Rights.PossessProperty 和操作的 Action。最后将这些声明组成一个声明集添加到 EvaluationContext 中。

2. 自定义 ServiceAuthorizationManager

当授权相关的声明集通过自定义的 AuthorizationPolicy 被初始化之后，我们通过自定义 ServiceAuthorizationManager 来分析这些声明，并做出当前操作是否被授权调用的最终判断。我们将自定义的 ServiceAuthorizationManager 取名为 SimpleServiceAuthorizationManager，下面是整个定义。

```

using System.IdentityModel.Claims;
using System.Security.Principal;
using System.ServiceModel;
namespace Artech.WcfServices.Service
{
    public class SimpleServiceAuthorizationManager : ServiceAuthorizationManager
    {
        protected override bool CheckAccessCore(OperationContext operationContext)
        {
            string action =
                operationContext.RequestContext.RequestMessage.Headers.Action;
            foreach (ClaimSet claimSet in
                operationContext.ServiceSecurityContext.AuthorizationContext.ClaimSets)
            {
                if (claimSet.Issuer == ClaimSet.System)
                {
                    foreach (Claim c in
                        claimSet.FindClaims(
                            SimpleAuthorizationPolicy.ClaimType4AllowedOperation,
                            Rights.PossessProperty))
                    {
                        if (action == c.Resource.ToString())
                        {
                            GenericIdentity identity = new GenericIdentity("");
                            operationContext.ServiceSecurityContext.AuthorizationContext.Properties["Principal"]
                                = new GenericPrincipal(identity, null);
                            return true;
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
    return false;
}
}
}

```

由于基于被授权操作的声明已经通过 `SimpleAuthorizationPolicy` 被成功添加到 `EvaluationContext` 的声明集列表,并最终作为当前 `AuthorizationContext` 声明集的一部分,因此如果在这些代表被授权操作的声明中,有一个是基于当前被调用的服务操作的声明,就意味着当前的服务操作调用被授权了。这样的逻辑实现在重写的 `CheckAccessCore` 方法中。还有一点需要注意的是,在做出成功授权的情况下,需要设置当前的安全主体,因为不管这个安全主体是否需要, `WCF` 总是会试图从当前 `AuthorizationContext` 的属性列表中去获取该安全主体。如果没有则会抛出异常。

3. 应用自定义 `AuthorizationPolicy` 和 `ServiceAuthorizationManager`

两个核心的自定义对象 (`SimpleAuthorizationPolicy` 和 `SimpleServiceAuthorizationManager`) 都已经创建好了,我们现在通过配置的方式将它们设置到应用到服务的 `ServiceAuthorizationBehavior` 服务行为上。下面两段 XML 片段分别表示服务寄宿配置和客户端配置。

服务寄宿配置:

```

<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService"
        behaviorConfiguration="useCustomAuthorization">
        <endpoint address="http://127.0.0.1:3721/calculatorservice"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.ICalculator"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="useCustomAuthorization">
          <serviceAuthorization principalPermissionMode="Custom"
            serviceAuthorizationManagerType="Artech.WcfServices.Service.SimpleService
            AuthorizationManager,Artech.WcfServices.Service">
            <authorizationPolicies >
              <add policyType="Artech.WcfServices.Service.SimpleAuthorizationPolicy,
                Artech.WcfServices.Service" />
            </authorizationPolicies>
          </serviceAuthorization>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

客户端配置:

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="calculatorService"
        address="http://127.0.0.1:3721/calculatorservice"
        binding="ws2007HttpBinding"
        contract="Artech.WcfServices.Service.Interface.ICalculator"/>
    </client>
  </system.serviceModel>
</configuration>
```

我们最终需要验证 WCF 是否能够按照我们自定义的策略进行授权。为了演示方便, 我创建了如下一个名称为 `Invoke` 的辅助方法。`Invoke` 方法的三个参数分别代表进行服务调用的委托、服务代理对象和操作名称。服务操作调用会在该方法中执行, 并最终输出相应的文字表示服务调用是否成功。

```
static void Invoke(Action<ICalculator> action, ICalculator proxy, string
operation)
{
    try
    {
        action(proxy);
        Console.WriteLine("服务操作\"{0}\"调用成功...", operation);
    }
    catch (Exception ex)
    {
        Console.WriteLine("服务操作\"{0}\"调用失败...", operation);
    }
}
```

在如下的代码中, 我们分别以用户名 `Foo` 和 `Bar` 的名义通过上面的 `Invoke` 辅助方法对计算服务的 4 个操作进行访问。而程序执行的最终结果和我们自定义的授权策略是一致的, 即用户 `Foo` 仅授予了调用 `Add` 和 `Subtract` 操作的权限, 而其余两个授权给用户 `Bar`。

```
ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService");
NetworkCredential credential =
channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Foo";
credential.Password = "Password";
ICalculator calculator = channelFactory.CreateChannel();
Invoke(proxy => proxy.Add(1, 2), calculator, "Add");
Invoke(proxy => proxy.Subtract(1, 2), calculator, "Subtract");
Invoke(proxy => proxy.Multiply(1, 2), calculator, "Multiply");
Invoke(proxy => proxy.Divide(1, 2), calculator, "Divide");
Console.WriteLine();

channelFactory = new ChannelFactory<ICalculator>("calculatorService");
credential = channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Bar";
credential.Password = "Password";
calculator = channelFactory.CreateChannel();
Invoke(proxy => proxy.Add(1, 2), calculator, "Add");
```

```
Invoke(proxy => proxy.Subtract(1, 2), calculator, "Subtract");
Invoke(proxy => proxy.Multiply(1, 2), calculator, "Multiply");
Invoke(proxy => proxy.Divide(1, 2), calculator, "Divide");
```

输出结果:

```
服务操作"Add"调用成功...
服务操作"Subtract"调用成功...
服务操作"Multiply"调用失败...
服务操作"Divide"调用失败...

服务操作"Add"调用失败...
服务操作"Subtract"调用失败...
服务操作"Multiply"调用成功...
服务操作"Divide"调用成功...
```

8.5 安全审核 (Security Auditing)

WCF 所谓的安全审核就是针对认证和授权所做的针对 EventLog 的日志记录。我们不但可以设置进行审核的事件 (认证成功/失败和授权成功或失败), 还可以选择记录信息被写入的 EventLog 类型, 即应用程序日志 (Application) 还是安全日志 (Security)。WCF 的安全审核是通过 ServiceSecurityAuditBehavior 服务行为实现的。

8.5.1 ServiceSecurityAuditBehavior 服务行为

针对 WCF 安全审核的编程只涉及 System.ServiceModel.Description.ServiceSecurityAuditBehavior 服务行为。下面给出了定义在 ServiceSecurityAuditBehavior 中对具体审核行为进行控制的三个可读写的属性。

```
public sealed class ServiceSecurityAuditBehavior : IServiceBehavior
{
    //其他成员
    public AuditLogLocation AuditLogLocation { get; set; }
    public AuditLevel MessageAuthenticationAuditLevel { get; set; }
    public AuditLevel ServiceAuthorizationAuditLevel { get; set; }
    public bool SuppressAuditFailure { get; set; }
}
```

属性 AuditLogLocation 代表的是日志信息被写入的 EventLog 类型, 该属性的类型是一个具有如下定义的 System.ServiceModel.AuditLogLocation 枚举。Application 和 Security 分别代表应用程序日志和安全日志。如果选择 Default, 则最终日志被写入的位置决定于当前的操作系统。如果支持写入安全日志, 则选择安全日志类型, 否则选择应用程序日志类型。Default 是默认选项。

```
public enum AuditLogLocation
{
```

```

    Default,
    Application,
    Security
}

```

`MessageAuthenticationAuditLevel` 和 `ServiceAuthorizationAuditLevel` 两个属性分别代表针对认证和授权审核的级别。所谓审核的级别在这里指的是应该在审核事件（认证和授权）在成功或者失败的情况下进行日志记录。审核级别通过具有如下定义的 `System.ServiceModel.AuditLevel` 枚举表示。`Success` 和 `Failure` 代表分别针对认证/授权成功和失败进行审核日志。`SuccessOrFailure` 则意味着不管认证/授权是成功还是失败，都会进行审核日志记录。`None` 为默认值，表示不进行审核日志记录。

```

public enum AuditLevel
{
    None,
    Success,
    Failure,
    SuccessOrFailure
}

```

布尔类型的 `SuppressAuditFailure` 属性表示审核日志失败是否会影响应用本身。在默认的情况下该属性值为 `True`，意味着为认证和授权进行审核日志记录的时候出现的异常不会对应用（服务）本身造成任何影响。

既然是服务行为，那么可以通过将创建的 `ServiceSecurityAuditBehavior` 添加到服务的行为列表的方式来进行安全审核的控制。当然还是推荐采用配置的方式来进行安全审核的相关设置。服务行为 `ServiceSecurityAuditBehavior` 对应的配置节是 `<serviceSecurityAudit>`。在下面的配置中，我们定义了一个包含了 `ServiceSecurityAuditBehavior` 的服务行为，并对其 4 个属性进行了显式设置。

```

<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior ...>
          <serviceSecurityAudit auditLogLocation = "Application"
                                messageAuthenticationAuditLevel = "Failure"
                                serviceAuthorizationAuditLevel = "SuccessOr Failure"
                                suppressAuditFailure = "true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

8.5.2 安全审核的实现

WCF 最终进行安全审核的控制信息是从基于某个终结点的分发运行时中获取的。如下面的代码片段所示，对于定义在 `ServiceSecurityAuditBehavior` 中的 4 个属性，在

DispatchRuntime 中具有相应的定义。WCF 在认证和授权成功或者失败的时候, 会根据该运行时对这 4 个属性进行相应安全审核日志记录。

```
public sealed class DispatchRuntime
{
    //其他成员
    public AuditLogLocation SecurityAuditLogLocation { get; set; }
    public AuditLevel MessageAuthenticationAuditLevel { get; set; }
    public AuditLevel ServiceAuthorizationAuditLevel { get; set; }
    public bool SuppressAuditFailure { get; set; }
}
```

作为服务行为的 ServiceSecurityAuditBehavior, 最终的目的就是将定义在自身的这 4 个属性赋值给分发运行时。具体操作定义在 ApplyDispatchBehavior 方法上, 整个逻辑大体上如下面的代码所示。

```
public sealed class ServiceSecurityAuditBehavior : IServiceBehavior
{
    //其他成员
    public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
        ServiceHostBase serviceHostBase)
    {
        foreach (ChannelDispatcher channelDispatcher in
            serviceHostBase.ChannelDispatchers)
        {
            foreach (EndpointDispatcher endpointDispatcher in
                channelDispatcher.Endpoints)
            {
                DispatchRuntime dispatchRuntime = endpointDispatcher.DispatchRuntime;
                dispatchRuntime.SecurityAuditLogLocation = this.AuditLogLocation;
                dispatchRuntime.MessageAuthenticationAuditLevel =
                    this.MessageAuthenticationAuditLevel;
                dispatchRuntime.ServiceAuthorizationAuditLevel =
                    this.ServiceAuthorizationAuditLevel;
                dispatchRuntime.SuppressAuditFailure = this.SuppressAuditFailure;
            }
        }
    }
}
```

8.5.3 实例演示: 如何实施安全审核

接下来通过一个简单的实例来演示如何进行针对认证和授权的安全审核, 并且看看在认证或者授权成功或者失败的情况下, 会有怎样的日志信息被记录下来。

1. 基于认证的安全审核 (S807)

还是直接使用计算服务的例子, 服务契约和服务的定义我们已经很熟悉了, 现在只来介绍采用的服务端的配置。从下面的配置可以看出, 被寄宿的服务具有一个基于 WS2007HttpBinding 的终结点, 该绑定采用默认的 Windows 认证。通过服务行为, 我们将安

全审核的 `AuditLogLocation` 和 `MessageAuthenticationAuditLevel` 分别设置为 `Application` 和 `SuccessOrFailure`。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService"
        behaviorConfiguration="authenticationAudit">
        <endpoint address="http://127.0.0.1:3721/calculatorService"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.ICalculator"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="authenticationAudit">
          <serviceSecurityAudit auditLogLocation ="Application"
            messageAuthenticationAuditLevel
            ="SuccessOrFailure"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

客户端利用如下的代码进行服务的调用，两次服务调用分别采用 `Foo` 和 `Bar` 两个本地 Windows 账号进行，其中基于账号 `Foo` 给定的密码是正确的，基于账号 `Bar` 给定的密码是错误的。辅助方法 `Invoke` 旨在避免认证失败导致的异常使程序终止，在前面的部分具有相应的定义。

```
ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService");
NetworkCredential credential =
channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Foo";
credential.Password = "Password";
ICalculator calculator = channelFactory.CreateChannel();
Invoke(calculator);

channelFactory = new ChannelFactory<ICalculator>("calculatorService");
credential = channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Bar";
credential.Password = "InvalidPass";
calculator = channelFactory.CreateChannel();
Invoke(calculator);
```

由于我们通过配置将应用在服务上的 `ServiceSecurityAuditBehavior` 服务行为的 `AuditLogLocation` 和 `MessageAuthenticationAuditLevel` 分别设置为 `Application` 和 `SuccessOrFailure`，因此意味着不论是认证成功或者失败都会进行安全审核日志记录。而审核日志最终会被写入 `EventLog` 的应用程序日志。当程序执行后，在事件查看器的应用程序节点，你会发现有如图 8-2 所示的 4 条新的日志（之前的日志在程序运行前被清空）。

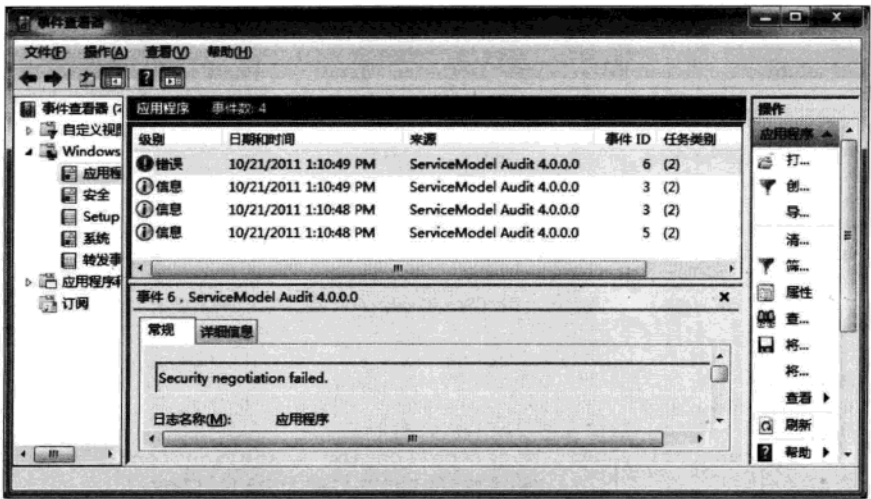


图 8-2 认证安全审核被写入的日志

下面列出了这 4 条日志的内容。其中前 3 条是基于认证成功的信息 (Information) 日志，最后一条是基于认证失败的错误 (Error) 日志。

```
Security negotiation succeeded.
Service: http://127.0.0.1:3721/calculator-service
Action: http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Issue
ClientIdentity: Jinnan-PC\Foo; S-1-5-21-3534336654-2901585401-846244909-1006
ActivityId: <null>
Negotiation: SpnegoTokenAuthenticator

Message authentication succeeded.
Service: http://127.0.0.1:3721/calculator-service
Action: http://docs.oasis-open.org/ws-sx/ws-trust/200512/RST/SCT
ClientIdentity: Jinnan-PC\Foo; S-1-5-21-3534336654-2901585401-846244909-1006
ActivityId: <null>

Message authentication succeeded.
Service: http://127.0.0.1:3721/calculator-service
Action: http://www.artech.com/ICalculator/Add
ClientIdentity: Jinnan-PC\Foo; S-1-5-21-3534336654-2901585401-846244909-1006
ActivityId: <null>

Security negotiation failed.
Service: http://127.0.0.1:3721/calculator-service
Action: http://docs.oasis-open.org/ws-sx/ws-trust/200512/RSTR/Issue
ClientIdentity:
ActivityId: <null>
Negotiation: SpnegoTokenAuthenticator
Win32Exception: The Security Support Provider Interface (SSPI) negotiation
failed.
```

2. 基于授权的安全审核 (S808, S809)

接下来演示授权的安全审核，并查看在授权成功和失败的情况下分别有怎样的日志被写

入到 EventLog 中。我们首先按照如下的方式在服务类型 CalculatorService 的 Add 方法上应用 PrincipalPermissionAttribute 特性使该方法只有管理员才有权限调用。

```
public class CalculatorService : ICalculator
{
    [PrincipalPermission(SecurityAction.Demand, Role="Administrators")]
    public double Add(double x, double y)
    {
        return x + y;
    }
}
```

然后将服务端配置中关于安全审核的相关配置进行如下的修改。我们只关心授权相关的安全审核，所以将 messageAuthenticationAuditLevel 属性替换成 serviceAuthorizationAuditLevel。

```
<configuration>
  <system.serviceModel>
    ...
    <behaviors>
      <serviceBehaviors>
        <behavior name="authenticationAudit">
          <serviceSecurityAudit auditLogLocation="Application"
                                serviceAuthorizationAuditLevel
                                  ="SuccessOrFailure"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

客户端在进行服务调用的时候需要为账号 Bar 指定正确的密码。这里的账号 Foo 被预先添加到管理员用户组中而 Bar 则没有，所以只有第一次服务调用才是被成功授权的。

```
ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorService");
NetworkCredential credential =
channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Foo";
credential.Password = "Password";
ICalculator calculator = channelFactory.CreateChannel();
Invoke(calculator);

channelFactory = new ChannelFactory<ICalculator>("calculatorService");
credential = channelFactory.Credentials.Windows.ClientCredential;
credential.UserName = "Bar";
credential.Password = "Password";
calculator = channelFactory.CreateChannel();
Invoke(calculator);
```

当客户端完成两次服务调用后，如下两条基于授权的审核日志被写入应用程序日志。虽然只有第一次服务调用才是真正被授权的操作，但是从日志的内容我们却发现两条均是“授权成功”的审核日志。(S808)

```
Service authorization succeeded.
Service: http://127.0.0.1:3721/calculatorService
Action: http://www.artech.com/ICalculator/Add
ClientIdentity: Jinnan-PC\Foo; S-1-5-21-3534336654-2901585401-846244909-1006
AuthorizationContext: uuid-528e86ce-a4f4-48b3-9a2e-b713e1dea539-1
ActivityId: <null>
ServiceAuthorizationManager: <default>
```

Service authorization succeeded.

```
Service: http://127.0.0.1:3721/calculatorService
Action: http://www.artech.com/ICalculator/Add
ClientIdentity: Jinnan-PC\Bar; S-1-5-21-3534336654-2901585401-846244909-1007
AuthorizationContext: uuid-528e86ce-a4f4-48b3-9a2e-b713e1dea539-2
ActivityId: <null>
ServiceAuthorizationManager: <default>
```

实际上安全审核中所谓的授权失败指的是调用 `ServiceAuthorizationManager` 的 `CheckAccess` 方法返回 `False` 的情况。在默认的情况下, `ServiceAuthorizationManager` 的 `CheckAccess` 方法总是返回 `True`, 所以授权总是会“成功”。

为了迎合安全审核对“授权失败”的判断, 我在 `Service` 项目中创建了如下一个简单的自定义 `ServiceAuthorizationManager`。在重写的 `CheckAccessCore` 方法中, 如果当前用户是 `Foo (Jinnan-PC\Jinnan)` 就返回 `True`, 否则返回 `False`。

```
using System.ServiceModel;
namespace Artech.WcfServices.Service
{
    public class MyServiceAuthorizationManager : ServiceAuthorizationManager
    {
        protected override bool CheckAccessCore(OperationContext operationContext)
        {
            string userName =
                operationContext.ServiceSecurityContext.PrimaryIdentity.Name;
            return string.Compare(userName, @"Jinnan-PC\Foo", true) == 0;
        }
    }
}
```

然后通过如下的配置将这个自定义的 `MyServiceAuthorizationManager` 应用到 `ServiceAuthorizationBehavior` 服务行为上。再次运行程序, 将会得到分别代表授权成功和失败的两条审核日志, 并且在日志中还包含了我们自定义的 `ServiceAuthorizationManager` 类型 (`ServiceAuthorizationManager: MyServiceAuthorizationManager`)。(S809)

```
<configuration>
  <system.serviceModel>
    ...
    <behaviors>
      <serviceBehaviors>
        <behavior name="authenticationAudit">
          <serviceAuthorization
            serviceAuthorizationManagerType="Artech.WcfServices.Service.MyServiceAuthorizationManager, Artech.WcfServices.Service">
          </serviceAuthorization>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

```

    <serviceSecurityAudit auditLogLocation ="Application"
                        serviceAuthorizationAuditLevel
                        ="SuccessOrFailure"/>
  </behavior>
</serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

授权成功/失败审核日志:

Service authorization succeeded.

Service: http://127.0.0.1:3721/calculatorService

Action: http://www.artech.com/ICalculator/Add

ClientIdentity: Jinnan-PC\Foo; S-1-5-21-3534336654-2901585401-846244909-1006

AuthorizationContext: uuid-91ff29c8-84cb-4c1a-837c-0913c12c0be5-1

ActivityId: <null>

ServiceAuthorizationManager: MyServiceAuthorizationManager

Service authorization failed.

Service: http://127.0.0.1:3721/calculatorService

Action: http://www.artech.com/ICalculator/Add

ClientIdentity: Jinnan-PC\Bar; S-1-5-21-3534336654-2901585401-846244909-1007

AuthorizationContext: uuid-91ff29c8-84cb-4c1a-837c-0913c12c0be5-2

ActivityId: <null>

ServiceAuthorizationManager: MyServiceAuthorizationManager

FaultException: Access is denied.



第9章 扩展

(Extension)

几乎在本书的每一章，我们都在提及 WCF 的扩展性。WCF 从第一个版本的推出到现在，整个架构体系基本上没有出现大的变化。在我看来，可扩展性设计是 WCF 的架构体系能够在一段不短的时期内保持稳定的一个最主要的原因。而反观与 WCF 几乎同时推出的 WF (Windows Workflow Foundation) 各个版本演进的历史，你会发现 WF 推出的每一个版本几乎都是“革新性”的改变。正是因为 WCF 具有极大的扩展性，WCF 新的版本的功能本身就可以通过这些丰富的扩展点来实现，而作为最终使用者的我们也可以实现自己的扩展。

可以简单地将 WCF 看成是一个消息处理的管道。客户端将服务操作转换成消息并对外发送，同时将接收到的回复消息转换成方法调用的返回值（或者 out/ref 参数）。服务端管道接收请求消息并激活相应的服务实例，将请求消息转换成参数并调用操作方法。最后将执行操作方法返回的结果通过消息进行封装，返回给客户端。

无论是客户端管道还是服务端管道，它们的每一个环节大多由一组标准的组件构成。所以说它们是标准的组件，是因为它们大多具有相应的接口。我们可以通过实现这些接口自定义相应的组件，最终将它们添加到该管道中。除了动态地注册自定义组件，我们还可以改变运行时属性从而改变 WCF 运行的方式。对 WCF 的扩展主要体现在两个方面：

- 出于某种扩展的需求定义相应的组件，并将自定义的组件注册到 WCF 的运行时框架中。
- 通过修改运行时相应的属性让 WCF 按照我们期望的方式工作。

如果读者想得心应手地对 WCF 进行扩展，就需要先对 WCF 的服务端和客户端运行时框架有充分的了解。

9.1 服务端架构体系的构建

完成一个服务寄宿只需要用到一个唯一的对象，那就是 ServiceHost。甚至在某种语境下，我们所说的服务实际上就是指某个 ServiceHost 对象。上册的第 7 章“服务寄宿 (Hosting)”已经对 ServiceHost 进行了非常详尽的介绍。如果你阅读过该章节，应该知道整个服务寄宿过程包括“服务描述的创建”和“端运行框架的建立”这两个阶段。而第一个阶段创建的服务描述是为第二个阶段对服务端运行时框架建立服务的，所以我们有必要再对服务描述进行简单的回顾。

9.1.1 再谈服务描述 (Service Description)

ServiceHost 在被实例化的过程中，用于描述整个服务的 ServiceDescription 对象被创建出来。对于一个服务来说，它的核心包括一组终结点列表和一组服务行为列表。这可以通过如下所示的 ServiceDescription 的定义看出来。

```
public class ServiceDescription
{
    //其他成员
    public KeyedByTypeCollection<IServiceBehavior> Behaviors { get; }
    public ServiceEndpointCollection Endpoints { get; }
}
```

对于终结点，它的 ABC 三要素，即地址 (Address)、绑定 (Binding) 和契约 (Contract) 我们早已了然于胸了。用于描述终结点的 ServiceEndpoint 类型具有 Address、Binding 和 Contract 三个核心属性。此外还有基于该终结点的行为列表，通过 Behaviors 属性表示。ServiceEndpoint 的定义如下所示。

```

public class ServiceEndpoint
{
    //其他成员
    public EndpointAddress Address { get; set; }
    public Binding Binding { get; set; }
    public ContractDescription Contract { get; }
    public KeyedByTypeCollection<IEndpointBehavior> Behaviors { get; }
}

```

服务契约本质上是一组相关操作的组合, 所以代表契约描述的 `ContractDescription` 类型的核心属性是如下所示的表示所有操作描述的 `Operations` 属性。除了操作描述列表之外, 还有基于服务契约本身的行为列表。

```

public class ContractDescription
{
    //其他成员
    public OperationDescriptionCollection Operations { get; }
    public KeyedByTypeCollection<IContractBehavior> Behaviors { get; }
}

```

至于对服务操作的描述, 对应的类型为 `OperationDescription`。 `OperationDescription` 中定义了一系列基于服务操作的属性, 在这里我们主要关注的是用于表示操作行为列表的属性 `Behaviors`。

```

public class OperationDescription
{
    //其他成员
    public KeyedByTypeCollection<IOperationBehavior> Behaviors { get; }
}

```

上述从服务 `ServiceDescription` 到 `ServiceEndpoint`, 从 `ServiceEndpoint` 到 `ContractDescription`, 最终到 `OperationDescription` 的层次结构基本上可以通过图 9-1 来表示。

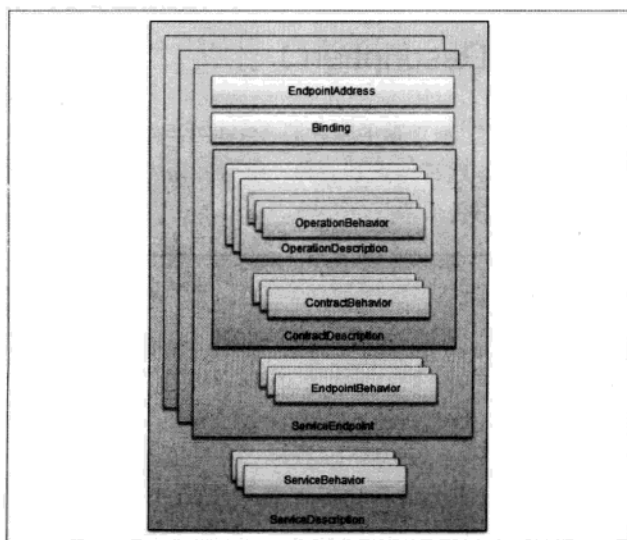


图 9-1 ServiceDescription 的层次结构

在构建 `ServiceHost` 过程中创建的用于描述整个服务的 `ServiceDescription` 对象，最终成为构建服务端运行时架构体系的基础。而该架构体系在 `ServiceHost` 开启的过程中被构建出来，这也是为什么在 `ServiceHost` 开启之后对服务描述所做的任何改变都无效的根本原因。

为了让读者对服务端运行时架构体系的结构具有更加深刻的认识，我们针对一个具体的服务寄宿应用场景来进行介绍。假设采用如下的配置对服务 `CalculatorService` 进行寄宿。通过这段配置，添加了三个基于 `WSHttpBinding` 的终结点。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.CalculatorService">
        <endpoint address="http://127.0.0.1:7777/CalculatorService"
          binding="wsHttpBinding"
          contract="Artech.WcfServices.ICalculator"
          listenUri="http://127.0.0.1:6666/CalculatorService"
          listenUriMode="Explicit"/>
        <endpoint address="http://127.0.0.1:8888/CalculatorService"
          binding="wsHttpBinding"
          contract="Artech.WcfServices.ICalculator"
          listenUri="http://127.0.0.1:6666/CalculatorService"
          listenUriMode="Explicit"/>
        <endpoint address="http://127.0.0.1:9999/CalculatorService"
          binding="wsHttpBinding"
          contract="Artech.WcfServices.ICalculator"
          listenUri="http://127.0.0.1:6666/CalculatorService"
          listenUriMode="Unique"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

虽然配置的三个终结点具有不同的地址，但是它们却使用了相同的监听 URI（通过 `listenUri` 属性设置）。虽然三个终结点具有相同的监听 URI，但是它们的监听 URI 模式（通过 `listenUriMode` 属性设置）不同，前两个终结点为 `Explicit`，第三个为 `Unique`，所以最终的监听地址具有两个：

- `http://127.0.0.1:6666/CalculatorService`
- `http://127.0.0.1:6666/CalculatorService/⟨⟨guid⟩⟩`

当 `ServiceHost` 正常开启后，WCF 最终会创建如图 9-2 所示的架构体系。首先通过调用绑定的 `BuildChannelListener` 方法创建信道监听器（实际上是多个信道监听器构成的信道监听器管道，最终返回的是最上层的信道监听器）。这两个信道监听器分别绑定到上述的两个监听地址进行请求消息的监听。

针对这两个信道监听器，WCF 会创建相应的信道分发器（`ChannelDispatcher`）。而针对在配置中定义三个终结点，它们则分别对应着一个终结点分发器（`EndpointDispatcher`）。每个终结点分发器都具有各自的运行时，被称为分发运行时（`DispatchRuntime`）。

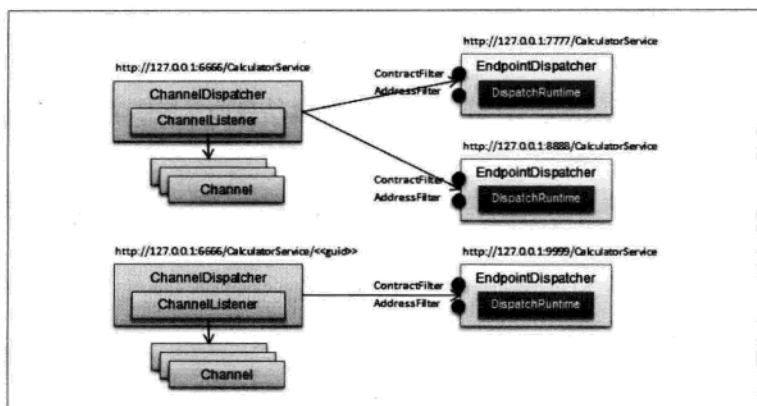


图 9-2 服务端架构体系

当信道监听器成功监听到抵达的请求消息时，它会利用创建的信道栈对消息进行接收和处理，并最终分发给相应的终结点分发器。终结点最终将接收到的消息在自己的分发运行中进行处理。处理后的结果被封装在创建的回复消息中回传给信道分发器，并最终通过信道栈返回给客户端。

如图 9-2 所示，如果多个终结点共享同一个监听地址，一个信道分发器就会对应多个终结点分发器。信道监听器在接收到经过信道栈接收和处理的消息后，如何判断需要将消息转发给哪个终结点分发器呢？

9.1.2 终结点分发器选择机制

除了分发运行时，每个终结点分发器还具有两个重要消息筛选器，一个是地址筛选器 (AddressFilter)，另一个是契约筛选器 (ContractFilter)。信道分发器就是通过这两个消息筛选器最终决定应该选择哪个终结点分发器来接收请求消息的。

消息筛选器继承自具有如下定义的 MessageFilter 抽象类。MessageFilter 具有两个重载的分别以 Message 和 MessageBuffer 作为参数的方法。信道分发器在决定应该将接收的消息路由给哪个终结点分发器之前，会将路由的消息作为输入调用所有终结点分发器的两个消息筛选器的 Match 方法。如果方法返回 True，则表明该终结点分发器与当前的消息相匹配。

```
public abstract class MessageFilter
{
    public abstract bool Match(Message message);
    public abstract bool Match(MessageBuffer buffer);
}
```

终结点分发器在 WCF 的应用编程接口中通过具有如下定义的 EndpointDispatcher 类型表示。除了代表上述两个消息筛选器的两个属性 AddressFilter 和 ContractFilter 之外，还有一个

额外的整型的 `FilterPriority` 属性。`FilterPriority` 属性表示筛选的优先级。在两个以上终结点分发器同时与路由的消息匹配的情况下，优先级最高的终结点分发器会被选用。代表 `FilterPriority` 的数据越大，意味着优先级越高。如果同时有两个或两个以上具有最高筛选优先级的终结点分发器，系统会抛出一个 `System.ServiceModel.Dispatcher.MultipleFilterMatchesException` 异常。

```
public class EndpointDispatcher
{
    //其他成员
    public MessageFilter AddressFilter { get; set; }
    public MessageFilter ContractFilter { get; set; }
    public int FilterPriority { get; set; }
}
```

为了满足各种消息路由的需要，WCF 为我们定义了如下 6 种典型的消息筛选器。如果这 6 种消息筛选器依然不能满足需求，可以通过继承 `MessageFilter` 这个抽象类创建自定义的消息筛选器。

- `ActionMessageFilter`: 判断请求消息 (SOAP) 的 <Action> 报头是否和终结点契约中的任何一个操作的 `Action` 属性相匹配;
- `EndpointAddressMessageFilter`: 判断请求消息 (SOAP) 的 <To> 报头是否和终结点地址相匹配;
- `XPathMessageFilter`: 判断请求消息 (XML) 是否和指定的 XPath 表达式相匹配;
- `PrefixEndpointAddressMessageFilter`: 和 `EndpointAddressMessageFilter` 筛选机制类似，不同的是 `PrefixEndpointAddressMessageFilter` 采用“最长前缀匹配”机制。比如终结点地址指定的 URI 为 `http://www.artech.com/Foo`，而请求消息的 <To> 报头的 URI 为 `http://www.artech.com/Foo/Bar`，这样可以被认为是匹配的;
- `MatchAllMessageFilter`: 不管消息的内容是什么，都会匹配成功;
- `MatchNoneMessageFilter`: 和 `MatchAllMessageFilter` 相反，不管消息的内容是什么，都不会匹配成功。

在默认的情况下，终结点分发器的 `AddressFilter` 和 `ContractFilter` 分别采用的是 `EndpointAddressMessageFilter` 和 `ActionMessageFilter`。可以通过 `ServiceBehaviorAttribute` 特性的 `AddressFilterMode` 指定改变终结点分发器的 `AddressFilter`。该属性值是 `AddressFilterMode` 枚举。三个枚举值代表 `EndpointAddressMessageFilter`、`PrefixEndpointAddressMessageFilter` 和 `MatchAllMessageFilter` 这三种消息筛选器。

```
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public AddressFilterMode AddressFilterMode { get; set; }
}
public enum AddressFilterMode
```

```
{
    Exact,
    Prefix,
    Any
}
```

9.1.3 信道分发器 (ChannelDispatcher)

信道分发器通过类型 `System.ServiceModel.Dispatcher.ChannelDispatcher` 表示。我们从可扩展的角度来分析一下信道分发器具有哪些扩展点。先来看看它具有哪些可供扩展的组件。

1. 可扩展组件

如图 9-3 所示，信道分发器的核心对象是用于消息监听的信道监听器 (`ChannelListener`)。除此之外，它还具有三种类型的组件，包括用于进行异常处理的一组错误处理器 (`ErrorHandler`)，进行流量控制的服务限流器 (`ServiceThrottle`) 和一组用于初始化信道的信道初始化器 (`ChannelInitializer`)。

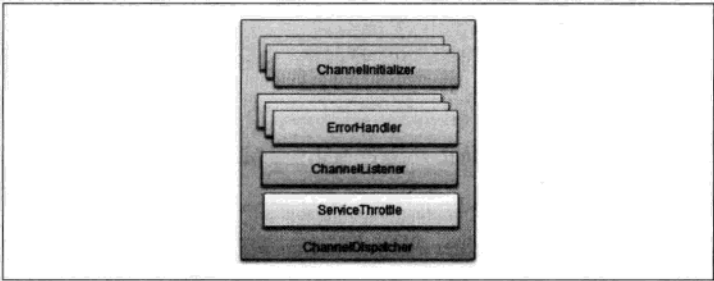


图 9-3 信道分发器结构

(1) 异常处理

如下面的代码所示，`ChannelDispatcher` 具有一个元素类型为 `IErrHandler` 的集合属性 `ErrorHandlers`。该集合中的元素被称为错误处理器 (`ErrorHandler`)。我们可以将自定义的错误处理器添加到信道分发器的 `ErrorHandlers` 集合中，实现对异常处理和对错误消息的定制。基于错误处理器的扩展在本书的第 1 章“异常处理 (Exception Handling)”有详细的介绍。

```
public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public Collection<IErrHandler> ErrorHandlers { get; }
}

public interface IErrHandler
{
    bool HandleError(Exception error);
    void ProvideFault(Exception error, MessageVersion version, ref Message
    fault);
}
```

(2) 流量控制

如下面的代码片段所示，用于进行流量控制的 `ServiceThrottle` 对象作为属性定义在信道分发器上。而包含三个限流指标（最大并发调用、最大并发实例和最大并发会话）的 `ServiceThrottle` 最终是通过服务行为 `ServiceThrottlingBehavior` 被应用到信道分发器上的。流量限制相关的编程和实现原理在本书第4章“并发与限流（Concurrency and Throttling）”中有详细的介绍。

```
public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public ServiceThrottle ServiceThrottle { get; set; }
}
public sealed class ServiceThrottle
{
    //其他成员
    public int MaxConcurrentCalls { get; set; }
    public int MaxConcurrentInstances { get; set; }
    public int MaxConcurrentSessions { get; set; }
}
```

(3) 信道初始化

如下面的代码片段所示，信道分发器具有一个集合类型的 `ChannelInitializers` 属性。集合中的元素被称为信道初始化器（`ChannelInitializer`），它们实现了 `System.ServiceModel.Dispatcher.IChannelInitializer` 接口。信道初始化用于实现针对服务端信道的初始化。

```
public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public SynchronizedCollection<IChannelInitializer> ChannelInitializers { get; }
}
public interface IChannelInitializer
{
    void Initialize(IClientChannel channel);
}
```

2. 可扩展属性

信道分发器本身具有一些用于控制消息分发行为的属性。也可以根据需要改变这些属性使信道分发器按照你希望的行为进行消息的分发。

(1) 异常细节信息传播

信道分发器的属性 `IncludeExceptionDetailInFaults` 用于控制服务端异常细节信息的传播。该属性的初始化由服务行为 `ServiceDebugBehavior` 完成。可以通过 `ServiceBehaviorAttribute` 特性来控制这个开关。如下面的代码所示，`ServiceDebugBehavior` 和 `ServiceBehaviorAttribute` 均具有可读写的 `IncludeExceptionDetailInFaults` 属性。关于异常详细信息的传播机制在本书第1章“异常处理（Exception Handling）”中有非常详细的介绍。

```

public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public bool IncludeExceptionDetailInFaults { get; set; }
}
public class ServiceDebugBehavior : IServiceBehavior
{
    //其他成员
    public bool IncludeExceptionDetailInFaults { get; set; }
}
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public bool IncludeExceptionDetailInFaults { get; set; }
}

```

(2) 手工寻址

如果采用 WS-Addressing 寻址机制, SOAP 消息在发送之前会被赋予基于 WS-Addressing 的寻址报头 (比如<To>、<ReplyTo>和<RelatesTo>等)。WS-Addressing 报头的添加是通过信道层底层的传输信道完成的。如下面的代码所示, 传输信道对应的绑定元素类 `TransportBindingElement` 中具有同名的属性。

```

public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public bool ManualAddressing { get; set; }
}
public abstract class TransportBindingElement : BindingElement
{
    //其他成员
    public bool ManualAddressing { get; set; }
}

```

如果需要进行手工方式寻址, 可以通过将信道分发器的 `ManualAddressing` 属性设置为 `True` 来实现。在这种情况下, SOAP 消息传入信道层被发送之前, 传输信道将不再自定义添加 WS-Addressing 报头。

(3) 最大挂起消息数

信道分发器具有一个整型的属性 `MaxPendingReceives` 表示允许的最大挂起的消息数。可以通过终结点行为 `System.ServiceModel.Description.DispatcherSynchronizationBehavior` 设置该属性。如下面的代码片段所示, `DispatcherSynchronizationBehavior` 具有一个同名的属性。 `MaxPendingReceives` 的默认值为 1。

```

public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public int MaxPendingReceives { get; set; }
}

```

```
public class DispatcherSynchronizationBehavior : IEndpointBehavior
{
    //其他成员
    public int MaxPendingReceives { get; set; }
}
```

(4) 同步/异步消息接收

信道分发器的属性 `ReceiveSynchronously` 表示消息的接收应该采用同步还是异步的方式。如果该属性为 `True`，则采用同步方式进行消息接收。反之则采用异步的方式。信道分发器的 `ReceiveSynchronously` 实际上和接口 `System.ServiceModel.Channels.IBindingRuntimePreferences` 有关系。如下面的代码片段所示，该接口具有一个同名的只读属性。

```
public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public bool ReceiveSynchronously { get; set; }
}
public interface IBindingRuntimePreferences
{
    bool ReceiveSynchronously { get; }
}
```

所有系统绑定均实现了 `IBindingRuntimePreferences` 接口。对于常用的几个系统绑定，除了两个基于 MSMQ 的两个绑定（`NetMsmqBinding` 和 `MsmqIntegrationBinding`）的该属性返回 `True` 之外，其他均返回 `False`。我们可以应用终结点行为 `System.ServiceModel.Description.SynchronousReceiveBehavior` 将信道分发器的 `ReceiveSynchronously` 设置为 `True`。

(5) 事务控制

接下来介绍信道分发器的 4 个关于事务控制方面的属性。布尔类型的属性 `IsTransactedReceive` 表示是否将消息的接收操作纳入事务中进行。接口 `System.ServiceModel.Channels.ITransactedBindingElement` 与这个属性有关。如下面的代码片段所示，`ITransactedBindingElement` 具有一个对应的只读属性。

`ITransactedBindingElement` 是为绑定元素定义的接口。目前只有基于 MSMQ 的传输绑定元素 `MsmqBindingElementBase` 实现了这个接口。`MsmqBindingElementBase` 的 `IsTransactedReceive` 的值与 `ExactlyOnce` 具有相同的值。对于队列服务来说，需要通过事务性消息接收，避免接收消息因异常而丢失。

至于另一个属性 `MaxTransactedBatchSize`，则表示允许纳入同一个事务进行的最大消息接收操作数。该属性值可以通过终结点行为 `System.ServiceModel.Description.TransactedBatchingBehavior` 进行相应的设置，对应 `TransactedBatchingBehavior` 的 `MaxBatchSize` 属性。这两个属性只应用于基于 MSMQ 的队列服务，第 6 章“队列服务（Queued Service）”中有详细的介绍。

信道分发器的 `TransactionIsolationLevel` 和 `TransactionTimeout` 分别表示事务的隔离级别和超时时限, 它们可以通过 `ServiceBehaviorAttribute` 特性的同名属性进行相应的设置。

```
public class ChannelDispatcher : ChannelDispatcherBase
{
    //其他成员
    public bool IsTransactedReceive { get; set; }
    public int MaxTransactedBatchSize { get; set; }
    public IsolationLevel TransactionIsolationLevel { get; set; }
    public TimeSpan TransactionTimeout { get; set; }
}
public interface ITransactedBindingElement
{
    bool TransactedReceiveEnabled { get; }
}
public class TransactedBatchingBehavior : IEndpointBehavior
{
    //其他成员
    public int MaxBatchSize { get; set; }
}
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public IsolationLevel TransactionIsolationLevel { get; set; }
    public string TransactionTimeout { get; set; }
}
```

9.1.4 终结点分发器 (EndpointDispatcher)

作为 WCF 中的一个核心概念, 终结点在不同的语境中实际上指代不同的对象。站在服务描述的角度, 我们所说的终结点实际上是指 `ServiceEndpoint` 对象。对于 WCF 服务端运行时框架来说, 终结点实际上指代的是终结点分发器 (`EndpointDispatcher`)。而 `ServiceEndpoint` 与 `EndpointDispatcher` 是一一匹配的, 并且前者是创建后者的基础。

除了之前介绍的三个辅助信道分发器向匹配的终结点分发器实施消息路由的三个属性 (`AddressFilter`、`ContractFilter` 和 `FilterPriority`) 之外, 还可以通过属性 `ContractName` 和 `ContractNamespace` 得到服务契约的名称和命名空间, 通过 `EndpointAddress` 属性得到相应的终结点地址。将消息路由到该终结点分发器的信道分发器可以通过属性 `ChannelDispatcher` 获得。但是对于终结点分发器来说, 最重要的还是通过属性 `DispatchRuntime` 表示的分发运行时。

```
public class EndpointDispatcher
{
    //其他成员
    public string ContractName { get; }
    public string ContractNamespace { get; }

    public MessageFilter AddressFilter { get; set; }
```



```

public MessageFilter ContractFilter { get; set; }
public int FilterPriority { get; set; }

public ChannelDispatcher ChannelDispatcher { get; }
public DispatchRuntime DispatchRuntime { get; }
public EndpointAddress EndpointAddress { get; }
}

```

9.1.5 分发运行时 (DispatchRuntime)

毫不夸张地说,终结点分发器的分发运行时是 WCF 整个服务端运行时架构体系的核心,同时也是对 WCF 服务端服务模型进行扩展重点考虑的对象。分发运行时之所以具有如此重要的地位,是因为终结点分发器接收到从信道分发器路由的消息的整个处理是在分发运行时中进行的。终结点的分发运行时对应的类型为 `System.ServiceModel.Dispatcher.DispatchRuntime`。

1. 可扩展组件

如图 9-4 所示,众多的运行时组件“附着”在分发运行上,其中包括与安全有关的 `ServiceAuthenticationManager`、`ServiceAuthorizationManager`、`RoleProvider` 和 `AuthorizationPolicy` 列表;与服务实例激活有关的实例提供者、实例上下文提供者、单例实例上下文和实例上下文初始化器;与并发控制有关的同步上下文;可以对接收的请求消息和回复消息进行相应操作的消息检验器;运行时操作列表和用于选择操作的选择器。

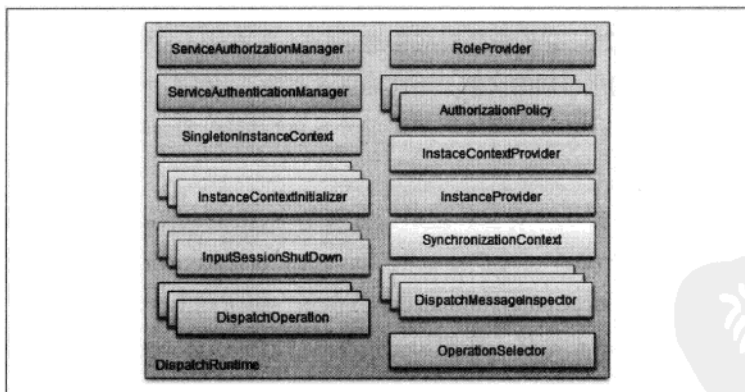


图 9-4 分发运行时结构

(1) 回调客户端运行时 (CallbackClientRuntime)

WCF 的服务端和客户端都具有一个运行时,分别叫做分发运行时和客户端运行时 (`ClientRuntime`)。正如分发运行时是服务端架构体系的核心一样,客户端运行时也是整个客户端架构体系的核心。服务端可以通过分发运行时的 `CallbackClientRuntime` 属性获得客户端运行时,对客户端的回调行为进行相应的控制。

```
public sealed class DispatchRuntime
{
    //其他成员
    public ClientRuntime CallbackClientRuntime { get; }
}
```

(2) 认证与授权

分发运行时具有一组与安全相关的组件, 包括分别用于进行认证和授权的 `ServiceAuthorizationManager` 和 `ServiceAuthorizationManager`, 实现 ASP.NET Roles 授权的 `RoleProvider`, 以及通过 `ExternalAuthorizationPolicies` 属性表示的自定义的授权策略。如果你阅读了本书的第7章“安全传输 (Transfer Security)”和第8章“授权与审核 (Authhorization and Auditing)”, 相信对这4个对象不会感到陌生。

```
public sealed class DispatchRuntime
{
    //其他成员
    public ServiceAuthorizationManager ServiceAuthorizationManager { get; set; }
    public ServiceAuthenticationManager ServiceAuthenticationManager { get; set; }
    public RoleProvider RoleProvider { get; set; }
    public ReadOnlyCollection<IAuthorizationPolicy> ExternalAuthorizationPolicies { get; set; }
}
```

(3) 服务实例、实例上下文和它们的提供者

被激活的服务实例被封装在实例上下文中, 而服务实例和服务实例上下文分别通过实例提供者和实例上下文提供者创建, 它们分别对应信道运行时的 `InstanceProvider` 和 `InstanceContextProvider` 属性。WCF 采用不同的实例上下文提供者实现了不同实例上下文模式。如果采用单例实例上下文模式, 单例服务实例上下文通过分发运行时的 `SingletonInstanceContext` 表示。

实例提供者和实例上下文提供者对应的接口, 分别是 `System.ServiceModel.Dispatcher.IInstanceProvider` 和 `System.ServiceModel.Dispatcher.IInstanceContextProvider`。除此之外, 分发运行时还通过只读属性 `InstanceContextInitializers` 提供了一组实例上下文初始化器, 用于对创建的实例上下文进行相应的初始化。实例初始化器对应的接口是 `System.ServiceModel.Dispatcher.IInstanceContextInitializer`。

```
public sealed class DispatchRuntime
{
    //其他成员
    public IInstanceContextProvider InstanceContextProvider { get; set; }
    public SynchronizedCollection<IInstanceContextInitializer> InstanceContextInitializers { get; }
    public InstanceContext SingletonInstanceContext { get; set; }
    public IInstanceProvider InstanceProvider { get; set; }
}

public interface IInstanceContextProvider
```

```

{
    InstanceContext GetExistingInstanceContext(Message message, IContextChannel
        channel);
    void InitializeInstanceContext(InstanceContext instanceContext, Message
        message, IContextChannel channel);
    bool IsIdle(InstanceContext instanceContext);
    void NotifyIdle(InstanceContextIdleCallback callback, InstanceContext
        instanceContext);
}
public interface IInstanceContextInitializer
{
    void Initialize(InstanceContext instanceContext, Message message);
}
public interface IInstanceProvider
{
    object GetInstance(InstanceContext instanceContext);
    object GetInstance(InstanceContext instanceContext, Message message);
    void ReleaseInstance(InstanceContext instanceContext, object instance);
}

```

(4) 输入会话关闭通知

分发运行时具有一个只读的集合类型的属性 `InputSessionShutdownHandlers`。该集合的元素是一个实现了 `System.ServiceModel.Dispatcher.IInputSessionShutdown` 接口的对象，我们姑且称之为输入会话关闭处理器。会话关闭处理器控制服务端关闭输入会话的方式。

在基于双工会话信道（实现了 `IDuplexSessionChannel` 接口）的消息交换中，如果服务调用信道的 `Receive` 方法进行消息的接收并返回 `Null`，意味着客户端关闭了当前输入会话。在这种情况下，WCF 会遍历当前分发运行时的输入会话关闭处理器并调用其 `DoneReceiving` 方法。而传入该方法的参数就是当前的双工会话信道，服务端可以在此方法中执行一些清理工作，或者在信道关闭之前向客户端发送一个响应消息。而另一个方法 `ChannelFaulted` 则会在信道出错的情况下被调用。

```

public sealed class DispatchRuntime
{
    //其他成员
    public SynchronizedCollection<IInputSessionShutdown>
        InputSessionShutdownHandlers { get; }
}
public interface IInputSessionShutdown
{
    void ChannelFaulted(IDuplexContextChannel channel);
    void DoneReceiving(IDuplexContextChannel channel);
}

```

(5) 同步上下文 (SynchronizationContext)

在默认的情况下，如果服务寄宿过程中的当前线程具有同步上下文，那么后续的处理将在该同步上下文中进行。在本书第4章“并发与限流 (Concurrency and Throttling)”中将此称为线程亲和性 (Thread Affinity)。可以通过分发运行时如下的 `SynchronizationContext` 属性得到该同步上下文对象。

可以在服务类型上应用 `ServiceBehaviorAttribute` 特性并通过指定 `UseSynchronizationContext` 属性为 `False` 来破坏线程亲和性, 以使并发的请求能够及时地得到处理。

```
public sealed class DispatchRuntime
{
    //其他成员
    public SynchronizationContext SynchronizationContext { get; set; }
}
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public bool UseSynchronizationContext { get; set; }
}
```

(6) 消息查看与修改

WCF 服务端运行时框架允许针对接收到的请求消息和生成的回复消息进行查看和修改, 可以通过创建自定义的分发消息检验器来实现。如下面的代码所示, 分发运行时具有一个 `MessageInspectors` 的集合属性, 其元素实现了 `System.ServiceModel.Dispatcher.IDispatchMessageInspector` 接口, 我们将其称为分发消息检验器。

接收的消息在被反序列化之前, 以及操作执行结果序列化生成回复消息之后, WCF 会遍历当前分发运行时的所有分发消息检验器, 调用其 `AfterReceiveRequest` 和 `BeforeSendReply` 方法分别对请求消息和回复消息进行“检验”。

```
public sealed class DispatchRuntime
{
    //其他成员
    public SynchronizedCollection<IDispatchMessageInspector> MessageInspectors
    { get; }
}
public interface IDispatchMessageInspector
{
    object AfterReceiveRequest(ref Message request, IClientChannel channel,
        InstanceContext instanceContext);
    void BeforeSendReply(ref Message reply, object correlationState);
}
```

(7) 操作与操作选择

服务端分发体系对消息请求处理最终体现在对相应操作方法的执行。在服务描述中的操作通过类型 `OperationDescription` 表示, 它在运行时被转换成通过类型 `DispatchOperation` 表示的分发操作。`DispatchRuntime` 的 `Operations` 属性就代表当前终结点的所有分发操作集合。这是一个类似于字典的集合类型, 而代表键值的字符串为操作的名称。

由于当前的分发运行时中具有多个分发操作, 因此必须具有某种机制实现根据接收的消息解析出对应的目标操作。这样一种操作的选择机制在 WCF 分发运行时中是通过一个被称为操作选择器 (`OperationSelector`) 的组件来实现的。操作选择器对应的接口为 `System.ServiceModel.Dispatcher.IDispatchOperationSelector`, 针对消息对操作的选择通过

SelectOperation 实现，该方法的返回值代表操作的名称。从该方法得到正确的操作名称后，WCF 就可以从 Operations 属性代表的操作列表中选择正确的分发操作了。

```
public sealed class DispatchRuntime
{
    //其他成员
    public SynchronizedKeyedCollection<string, DispatchOperation> Operations
    { get; }
    public IDispatchOperationSelector OperationSelector { get; set; }
}
public interface IDispatchOperationSelector
{
    string SelectOperation(ref Message message);
}
```

2. 可扩展属性

介绍了分发运行时可供扩展的组件之后，我们来看它具有哪些可以修改的属性，以及通过修改这些属性会对整个消息分发、实例上下文的激活及服务操作的执行等行为产生怎样的影响。

(1) 授权

DispatchRuntime 的 PrincipalPermissionMode 和 ImpersonateCallerForAllOperations 属性与授权相关。前者表示安全主体权限模式，后者表示是否以模拟的客户端 Windows 账号执行所有的操作。它们对应于服务行为 ServiceAuthorizationBehavior 的同名属性。相关的内容在本书第 8 章“授权与审核 (Authorization and Auditing)”中有详细的介绍。

```
public sealed class DispatchRuntime
{
    //其他成员
    public PrincipalPermissionMode PrincipalPermissionMode { get; set; }
    public bool ImpersonateCallerForAllOperations { get; set; }
}
public sealed class ServiceAuthorizationBehavior : IServiceBehavior
{
    //其他成员
    public bool ImpersonateCallerForAllOperations { get; set; }
    public PrincipalPermissionMode PrincipalPermissionMode { get; set; }
}
```

(2) 安全审核

DispatchRuntime 具有 4 个与安全审核相关的属性。其中 SecurityAuditLogLocation 决定了安全审核日志应该被写入 EventLog 的类型，即应用程序日志或（和）安全日志。SuppressAuditFailure 表示在进行安全日志写入的过程中抛出的异常是否会影响当前的应用。两个具有相同类型返回值的属性 MessageAuthenticationAuditLevel 和 ServiceAuthorizationAuditLevel 则表示应该在认证/授权成功或（和）失败时进行安全审核。这 4 个属性对应于服务行为 ServiceSecurityAuditBehavior 的同名属性。相关的内容请参阅本书第 8 章“授权与审

核 (Authorization and Auditing)”。

```
public sealed class DispatchRuntime
{
    //其他成员
    public AuditLogLocation SecurityAuditLogLocation { get; set; }
    public bool SuppressAuditFailure { get; set; }
    public AuditLevel ServiceAuthorizationAuditLevel { get; set; }
    public AuditLevel MessageAuthenticationAuditLevel { get; set; }
}

public sealed class ServiceSecurityAuditBehavior : IServiceBehavior
{
    //其他成员
    public AuditLogLocation AuditLogLocation { get; set; }
    public bool SuppressAuditFailure { get; set; }
    public AuditLevel MessageAuthenticationAuditLevel { get; set; }
    public AuditLevel ServiceAuthorizationAuditLevel { get; set; }
}
```

(3) 事务与会话

接下来介绍 4 个与会话和事务相关的属性。`AutomaticInputSessionShutdown` 表示服务端在客户端关闭输出会话 (Output Session) 的时候是否关闭输入会话 (Input Session)。关于会话的相关内容, 在上册的第 9 章“实例化与会话 (Instanting and Session)”中有详细的介绍。

属性 `ReleaseServiceInstanceOnTransactionComplete` 和 `TransactionAutoCompleteOnSessionClose` 分别表示在事务提交之后是否自动释放服务实例, 以及在会话关闭之后是否自动提交事务。这两个属性分别对应 `ServiceBehaviorAttribute` 的同名属性。而 `AutomaticInputSessionShutdown` 属性则对应于服务行为特性 `ServiceBehaviorAttribute` 的 `AutomaticSessionShutdown` 属性。

```
public sealed class DispatchRuntime
{
    //其他成员
    public bool AutomaticInputSessionShutdown { get; set; }
    public bool ReleaseServiceInstanceOnTransactionComplete { get; set; }
    public bool TransactionAutoCompleteOnSessionClose { get; set; }
    public bool IgnoreTransactionMessageProperty { get; set; }
}

[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public bool AutomaticSessionShutdown { get; set; }
    public bool ReleaseServiceInstanceOnTransactionComplete { get; set; }
    public bool TransactionAutoCompleteOnSessionClose { get; set; }
}
```

为了实现基于消息交换的事务传播, 事务本身被封装成一个 `System.ServiceModel.Channels.TransactionMessageProperty` 对象并置于请求消息之中。而分发运行时的 `IgnoreTransactionMessageProperty` 表示是否要忽略接收消息的 `TransactionMessageProperty` 消

息属性。

是否忽略消息中的 `IgnoreTransactionMessageProperty` 属性决定于终结点的两个要素，即绑定和契约。如果绑定不支持事务流转（Transaction Flow），则该属性返回 `True`。否则还需要分析服务契约中应用在操作上的 `TransactionFlowAttribute` 设置。关于事务在 WCF 中的应用，请参阅本书第 3 章“事务（Transaction）”。

（4）未处理操作

可以通过 `OperationContractAttribute` 将操作的 Action 定义成“*”。这样的操作被称为未处理操作（Unhandled Operation）。未处理操作最终也会被转换成 `DispatchOperation` 对象，并作为分发运行时的 `UnhandledDispatchOperation` 属性而存在。对于请求的消息，如果不能从分发运行时的 `Operations` 属性表示的操作列表中找到一个相匹配的操作，这个未处理操作会被选用。

```
public sealed class DispatchRuntime
{
    //其他成员
    public DispatchOperation UnhandledDispatchOperation { get; set; }
}
```

（5）SOAP 报头验证

`DispatchRuntime` 的 `ValidateMustUnderstand` 属性用于开启和关闭对到达的消息头的强制执行验证。在正常执行过程中，将消息头与 `UnderstoodHeaders` 属性进行比较，来确认是否由服务显式处理到达的消息。将此属性设置为 `false` 可以禁用此检查。当设置为 `True` 时，应用程序必须检查有 `MustUnderstand="true"` 标记的标头，如果其中一个或多个标头没有被理解，则返回错误。我们可以通过服务行为 `ServiceBehaviorAttribute` 来设置该属性。如下面的代码所示，`ServiceBehaviorAttribute` 具有同名的属性。

```
public sealed class DispatchRuntime
{
    //其他成员
    public bool ValidateMustUnderstand { get; set; }
}
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public bool ValidateMustUnderstand { get; set; }
}
```

（6）并发控制

最后一个属性 `ConcurrencyMode` 与并发有关，用于指定三种并发模式（Single、Reentrant 和 Multiple）中的某一种。它同样对应于 `ServiceBehaviorAttribute` 特性的同名属性。关于并发的相关内容，请参阅本书第 4 章“并发与限流（Concurrency and Throttling）”。

```
public sealed class DispatchRuntime
{
    //其他成员
    public ConcurrencyMode ConcurrencyMode { get; set; }
}
[AttributeUsage(AttributeTargets.Class)]
public sealed class ServiceBehaviorAttribute : Attribute, IServiceBehavior
{
    //其他成员
    public ConcurrencyMode ConcurrencyMode { get; set; }
}
```

9.1.6 分发操作 (DispatchOperation)

从服务描述的角度来看,操作是一个 `OperationDescription` 对象。而服务端分发运行时的操作则代表的是一个 `DispatchOperation` 对象。当 `ServiceHost` 被正常开启时, `OperationDescription` 对象转换成代表分发操作的 `DispatchOperation` 对象。`DispatchRuntime` 的 `Operations` 属性表示当前终结点分发器中的所有分发操作列表。

图 9-5 为你展示了附加在一个分发操作上面的可扩展组件,其中包括一组进行执行上下文初始化的调用上下文初始化器 (`CallContextInitializer`); 一组对操作方法参数进行检验的参数检验器 (`ParameterInspector`); 用于消息序列化和反序列化的分发消息格式化器 (`DispatchMessageFormatter`); 用于操作方法的执行的操作调用器 (`OperationInvoker`)。接下来分别介绍这些组件及 `DispatchOperation` 用于控制操作执行的相关属性。

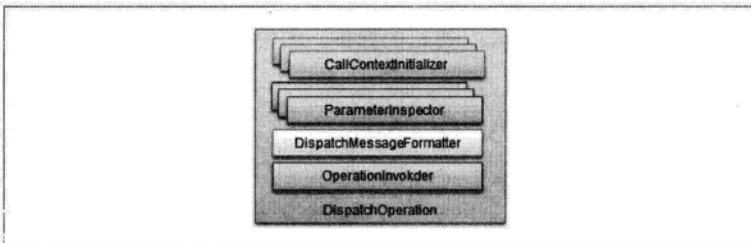


图 9-5 附加在一个分发操作上面的可扩展组件

1. 序列化与反序列化

服务操作的执行最终体现在执行服务实例的某个相应的操作方法。调用方法需要传入参数,而服务调用信息均被封装在请求消息中。在真正执行方法调用之前,首要的任务就是从请求消息中提取相应的信息并将其反序列化成方法的输入参数。当操作方法被正确执行后,如果要将执行结果正确地回复给客户端,需要将方法的返回值(或者 `ref/out` 参数)序列化成消息。

WCF 服务端通过一个实现了接口 `System.ServiceModel.Dispatcher.IDispatchMessageFormatter` 的被称为分发消息格式化器的组件完成消息的序列化和反序列化。如下面的代码

所示, 针对请求消息的反序列化和针对回复消息的序列化分别实现在 `DeserializeRequest` 和 `SerializeReply` 方法中。分发操作的 `Formatter` 属性决定最终采用的消息格式化器。

与序列化相关的还有两个布尔类型的属性 `DeserializeRequest` 和 `SerializeReply`。它们分别表示是否需要进行请求消息的反序列化和回复消息的序列化。如果操作方法具有一个唯一的类型为 `Message` 的参数, 那么对请求消息的反序列化是不需要的。如果操作方法的返回值 (并且没有 `ref/out` 参数) 类型为 `Message`, 那么也不需要进行对回复消息的序列化。关于针对分发消息格式化器在服务端如何实施序列化和反序列化, 可以参考上册的第 5 章“序列化 (Serialization)”。

如果出现异常, 可以针对定义在操作上的错误契约抛出 `FaultException<TDetail>` 异常。为了将封装错误消息的 `TDetail` 对象序列化成错误消息。一些辅助性信息被封装在一个 `FaultContractInfo` 对象中, 而 `DispatchOperation` 的 `FaultContractInfos` 表示与该操作的所有错误契约相关的 `FaultContractInfo` 集合。关于错误消息的序列化, 可以参考本书的第 1 章“异常处理 (Exception Handling)”。

```
public sealed class DispatchOperation
{
    //其他成员
    public bool DeserializeRequest { get; set; }
    public bool SerializeReply { get; set; }
    public SynchronizedCollection<FaultContractInfo> FaultContractInfos { get; }
    public IDispatchMessageFormatter Formatter { get; set; }
}
public interface IDispatchMessageFormatter
{
    void DeserializeRequest(Message message, object[] parameters);
    Message SerializeReply(MessageVersion messageVersion, object[] parameters,
        object result);
}
```

2. 调用上下文初始化

如下面的代码所示, `DispatchOperation` 具有一个只读的集合属性 `CallContextInitializers`。该集合中的元素是实现了具有如下定义的 `System.ServiceModel.Dispatcher.ICallContextInitializer` 接口的对象, 我们将其称为调用上下文初始化器 (`CallContextInitializer`)。

在操作方法执行前后, WCF 会遍历当前分发操作的所有调用上下文, 分别调用其 `BeforeInvoke` 和 `AfterInvoke` 方法以实现针对当前执行上下文的初始化和清理工作。`BeforeInvoke` 方法的返回值会作为输入参数传递给 `AfterInvoke` 方法。

```
public sealed class DispatchOperation
{
    //其他成员
    public SynchronizedCollection<ICallContextInitializer> CallContextInitializers
    { get; }
}
public interface ICallContextInitializer
```

```

{
    void AfterInvoke(object correlationState);
    object BeforeInvoke(InstanceContext instanceContext, IClientChannel channel,
        Message message);
}

```

3. 参数的检验

分发操作具有如下一个集合类型的只读属性 `ParameterInspectors`，其元素是一组被称为分发参数检验器的对象。分发参数检验器实现了 `System.ServiceModel.Dispatcher.IParameterInspector` 接口，其 `BeforeCall` 和 `AfterCall` 方法分别在操作执行前后对输入参数和返回结果实施所谓的参数检验。操作名称和作为输入参数或返回结果的对象会被传入这两个方法。而 `AfterCall` 的参数 `correlationState` 代表执行 `BeforeCall` 方法的返回值。

```

public sealed class DispatchOperation
{
    //其他操作
    public SynchronizedCollection<IParameterInspector> ParameterInspectors
}
public interface IParameterInspector
{
    void AfterCall(string operationName, object[] outputs, object returnValue,
        object correlationState);
    object BeforeCall(string operationName, object[] inputs);
}

```

4. 服务实例的释放

服务实例释放的时机涉及分发操作的 `ReleaseInstanceBeforeCall` 和 `ReleaseInstanceAfterCall` 属性。它们分别表示服务实例的释放应该在操作方法执行前还是执行后进行。服务实例的释放是通过调用服务实例上下文的 `ReleaseServiceInstance` 方法实现的，而该方法一般来说通过调用实例提供者的 `ReleaseInstance` 方法来真正地释放服务实例。

```

public sealed class DispatchOperation
{
    //其他操作
    public bool ReleaseInstanceAfterCall { get; set; }
    public bool ReleaseInstanceBeforeCall { get; set; }
}
public sealed class InstanceContext : CommunicationObject,
    IExtensibleObject<InstanceContext>
{
    //其他成员
    public void ReleaseServiceInstance();
}
public interface IInstanceProvider
{
    //其他成员
    void ReleaseInstance(InstanceContext instanceContext, object instance);
}

```

这两个值可以通过操作行为 `OperationBehaviorAttribute` 特性来控制。如下面的代码所示，它具有一个类型为 `ReleaseInstanceMode` 枚举的 `ReleaseInstanceMode` 属性。我们可以将该特性应用到相应的操作方法上并指定相应的实例释放模式来控制服务实例的回收是在操作调用前还是调用后执行。

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute,
IOperationBehavior
{
    //其他成员
    public ReleaseInstanceMode ReleaseInstanceMode { get; set; }
}
public enum ReleaseInstanceMode
{
    None,
    BeforeCall,
    AfterCall,
    BeforeAndAfterCall
}
```

5. 事务

分发操作的两个布尔类型的属性 `TransactionRequired` 和 `TransactionAutoComplete` 与事务相关。前者表示当前的操作是否应该在事务中执行，后者表示当操作执行之后是否自动提交事务。可以通过操作行为 `OperationBehaviorAttribute` 的 `TransactionScopeRequired` 和 `TransactionAutoComplete` 属性来控制它们。

```
public sealed class DispatchOperation
{
    //其他操作
    public bool TransactionAutoComplete { get; set; }
    public bool TransactionRequired { get; set; }
}
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute, IOperationBehavior
{
    //其他成员
    public bool TransactionAutoComplete { get; set; }
    public bool TransactionScopeRequired { get; set; }
}
```

6. 操作的执行

操作的执行最终落在了一个被称为操作执行器 (`OperationInvoker`) 的组件上，这个对象可以通过属性 `Invoker` 获取和设置。操作执行器具有自己的接口 `System.ServiceModel.Dispatcher.IOperationInvoker`。如下面的代码所示，`IOperationInvoker` 定义了 `Invoke` 和 `InvokeBegin/InvokeEnd` 方法，分别以同步和异步的方式完成操作的执行。

`Invoke/InvokeBegin` 方法中的输入参数 `instance` 和 `inputs` 分别表示用于通过

`InstanceProvider` 提供的服务实例和输入参数。而 `Invoke/InvokeEnd` 的返回值和输出参数 (outputs) 表示操作执行后得到的结果。

```
public sealed class DispatchOperation
{
    //其他操作
    public IOperationInvoker Invoker { get; set; }
}
public interface IOperationInvoker
{
    object[] AllocateInputs();
    object Invoke(object instance, object[] inputs, out object[] outputs);
    IAsyncResult InvokeBegin(object instance, object[] inputs, AsyncCallback
        callback, object state);
    object InvokeEnd(object instance, out object[] outputs, IAsyncResult result);
    bool IsSynchronous { get; }
}
```

至于应该采用同步还是异步的操作执行方式, 取决于只读属性 `IsSynchronous`。而 `AllocateInputs` 方法用于返回代表当前参数的数组, 并最终作为输入参数传入 `Invoke` 或者 `InvokeBegin` 方法中。

7. 参数和返回值的释放

如果服务操作成功执行, 并且执行的结果被序列化到回复消息中, 那么无论是作为参数的对象还是作为返回值的对象都变成了“垃圾对象”。在正常的情况下它们最终会被垃圾回收。如果这些对象引用一些需要释放的资源, 有可能造成内存泄漏。

我们在设计这种类型的时候, 一般会实现 `IDisposable` 接口, 并将资源释放操作实现在 `Dispose` 方法中。而 `DispatchOperation` 的 `AutoDisposeParameters` 属性决定了对于实现了 `IDisposable` 接口的类型的参数和返回值, 是否需要最终调用它们的 `Dispose` 方法。

我们可以通过操作行为特性 `OperationBehaviorAttribute` 的同名属性控制分发操作的 `AutoDisposeParameters` 属性值。在默认的情况下, `AutoDisposeParameters` 属性为 `True`。如果你希望直接避免参数和返回值的释放操作, 可以通过该特性将属性设置为 `False`。

```
public sealed class DispatchOperation
{
    //其他操作
    public bool AutoDisposeParameters { get; set; }
}
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute, IOperationBehavior
{
    //其他成员
    public bool AutoDisposeParameters { get; set; }
}
```

8. 身份模拟

最后一个类型为 `ImpersonationOption` 的属性 `Impersonation` 在本书第 8 章“授权与审核 (Authorization and Auditing)”中已经详细介绍了，用于表示是否在模拟客户端身份上下文中执行服务操作。它对应操作行为特性 `OperationBehaviorAttribute` 的同名属性。

```
public sealed class DispatchOperation
{
    //其他操作
    public ImpersonationOption Impersonation { get; set; }
}
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationBehaviorAttribute : Attribute, IOperationBehavior
{
    //其他成员
    public ImpersonationOption Impersonation { get; set; }
}
```

9.2 客户端架构体系的构建

WCF 通过服务代理进行服务的调用，`ChannelFactory<TChannel>` 是创建服务代理的工厂。对于 WCF 客户端应用编程接口来说，`ChannelFactory<TChannel>` 是一个核心类型。下面先来看看 `ChannelFactory<TChannel>` 在创建过程中都发生了什么。

9.2.1 创建 `ChannelFactory<TChannel>`

服务调用的本质实际上是针对服务的某个终结点的调用，说得具体些，应该是客户端通过相匹配的终结点调用服务的终结点。终结点具有 ABC 三要素，这里所说的“相匹配”的终结点具体体现了三要素的匹配。而服务调用最终体现在消息交换上。接下来我们从消息交换的角度来谈谈匹配终结点在服务调用中的必要性。

- **地址 (Address)**：地址作为调用服务的唯一标识并代表了服务所在的网络位置，客户端终结点必须具有一个正确的地址才能确保请求的消息被发送到正确的目的地。
- **绑定 (Binding)**：作为信道层的缔造者，绑定最终创建了用于实现消息处理和传输的信道栈。客户端必须具有一个与服务端一致的信道栈，才能确保消息的一致性处理。
- **契约 (Contract)**：契约最终决定了基于某个操作的服务调用应该采用的消息交换模式，以及参与消息交换的消息本身所具有的结构。为了让客户端和服务端就此达成一致，必须要求双方采用等效的契约。

`ChannelFactory<TChannel>` 本身就是基于某个具体的终结点创建的。我们可以通过编程的方式（构造函数）指定终结点的三要素，也可以将此三要素定义在配置文件中，通过终结点配置名称（构造函数的 `endpointConfigurationName` 参数）来创建它。

```

public abstract class ChannelFactory : CommunicationObject, IChannelFactory,
ICommunicationObject, IDisposable
{
    //其他成员
    public ServiceEndpoint Endpoint { get; }
}
public class ChannelFactory<TChannel> : ChannelFactory,
IChannelFactory<TChannel>, IChannelFactory, ICommunicationObject
{
    //其他成员
    public ChannelFactory(string endpointConfigurationName);
    protected ChannelFactory(Type channelType);
    public ChannelFactory(Binding binding, EndpointAddress remoteAddress);
    public ChannelFactory(Binding binding, string remoteAddress);
    public ChannelFactory(string endpointConfigurationName, EndpointAddress
        remoteAddress);
}

```

当我们通过调用构造函数创建一个 `ChannelFactory<TChannel>` 对象后, WCF 会根据指定的终结点创建一个 `ServiceEndpoint` 对象。而该 `ServiceEndpoint` 就是 `ChannelFactory<TChannel>` 对象的核心, 只读属性 `Endpoint` 返回的也就是该对象。`ServiceEndpoint` 在 `ChannelFactory<TChannel>` 中的结构分布如图 9-6 所示。

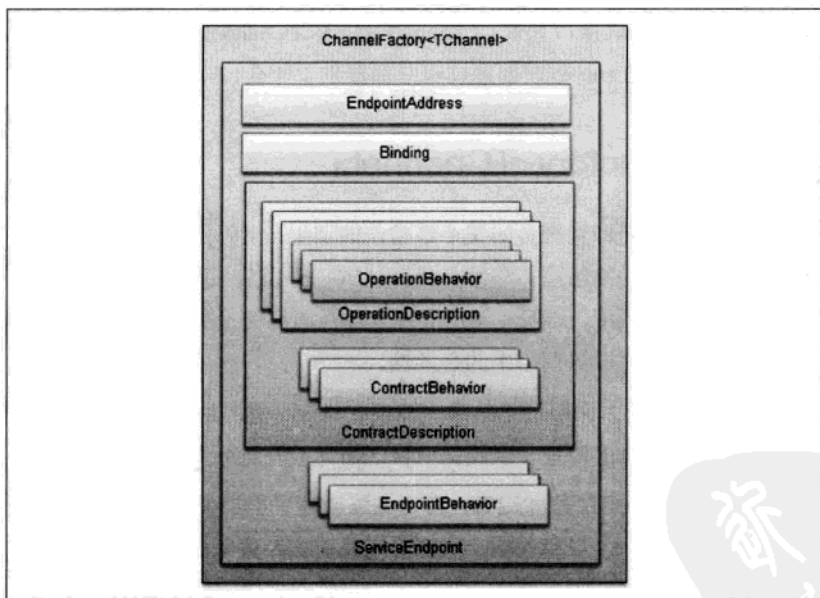


图 9-6 `ChannelFactory<TChannel>` 与 `ServiceEndpoint`

当我们开启 `ChannelFactory<TChannel>` 的时候, WCF 会根据之前创建的 `ServiceEndpoint` 来构建客户端的运行时架构体系。图 9-7 揭示了 WCF 客户端框架体系的大体结构。该架构体系以客户端运行时 (`ClientRuntime`) 为核心。

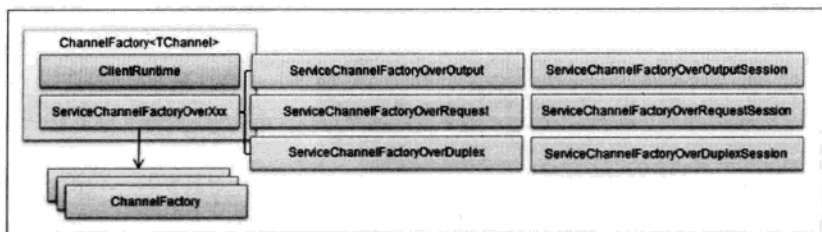


图 9-7 客户端架构体系

`ChannelFactory<TChannel>` 开启会伴随着绑定的 `BuildChannelFactory<TChannel>` 方法的调用。其结果是绑定的所有绑定元素的同名方法被调用并生成各自的信道工厂，这些信道工厂按照定义绑定元素的顺序组成一个链条。而连接它和客户端运行时的是一个名为 `ServiceChannelFactoryOverXxx` 的对象。根据由消息交换模式决定的信道形状（`Channel Shape`）和是否支持会话，`ServiceChannelFactoryOverXxx` 具体可分为 6 种：

- `ServiceChannelFactoryOverOutput`
- `ServiceChannelFactoryOverOutputSession`
- `ServiceChannelFactoryOverRequest`
- `ServiceChannelFactoryOverRequestSession`
- `ServiceChannelFactoryOverDuplex`
- `ServiceChannelFactoryOverDuplexSession`

9.2.2 客户端运行时（`ClientRuntime`）

客户端运行时是与分发运行时相匹配的运行时，也是整个客户端框架体系的核心。我们针对客户端的扩展会频繁地使用到客户端运行时。如图 9-8 所示，客户端运行时的核心是一组代表客户端操作的 `ClientOperation` 集合，而 `ClientOperationSelector` 用于实现操作的选择。除了一组信道初始化器（`ChannelInitializer`）之外，还具有一组交互式信道初始化器（`InteractiveChannelInitializer`）。而客户端信道检验器则旨在针对发送的请求消息和接收的回复消息的查看和修改。

1. 信道初始化

客户端运行时具有如下两个基于信道初始化器列表的属性，分别是 `ChannelInitializers` 和 `InteractiveChannelInitializers`。包含在 `ChannelInitializers` 集合中的是一般意义上的信道初始化器，而包含在 `InteractiveChannelInitializers` 集合中的则被称为交互式信道初始化器。

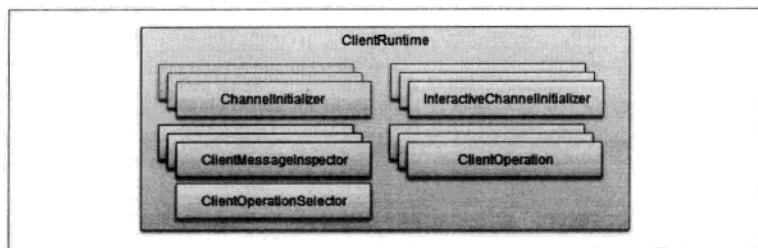


图 9-8 客户端运行时

属性 `ChannelInitializers` 表示的集合元素实现了 `System.ServiceModel.Dispatcher.IChannelInitializer` 接口。如下面的代码片段所示, `IChannelInitializer` 接口具有唯一的 `Initialize` 方法, 用于实现针对客户端信道 (`IClientChannel` 对象) 进行初始化。当客户端信道被创建之后, WCF 会遍历所有的 `IChannelInitializer` 对象并将信道作为参数调用它们的 `Initialize` 方法。

`InteractiveChannelInitializers` 属性代表的集合则包括另外一种旨在提供交互式的信道初始化的方式, 比如弹出一个供输入用户名和密码的 UI 以指定客户端凭证。包含在该集合中的交互式信道初始化器实现了接口 `System.ServiceModel.Dispatcher.IInteractiveChannelInitializer`。如下面的代码片段所示, 该接口具有一组以异步模式定义的方法 `BeginDisplayInitializationUI` 和 `EndDisplayInitializationUI`, 用于显示相应的基于交互式需要而定制的 UI。

```

public sealed class ClientRuntime
{
    //其他成员
    public SynchronizedCollection<IChannelInitializer> ChannelInitializers { get; }
    public SynchronizedCollection<IInteractiveChannelInitializer>
        InteractiveChannelInitializers { get; }
}

public interface IChannelInitializer
{
    void Initialize(IClientChannel channel);
}

public interface IInteractiveChannelInitializer
{
    IAsyncResult BeginDisplayInitializationUI(IClientChannel channel,
        AsyncCallback
            callback, object state);
    void EndDisplayInitializationUI(IAsyncResult result);
}

```

2. 消息检验

客户端运行时的客户端消息检验器 (`ClientMessageInspector`) 就相当于分发运行时的分发消息检验器 (`DispatchMessageInspector`)。后者在服务端实现针对接收到的请求消息和发送的回复消息的查看和修改, 前者则在客户端实现针对发送的请求消息和接收到的回复消息

的查看和修改。

如下面的代码所示,客户端运行时和分发运行时均具有一个 MessageInspectors 属性代表消息检验器集合。所不同的是,这里的消息检验器为实现了 System.ServiceModel.Dispatcher.IClientMessageInspector 接口的客户端消息检验器。

WCF 在发送请求消息之前和接收到回复消息之后会遍历当前客户端运行时的所有消息检验器,并分别调用其 BeforeSendRequest 和 AfterReceiveReply,以实现针对请求消息和回复消息的消息检验。BeforeSendRequest 方法执行的返回值会作为调用 AfterReceiveReply 方法的参数 (correlationState)。

```
public sealed class ClientRuntime
{
    //其他成员
    public SynchronizedCollection<IClientMessageInspector> MessageInspectors
    { get; }
}
public interface IClientMessageInspector
{
    void AfterReceiveReply(ref Message reply, object correlationState);
    object BeforeSendRequest(ref Message request, IClientChannel channel);
}
```

3. 操作和操作选择

作为操作描述的 OperationDescription 对象在服务端运行时被转化成 DispatchOperation 对象,而客户端则被转化成 ClientOperation 对象。客户端运行时的 Operations 属性包含一个 ClientOperation 的列表,用于表示定义在当前终结点契约中的所有操作。

分发运行时通过分发操作选择器实现针对分发操作的选择,与之类似,客户端运行时针对客户端操作的选择则是通过实现了如下 System.ServiceModel.Dispatcher.IClientOperationSelector 接口的客户端操作选择器来实现的。客户端运行时的 OperationSelector 属性用于设置和返回这个操作选择器。

至于客户端运行时的另一个代表未处理操作的只读属性 UnhandledClientOperation,和分发运行时的同名属性类似。在定义服务契约时,若将应用在方法上的 OperationContractAttribute 特性的 Action 属性设置成“*”,那么属性 UnhandledClientOperation 返回的就是对应于这个操作的 ClientOperation 对象。

```
public sealed class ClientRuntime
{
    //其他成员
    public SynchronizedKeyedCollection<string, ClientOperation> Operations { get; }
    public IClientOperationSelector OperationSelector { get; set; }
    public ClientOperation UnhandledClientOperation { get; }
}
public interface IClientOperationSelector
```

```
{
    string SelectOperation(MethodBase method, object[] parameters);
    bool AreParametersRequiredForSelection { get; }
}
```

9.2.3 客户端操作 (ClientOperation)

客户端运行时的核心是一组代表定义在当前终结点契约中的所有操作的客户端操作 (ClientOperation) 列表, 很有必要对 ClientOperation 进行深入的了解。下面的代码片段列出了定义在 ClientOperation 中的主要属性。

```
public sealed class ClientOperation
{
    //其他成员

    public string Name { get; }
    public string Action { get; }
    public string ReplyAction { get; }
    public bool IsOneWay { get; set; }

    public MethodInfo SyncMethod { get; set; }
    public MethodInfo BeginMethod { get; set; }
    public MethodInfo EndMethod { get; set; }

    public bool SerializeRequest { get; set; }
    public bool DeserializeReply { get; set; }

    public bool IsInitiating { get; set; }
    public bool IsTerminating { get; set; }

    public IClientMessageFormatter Formatter { get; set; }
    public SynchronizedCollection<FaultContractInfo> FaultContractInfos { get; }
    public SynchronizedCollection<IParameterInspector> ParameterInspectors { get; }
}
```

在定义服务契约的时候, 我们通过应用 OperationContractAttribute 特性将定义在契约接口或类中的某个方法定义成服务操作。当我们针对某个终结点创建 ChannelFactory<TChannel>的时候, 反映操作描述的 OperationDescription 被创建出来。而当我们开启了 ChannelFactory<TChannel>之后, OperationDescription 对象被转变成真正的运行时操作对象 ClientOperation。所以 ClientOperation 主要来源于 OperationDescription, 而最终决定于应用在操作方法上的 OperationContractAttribute 的定义。

ClientOperaiton 的 Name、Action、ReplayAction 和 IsOneway 对应于 OperationContractAttribute 特性的同名属性。而 SyncMethod 和 BeginMethod/EndMethod 则表示同步和异步调用时对应的 MethodInfo 对象。当我们通过将应用在操作方法的 OperationContractAttribute 特性的 AsyncPattern 属性设置成 true 来定义异步模式的服务操作时, BeginMethod/EndMethod 属性对应于 BeginXxx/EndXxx 方法。

布尔类型的属性 SerializeRequest/DeserializeReply 分别表示是否需要对请求消息进行序

列化及对回复消息进行反序列化。如果操作仅仅具有一个唯一的类型为 `Message` 的参数，就无须对参数进行序列化。相应地，如果返回值（或者 `ref/out` 参数）也是一个唯一的 `Message` 对象，那么也无须对回复消息进行反序列化。

另外一组布尔类型的属性 `IsInitiating/IsTerminating` 对应于 `OperationContractAttribute` 特性的同名属性，表示在支持会话（`Session`）的情况下，相应的操作是否是用于初始化/终止会话的操作。

`DispatchOperation` 使用实现了接口 `IDispatchMessageFormatter` 的分发消息格式化器进行请求消息的反序列化和回复消息的序列化。与之类似，客户端操作则使用客户端消息格式化器实现请求消息的序列化和回复消息的反序列化。

客户端消息格式化器实现了具有如下定义的 `System.ServiceModel.Dispatcher.IClientMessageFormatter` 接口。上述的序列化和反序列化的操作分别实现在 `SerializeRequest` 和 `DeserializeReply` 方法中。而真正被使用的 `ClientMessageFormatter` 定义在 `ClientOperation` 的 `Formatter` 属性中。

```
public interface IClientMessageFormatter
{
    object DeserializeReply(Message message, object[] parameters);
    Message SerializeRequest(MessageVersion messageVersion, object[] parameters);
}
```

最后一个 `FaultContractInfos` 属性表述一个元素为 `FaultContractInfo` 的集合。如果你阅读了本书的第 1 章“异常处理（`Exception Handling`）”，应该知道该集合最终用于在出现异常时辅助实现针对错误消息（`Fault Message`）的序列化和反序列化。

和 `DispatchOperation` 一样，`ClientOperation` 具有一个 `ParameterInspectors` 属性表示一组参数检验器列表。`DispatchOperation` 和 `ClientOperation` 的参数检验器实现了相同的接口 `IParameterInspector`。我们可以自定义参数检验器实现服务调用前对输入参数的验证，以及服务调用后对返回值和输出参数的验证。

9.2.4 服务代理与服务调用

当基于某个终结点创建的 `ChannelFactory<TChannel>` 被开启之后，位于服务模型层的客户端运行时框架被成功构建。站在编程的角度看 `ChannelFactory<TChannel>`，它就是一个创建用于服务调用的服务代理对象的工厂。由于服务调用需要借助于服务代理来完成，因此我们很有必要从整个客户端运行架构层面来了解服务代理和基于服务代理的服务调用是如何实现的。

1. 服务代理是一个透明代理

如果你阅读了上册的第 8 章“客户端（`Client`）”，应该知道通过 `ChannelFactory<TChannel>`

创建的服务代理对象是一个“透明代理 (Transparent Proxy)”对象。而这可以通过调用 `System.Runtime.Remoting.RemotingServices` 的静态方法 `IsTransparent Proxy` 来检验。为此笔者写了如下一段简单的检验程序，输出的结果证实了“服务代理是透明代理”的结论。

```
using (ChannelFactory<ICalculate> channelFactory = new
ChannelFactory<ICalculate>("calculateservice"))
{
    ICalculate calculator = channelFactory.CreateChannel();
    bool isTransparentProxy = RemotingServices.IsTransparentProxy(calculator);
    Console.WriteLine("服务代理是一个透明代理? {0}.",
        isTransparentProxy ? "Yes" : "No");
}
```

输出结果:

服务代理是一个透明代理? Yes.

既然服务代理是一个透明代理，那么它一定对应了具体的真实代理 (RealProxy)。实际上服务代理对象内部具有一个类型为 `ServiceChannelProxy` 的对象作为其真实代理对象。`ServiceChannelProxy` 是 WCF 中的一个继承自 `System.Runtime.Remoting.Proxies.RealProxy` 的类型，而其核心则是一个类型为 `ServiceChannel` 的对象。`ServiceChannelProxy` 和 `ServiceChannel` 均是定义在 `System.ServiceModel.Channels` 命名空间下的内部 (Internal) 类型。

当我们使用 `ChannelFactory<TChannel>` 创建一个服务代理的时候，WCF 会根据客户端运行时创建一个 `ServiceChannel` 对象，并且调用之前创建的信道工厂栈并最终创建信道栈。由于 `ServiceChannel` 同时引用着代表服务模型层核心的客户端运行时和信道层的信道栈，因此可以说 `ServiceChannel` 是连接 WCF 客户端服务模型层与信道层的纽带。当 `ServiceChannel` 被成功创建后，WCF 会基于该对象创建 `ServiceChannelProxy` 对象。最后返回这个真实代理对象的透明代理。

当我们通过显式 (将服务代理对象转换成 `ICommunicationObject` 类型，并显式调用其 `Open` 方法) 或者隐式 (如果服务代理在未开启的状态下用于服务调用，在进行服务调用之前会被隐式地开启) 方法开启时，整个信道栈会被开启。图 9-9 揭示了服务代理 (透明代理)、`ServiceChannelProxy` (真实代理)、`ServiceChannel`、`ClientRuntime` 和信道栈之间的关系。

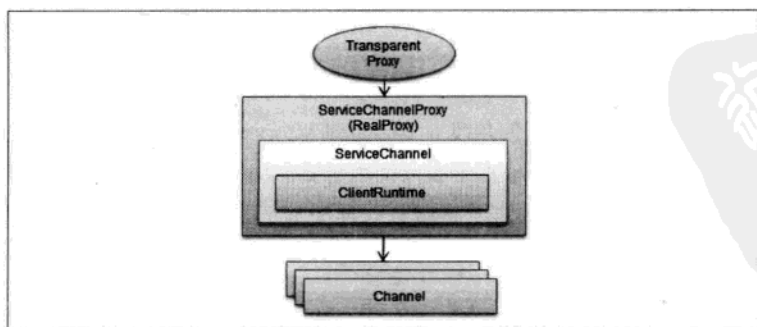


图 9-9 从服务代理到信道栈

2. 服务调用的流程

由于服务代理是一个透明代理，因此针对它的任何一个方法调用都会最终转换到对其真实代理（ServiceChannelProxy）的 Invoke 方法的调用。所以 ServiceChannelProxy 会接管所有针对于服务代理对象的服务调用，并最终将调用递交给内部的 ServiceChannel 处理。

接下来简单地介绍一下针对一次简单的针对服务代理的服务调用，ServiceChannel 在其内部是按照怎样的流程来处理的。实际上，相同的内容已经出现在了上册的第 8 章“客户端（Client）”中。图 9-10 体现了 ServiceChannel 进行服务调用的整个流程（以请求/回复消息交换模式为例）。

（1）操作选择：如果当前客户端运行时的 OperationSelector 属性具有一个操作选择器，则调用其 SelectOperation 方法获取针对当前服务调用的客户端操作。

（2）输入参数检验：遍历当前客户端运行时的 ParameterInspectors 属性表示的参数检验器列表，调用每个参数检验器的 BeforeCall 方法对输入参数实施检验。

（3）序列化请求消息：通过当前客户端操作的 SerializeRequest 属性判断是否需要请求消息的序列化。如果需要，则通过操作的 Formatter 属性获取消息格式化器，并调用 SerializeRequest 方法将以方法调用形式体现的服务调用序列化成请求消息。

（4）请求消息检验：遍历当前操作的 MessageInspectors 属性表示的消息检验器列表，并调用每个消息检验器的 BeforeSendRequest 方法对请求消息实施发送前的检验。

（5）请求消息的发送和回复消息的接收：将请求消息递交给信道层进行进一步处理，经过编码后的请求消息通过传输信道发送到服务端并等待回复。当回复消息抵达客户端后，信道层对其进行接收、解码相应的处理。

（6）回复消息的检验：遍历当前操作的 MessageInspectors 属性表示的消息检验器列表，并调用每个消息检验器的 AfterReceiveReply 方法对回复消息实施发送前的检验。

（7）反序列化回复消息：通过当前操作的 DeserializeReply 属性判断是否需要回复消息的反序列化。如果需要，则通过当前 ClientOperation 的 Formatter 属性获取消息格式化器，并调用 DeserializeReply 方法将包含在回复消息中的调用结果反序列化成方法调用的返回值或者 ref/out 参数对象。

（8）检验返回值（或者 ref/out 参数）：遍历当前客户端运行时的 ParameterInspectors 属性表示的参数检验器列表，调用每个参数检验器的 AfterCall 方法对返回值或者 ref/out 参数对象进行检验。

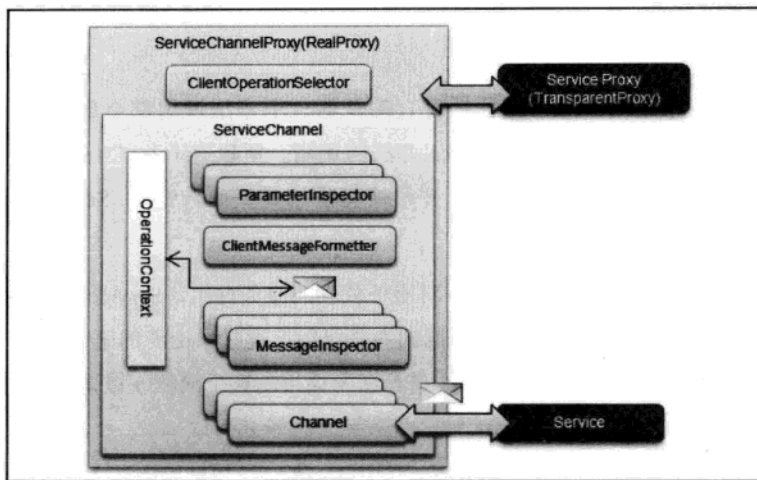


图 9-10 WCF 客户端进行服务调用的流程

9.3 通过定义四种行为对 WCF 的扩展

整个 WCF 框架由两个基本的层次构成，即服务模型层和信道层。对信道层的扩展主要体现在针对绑定的扩展，具体来说就是自定义绑定元素，以及相关的信道管理器（信道监听器和信道工厂）、信道来改变对消息的处理和传输方式。

而对于服务模式型层的扩展，则主要体现服务端和客户端运行时框架的定制，进而让 WCF 按照我们希望的方式进行运作。由于整个运行时框架由一系列的可扩展组件构成，并且大部分运行时属性也可以改写，所以针对服务模型层的扩展具体体现在：根据具体的需要定义相应的组件，并以某种情形将这些自定义的组件应用到运行时框架相应的地方，或者按照我们希望的方式定制相应的运行时属性。

WCF 为我们提供两种典型的应用自定义组件或者修改运行时属性的形式，即通过定义相应的行为（服务行为、终结点行为、契约行为和操作行为）和自定义 `ServiceHost`，我们把它们称为 WCF 的两种扩展形式。下面先来介绍 WCF 的四种行为。

9.3.1 WCF 四种类型的行为

作为最常用的扩展方式，WCF 的四种行为的使用主要体现在两个方面：（1）WCF 自身提供的很多特性和功能是通过行为的方式来实现的；（2）作为使用 WCF 的应用，可以通过自定义的行为来解决具体的需求。

根据应用目标的范围的不同，WCF 具有四种类型的行为：服务行为、终结点行为、契约行为和操作行为，它们的名称体现了行为本身的作用范围。对于 WCF 的这四种行为，读

者肯定不会感到陌生。因为 WCF 提供的很多功能和特性都是通过相应的行为来实现的，所以基本上在每一章中我们都会涉及针对相应行为的介绍。不过为了让读者对行为的本质有一个深刻的认识，能够帮助读者选择正确的行为类型来实现扩展，我们对 WCF 的四种行为做一个系统的介绍。

WCF 的四种类型的行为均具有各自的接口。除了服务行为只是应用于服务端之外，终结点行为、契约行为和操作行为都可以同时应用于服务端和客户端。所以后三者具有相同的方法定义。

```
public interface IEndpointBehavior
{
    void AddBindingParameters(ServiceEndpoint endpoint, BindingParameterCollection bindingParameters);
    void ApplyClientBehavior(ServiceEndpoint endpoint, ClientRuntime clientRuntime);
    void ApplyDispatchBehavior(ServiceEndpoint endpoint, EndpointDispatcher endpointDispatcher);
    void Validate(ServiceEndpoint endpoint);
}

public interface IContractBehavior
{
    void AddBindingParameters(ContractDescription contractDescription, ServiceEndpoint endpoint, BindingParameterCollection bindingParameters);
    void ApplyClientBehavior(ContractDescription contractDescription, ServiceEndpoint endpoint, ClientRuntime clientRuntime);
    void ApplyDispatchBehavior(ContractDescription contractDescription, ServiceEndpoint endpoint, DispatchRuntime dispatchRuntime);
    void Validate(ContractDescription contractDescription, ServiceEndpoint endpoint);
}

public interface IOperationBehavior
{
    void AddBindingParameters(OperationDescription operationDescription, BindingParameterCollection bindingParameters);
    void ApplyClientBehavior(OperationDescription operationDescription, ClientOperation clientOperation);
    void ApplyDispatchBehavior(OperationDescription operationDescription, DispatchOperation dispatchOperation);
    void Validate(OperationDescription operationDescription);
}
```

上面的代码给出了基于终结点行为、契约行为和操作行为相应接口的定义，从中可以看到它们具有四个相同的方法。

- **Validate:** 验证相应的描述 (ServiceEndpoint、ContractDescription 和 OperationDescription) 是否符合要求。
- **AddBindingParameters:** 向绑定上下文中 (BindingContext) 添加相应的绑定参数，它们一般提供给自定义的绑定元素，并最终被相应的信道获取以控制对消息的操作。
- **ApplyDispatchBehavior:** 将扩展应用到服务端分发运行时；
- **ApplyClientBehavior:** 将扩展应用到客户端运行时。

由于服务行为仅提供针对服务端的扩展实现，因此基于服务行为的接口并没有定义 `ApplyClientBehavior` 方法，下面的代码片段提供了服务行为接口 `IServiceBehavior` 的定义。

```
public interface IServiceBehavior
{
    void AddBindingParameters(ServiceDescription serviceDescription, ServiceHostBase
        serviceHostBase, Collection<ServiceEndpoint> endpoints,
        BindingParameterCollection bindingParameters);
    void ApplyDispatchBehavior(ServiceDescription serviceDescription,
        ServiceHostBase serviceHostBase);
    void Validate(ServiceDescription serviceDescription, ServiceHostBase
        serviceHostBase);
}
```

9.3.2 行为方法的执行

因为 WCF 四种类型的行为都属于“服务描述”的范畴，所以 `ServiceDescription`、`ServiceEndpoint`、`ContractDescription` 和 `OperationDescription` 均具有一个 `Behaviors` 属性。如下面的代码片段所示，这些属性的类型都是 `KeyedByTypeCollection<T>`，而泛型类型分别为上述的四个行为接口。

```
public class ServiceDescription
{
    //其他成员
    public KeyedByTypeCollection<IServiceBehavior> Behaviors { get; }
}
public class ServiceEndpoint
{
    //其他成员
    public KeyedByTypeCollection<IEndpointBehavior> Behaviors { get; }
}
public class ContractDescription
{
    //其他成员
    public KeyedByTypeCollection<IContractBehavior> Behaviors { get; }
}
public class OperationDescription
{
    //其他成员
    public KeyedByTypeCollection<IOperationBehavior> Behaviors { get; }
}
```

要想理解这四种类型的行为是如何实现对 WCF 的扩展的，就必须了解定义在行为中的这些方法执行的时机。下面就来讨论一下这些行为方法在服务端和客户端究竟是在什么时候执行的。

在进行服务寄宿的时候，与寄宿服务相关的所有类型行为的 `Validate`，`AddBindingParameters` 和 `ApplyDispatchBehavior` 都是在 `ServiceHost` 开启的时候被执行的。而此时，表示服务描述的 `ServiceDescription` 对象已经在初始化 `ServiceHost` 的时候被成功创建。具体来说，这三个方法执行的先后顺序是先执行 `Validate` 方法，然后执行

AddBindingParameters 方法, 最后执行 ApplyDispatchBehavior 方法。而执行这三个方法的方式都是类似的。

- 通过 ServiceDescription 的 Behaviors 得到所有服务行为, 并执行每个服务行为的方法。
- 通过 ServiceDescription 的 Endpoints 属性得到服务具有的所有终结点, 针对每个表示终结点的 ServiceEndpoint 对象, 通过其 Behaviors 属性得到所有终结点行为, 并执行终结点行为的方法。
- 针对每一个表示终结点的 ServiceEndpoint 对象, 通过 Contract 属性得到表示服务契约描述的 ContractDescription 对象。通过其 Behaviors 得到所有的契约行为, 并调用每个契约行为的方法。
- 针对每一个表示服务契约描述的 ContractDescription 对象, 通过其 Operations 属性得到服务契约所有的操作。针对每个表示操作描述的 OperationDescription 对象, 通过其 Behaviors 属性得到所有的操作行为, 并调用每个操作行为的方法。

对于客户端来说, 三个行为 (不包括服务行为) 的三个方法会在创建的 ChannelFactory<TChannel>开启的时候执行。具体的执行顺序为先执行 Validate 方法, 然后执行 AddBindingParameters 方法, 最后执行 ApplyClientBehavior 方法, 其执行的方式和服务端完全一致。

9.3.3 实例演示: 通过扩展确保语言文化一致性 (S901)

为了让读者对如何利用相应的行为对 WCF 进行扩展有深刻的认识, 在这里提供一个简单的实例演示。本实例模拟的场景是这样的: 创建一个支持多语言的资源服务, 该服务旨在为调用者提供基于某种语言的文本型资源。我们不希望客户端在每次调用服务的时候都显式地指定具体的语言, 而是根据客户端服务调用线程表示语言文化的上下文来自动识别所需的语言。

要让资源服务具有识别语言文化的能力, 必须将客户端服务调用线程当前的语言文化信息 (具体来说就是 Thread 的两个属性: CurrentUICulture 和 CurrentCulture) 自动传递到服务端。具体的实现原理是这样的: 将客户端服务调用线程的 CurrentUICulture 和 CurrentCulture 的语言文化代码保存在出栈消息的 SOAP 报头中, 并为它们设置一个预定义的名称和命名空间。服务操作在服务端执行之前, 我们根据这个预定义 SOAP 报头名称和命名空间将这两个语言文化代码从入栈消息中获取出来, 创建相应的 CultureInfo 对象并作为服务操作执行线程的 CurrentUICulture 和 CurrentCulture。服务操作在执行的时候, 只需要根据当前线程的语言文化上下文提供相应资源就可以了。

本实例还是采用我们一贯使用的三层结构 (Service.Interface、Service 和 Client), 其中类库项目 Service.Interface 用于定义服务契约, 控制台程序 Service 用于定义服务类型和服务寄宿程序, 而控制台程序 Client 作为进行服务调用的客户端程序。由于我们定义的一些扩展组件和行为会同时应用于客户端和服务端, 因此将它们都定义在 Service.Interface 中。

步骤一、创建自定义 CallContextInitializer: CultureReceiver

所谓客户端当前语言文化信息的传递,无外乎是客户端将当前线程的 `CurrentUICulture` 和 `CurrentCulture` 放到出栈消息中,而服务端将其从入栈消息中取出,并对当前线程的 `CurrentUICulture` 和 `CurrentCulture` 进行相应的设置。下面先来实现服务端用于进行语言文化信息获取的组件,笔者将其命名为 `CultureReceiver`。

由于 `CultureReceiver` 在从入栈消息中获取表示客户端线程的 `CurrentUICulture` 和 `CurrentCulture` 信息时,需要预先知道相应报头的名称和命名空间(命名空间仅仅用于 SOAP 报头),因此我们将其定义成如下一个名称为 `CultureMessageHeaderInfo` 的类。属性 `CurrentCultureName`, `CurrentUICultureName` 和 `Namespace` 分别表示代码客户端线程 `CurrentCulture` 和 `CurrentUICulture` 的报头名称与命名空间。

```
namespace Artech.WcfServices.Service.Interface
{
    internal class CultureMessageHeaderInfo
    {
        public string Namespace { get; set; }
        public string CurrentCultureName { get; set; }
        public string CurrentUICultureName { get; set; }
    }
}
```

我们进行语言文化报头接收及对服务操作执行线程当前语言文化的设置的 `CultureReceiver` 组件,被定义成一个实现了接口 `ICallContextInitializer` 的调用上下文初始化器。我们恰好通过实现 `BeforeInvoke` 方法将存放在入栈消息报头的表示客户端线程 `CurrentCulture` 和 `CurrentUICulture` 的内容取出,并以此创建相应的 `CultureInfo` 作为当前线程的 `CurrentCulture` 和 `CurrentUICulture`。

由于 WCF 服务端采用线程池的机制处理客户端请求,线程会被重用,因此我们有必要在操作方法执行之后将当前线程的语言文化设置恢复到之前的状态,而这恰好可以实现在 `AfterInvoke` 方法中。下面的代码片段表示 `CultureReceiver` 的全部定义。

```
using System.Globalization;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Threading;
namespace Artech.WcfServices.Service.Interface
{
    internal class CultureReceiver : ICallContextInitializer
    {
        public CultureMessageHeaderInfo messageHeaderInfo;
        public CultureReceiver(CultureMessageHeaderInfo messageHeaderInfo)
        {
            this.messageHeaderInfo = messageHeaderInfo;
        }

        public void AfterInvoke(object correlationState)
```

```

    {
        CultureInfo[] cultureInfos = correlationState as CultureInfo[];
        if (null != cultureInfos)
        {
            Thread.CurrentThread.CurrentCulture = cultureInfos[0];
            Thread.CurrentThread.CurrentUICulture = cultureInfos[1];
        }
    }

    public object BeforeInvoke(InstanceContext instanceContext, IClientChannel
        channel, Message message)
    {
        CultureInfo[] originalCulture = new CultureInfo[]
        { CultureInfo.CurrentCulture, CultureInfo.CurrentUICulture };
        CultureInfo currentCulture = null;
        CultureInfo currentUICulture = null;
        if (message.Headers.FindHeader(this.messageHeaderInfo.Current
            CultureName,
            this.messageHeaderInfo.Namespace) > -1)
        {
            currentCulture = new CultureInfo(message.Headers.GetHeader
                <string>(
                    this.messageHeaderInfo.CurrentCultureName,
                    this.messageHeaderInfo.Namespace));
            Thread.CurrentThread.CurrentCulture = currentCulture;
        }
        if (message.Headers.FindHeader(this.messageHeaderInfo.
            CurrentUICultureName,
            this.messageHeaderInfo.Namespace) > -1)
        {
            currentUICulture = new
                CultureInfo(message.Headers.GetHeader<string>(
                    this.messageHeaderInfo.CurrentUICultureName,
                    this.messageHeaderInfo.Namespace));
            Thread.CurrentThread.CurrentUICulture = currentUICulture;
        }
        return originalCulture;
    }
}

```

步骤二、自定义 ClientMessageInspector: CultureSender

在客户端，我们通过自定义一个实现了 IClientMessageInspector 接口的客户端消息检验器，将代表当前线程 CurrentCulture 和 CurrentUICulture 的语言文化代码以 SOAP 报头的形式植入请求消息中。我们将该自定义的 ClientMessageInspector 称为 CultureSender。如下面的代码所示，上述的关于客户端线程当前语言文化信息的发送实现在 BeforeSendRequest 方法中。

```

using System.Globalization;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
namespace Artech.WcfServices.Service.Interface
{
    internal class CultureSender : IClientMessageInspector
    {

```

```

private CultureMessageHeaderInfo messageHeaderInfo;
public CultureSender(CultureMessageHeaderInfo messageHeaderInfo)
{
    this.messageHeaderInfo = messageHeaderInfo;
}
public void AfterReceiveReply(ref Message reply, object correlation
State) { }
public object BeforeSendRequest(ref Message request, IClientChannel
channel)
{
    request.Headers.Add(MessageHeader.CreateHeader(
        this.messageHeaderInfo.CurrentCultureName,
        this.messageHeaderInfo.Namespace, CultureInfo.CurrentCulture.Name));
    request.Headers.Add(MessageHeader.CreateHeader(
        this.messageHeaderInfo.CurrentUICultureName,
        this.messageHeaderInfo.Namespace, CultureInfo.
        CurrentUICulture.Name));
    return null;
}
}
}

```

步骤三、创建行为: CulturePropagationBehaviorAttribute

到目前为止,真正实现语言文化信息从客户端到服务端传播的自定义调用上下文初始化器 (CultureReceiver) 和客户端消息检验器 (CultureSender) 都已经创建好了。目前需要做的是通过定义相应的行为将这两个自定义组件分别应用到 WCF 的服务端和客户端运行时框架中去。具体来说,需要创建 CultureReceiver 对象并将其添加到相应分发操作的 CallContextInitializer 列表之中,创建 CultureSender 对象并将其添加到客户端运行时的 MessageInspector 列表之中。

以下定义的 CulturePropagationBehaviorAttribute 就是这样一个行为。从下面给出的代码中可以看到, CulturePropagationBehaviorAttribute 实现了三个行为接口 (IServiceBehavior, IEndpointBehavior, IContractBehavior), 所以它既是一个服务行为,同时也是一个终结点行为和契约行为。我们自定义的 CultureReceiver 和 CultureSender 分别通过 ApplyDispatchBehavior 和 ApplyClientBehavior 方法应用到服务端和客户端运行时框架。

```

using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
namespace Artech.WcfServices.Service.Interface
{
    public class CulturePropagationBehaviorAttribute : Attribute,
        IServiceBehavior, IEndpointBehavior, IContractBehavior
    {
        private CultureMessageHeaderInfo messageHeaderInfo;

        public const string DefaultNamespace =
            "http://www.artech.com/culturepropagation";
        public const string DefaultCurrentCultureName = "CurrentCultureName";
        public const string DefaultCurrentUICultureName = "CurrentUICultureName";
    }
}

```

```

public string Namespace { get; set; }
public string CurrentCultureName { get; set; }
public string CurrentUICultureName { get; set; }

public CulturePropagationBehaviorAttribute()
{
    messageHeaderInfo = new CultureMessageHeaderInfo
    {
        Namespace = DefaultNamespace,
        CurrentCultureName = DefaultCurrentCultureName,
        CurrentUICultureName = DefaultCurrentUICultureName
    };
}

//IServiceBehavior
public void AddBindingParameters(ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
    BindingParameterCollection bindingParameters) { }
public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase)
{
    foreach (ChannelDispatcher channelDispatcher in
        serviceHostBase.ChannelDispatchers)
    {
        foreach (EndpointDispatcher endpoint in channelDispatcher.
            Endpoints)
        {
            foreach (DispatchOperation operation in
                endpoint.DispatchRuntime.Operations)
            {
                operation.CallContextInitializers.Add(new
                    CultureReceiver(messageHeaderInfo));
            }
        }
    }
}

public void Validate(ServiceDescription serviceDescription, Service
HostBase
    serviceHostBase) {}

//IEndpointBehavior
public void AddBindingParameters(ServiceEndpoint endpoint,
    BindingParameterCollection bindingParameters) {}
public void ApplyClientBehavior(ServiceEndpoint endpoint, ClientRuntime
clientRuntime)
{
    clientRuntime.MessageInspectors.Add(new
        CultureSender(messageHeaderInfo));
}
public void ApplyDispatchBehavior(ServiceEndpoint endpoint,
    EndpointDispatcher endpointDispatcher)
{
    foreach (DispatchOperation operation in
        endpointDispatcher.DispatchRuntime.Operations)
    {
        operation.CallContextInitializers.Add(new
            CultureReceiver(messageHeaderInfo));
    }
}

```

```

    }
    public void Validate(ServiceEndpoint endpoint) {}

    // IContractBehavior
    public void AddBindingParameters(ContractDescription contractDescription,
        ServiceEndpoint endpoint, BindingParameterCollection
        bindingParameters) {}
    public void ApplyClientBehavior(ContractDescription contractDescription,
        ServiceEndpoint endpoint, ClientRuntime clientRuntime)
    {
        clientRuntime.MessageInspectors.Add(new
            CultureSender(messageHeaderInfo));
    }
    public void ApplyDispatchBehavior(ContractDescription contractDescription,
        ServiceEndpoint endpoint, DispatchRuntime dispatchRuntime)
    {
        foreach (DispatchOperation operation in dispatchRuntime.Operations)
        {
            operation.CallContextInitializers.Add(new
                CultureReceiver(messageHeaderInfo));
        }
    }
    public void Validate(ContractDescription contractDescription,
        ServiceEndpoint endpoint) { }
}
}

```

由于作为服务行为和契约行为的 `CulturePropagationBehaviorAttribute` 又是一个自定义特性，因此我们可以直接将其应用到服务契约接口（作为契约行为）或者服务类型（作为服务行为）上。在应用该行为特性时，可以设置 `CurrentCultureName`、`CurrentUICultureName` 和 `Namespace` 属性改变封装客户端线程 `CurrentCulture` 和 `CurrentUICulture` 的 SOAP 报头的名称和命名空间。如果这三个属性没有经过显式设置，它们具有默认值。

作为契约行为：

```

[ServiceContract]
[CulturePropagationBehavior(CurrentCultureName = "culture",
    CurrentUICultureName = "uiCulture")]
public interface IResourceService
{
    [OperationContract]
    string GetString(string key);
}

```

作为服务行为：

```

[CulturePropagationBehavior(CurrentCultureName = "culture",
    CurrentUICultureName = "uiCulture")]
public class ResourceService : IResourceService
{
    public string GetString(string key)
    {
        //省略实现
    }
}

```

步骤四、为 CulturePropagationBehaviorAttribute 定义配置元素

除了通过编程的方式应用行为之外，对于 WCF 四种类型的行为，契约行为和操作行为只能通过自定义特性的方式以声明的方式分别应用到服务契约接口（或者类）和操作契约方法或操作实现方法上。终结点行为只能通过配置的方式应用到对应的终结点。而服务行为则可以同时采用声明和配置的方式应用到目标服务上面。

我们定义的 CulturePropagationBehaviorAttribute 既是服务行为，也是终结点行为。为了实现以配置的方式来使用该行为，需要为之创建一个配置元素的类。作为服务行为或者终结点行为的配置元素类均继承自抽象类 System.ServiceModel.Configuration.BehaviorExtensionElement。

在这里我们创建了如下一个 CulturePropagationBehaviorElement 类来定义 Culture Propagation BehaviorAttribute 的配置元素。在 BehaviorExtensionElement 中，定义了三个可选（IsRequired = false）的配置属性 CurrentCultureName、CurrentUICultureName 和 Namespace 分别代表用于封装客户端线程 CurrentCulture 和 CurrentUICulture 的 SOAP 报头名称和命名空间。

```
using System;
using System.Configuration;
using System.ServiceModel.Configuration;
namespace Artech.WcfServices.Service.Interface
{
    public class CulturePropagationBehaviorElement : BehaviorExtensionElement
    {
        [ConfigurationProperty("namespace", IsRequired = false,
            DefaultValue = CulturePropagationBehaviorAttribute.DefaultNamespace)]
        public string Namespace
        {
            get { return (string)this["namespace"]; }
            set { this["namespace"] = value; }
        }
        [ConfigurationProperty("currentCultureName", IsRequired = false, DefaultValue
            = CulturePropagationBehaviorAttribute.DefaultCurrentCultureName)]
        public string CurrentCultureName
        {
            get { return (string)this["currentCultureName"]; }
            set { this["currentCultureName"] = value; }
        }
        [ConfigurationProperty("currentUICultureName", IsRequired = false,
            DefaultValue = CulturePropagationBehaviorAttribute.DefaultCurrentUICultureName)]
        public string CurrentUICultureName
        {
            get { return (string)this["currentUICultureName"]; }
            set { this["currentUICultureName"] = value; }
        }
        public override Type BehaviorType
        {
            get { return typeof(CulturePropagationBehaviorAttribute); }
        }
        protected override object CreateBehavior()
        {

```

```

        return new CulturePropagationBehaviorAttribute
        {
            Namespace = this.Namespace,
            CurrentCultureName = this.CurrentCultureName,
            CurrentUICultureName = this.CurrentUICultureName
        };
    }
}
}

```

下面的 XML 片段反映了如何将 `CulturePropagationBehaviorAttribute` 作为服务行为以配置的方式应用到目标服务上。我们需要将基于服务行为的配置元素类型以行为扩展的形式定义在 `<extensions>/<behaviorExtensions>` 结点下, 并给它一个名称 (在这里将我们定义的扩展取名为 `culturePropagation`)。在配置服务行为的时候, 只需要在行为配置节点中添加以行为扩展名为元素名的 XML 结点, 并对定义在配置元素类型中的配置属性进行相应的设置即可。

在本例中我们定义了一个名称为 `defaultSvcBehavior` 的服务行为。该行为节点的子节点 `<culturePropagation>` 代表我们自定义的 `CulturePropagationBehaviorAttribute` 行为。在 `<culturePropagation>` 节点下, 我们对 `namespace`、`currentCultureName` 和 `currentUICultureName` 做了显式设置 (由于三个配置属性是可选的, 因此如果没有对它们进行显式设置, 它们将会具有一个默认值)。最终这个名称为 `defaultSvcBehavior` 的服务行为被应用到了 `ResourceService` 服务上。

```

<configuration>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration="defaultSvcBehavior"
        ...
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="defaultSvcBehavior">
          <culturePropagation
            namespace="http://www.artech.com/"
            currentCultureName="cultureName"
            currentUICultureName="uiCultureName" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <extensions>
      <behaviorExtensions>
        <add name="culturePropagation"
          type="Artech.WcfServices.Service.Interface.CulturePropagationBehaviorElement,
            Artech.WcfServices.Service.Interface,
            Version=1.0.0.0,
            Culture=neutral,
            PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>
  </system.serviceModel>
</configuration>

```

如果需要将 `CulturePropagationBehaviorAttribute` 以终结点行为的方式应用到服务的某个

终结点上，配置方式与此类似。同样需要将代表行为配置元素的类型名称定义成行为扩展，然后将代表该行为配置的 XML 节点添加到终结点配置节点即可。在下面这段配置中，寄宿服务唯一的终结点被应用于一个名称为 `defaultEndpointBehavior` 的终结点行为。而该终结点行为配置中包含了 `CulturePropagationBehaviorAttribute` 的相关设置。

```
<configuration>
  <system.serviceModel>
    <services>
      <service...>
        <endpoint behaviorConfiguration="defaultEndpointBehavior".../>
      </service>
    </services>
    <behaviors>
      <endpointBehaviors>
        <behavior name="defaultEndpointBehavior">
          <culturePropagation
            namespace="http://www.artech.com/"
            currentCultureName="cultureName"
            currentUICultureName="uiCultureName"/>
        </behavior>
      </endpointBehaviors>
    </behaviors>
    <extensions>
      <behaviorExtensions>
        <add name="culturePropagation"
          type="Artech.WcfServices.Service.Interface.CulturePropagationBehaviorElement,
            Artech.WcfServices.Service.Interface,
            Version=1.0.0.0,
            Culture=neutral,
            PublicKeyToken=null" />
      </behaviorExtensions>
    </extensions>
  </system.serviceModel>
</configuration>
```

步骤五、创建实例应用检验语言文化的自动传播

到目前为止，关于语言文化从客户端自动传播到服务端的所有扩展实现均已完成。为了检验我们自定义的行为 `CulturePropagationBehaviorAttribute` 是否真的能够实现这个目标，需要通过建立一个简单的 WCF 应用程序来检验。

我们采用之前介绍过的文本型资源提供服务，为此创建了如下一个简单的代表服务契约接口 `IResourceService`。该服务契约具有一个唯一的操作契约方法 `GetString` 用于获取基于给定的键值得到的对应的文本型资源的内容。需要注意的是，我们定义的 `CulturePropagationBehaviorAttribute` 以契约行为的形式应用到 `IResourceService` 接口之上。

```
using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    [CulturePropagationBehavior]
    public interface IResourceService
```

```

{
    [OperationContract]
    string GetString(string key);
}

```

为了提供对多语言的支持,我们将资源文本的内容定义在资源文件中。为此在定义服务类型的项目 Service 中添加了如图 9-11 所示的两个资源文件。其中 Resources.resx 代表语言文化中性的资源文件,而 Resources.zh-CN.resx 则代表基于中国(大陆)简体中文的资源文件。两个资源文件定义了英文和中文作为内容的两个文本资源条目 HappyNewYear 和 MerryChristmas。

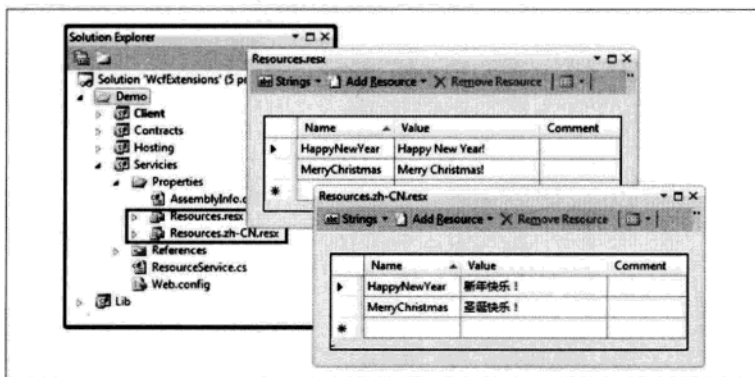


图 9-11 定义资源文本的两个资源文件

在默认的情况下,添加语言文化中性的资源文件会自动生成方便访问资源条目的代码。在服务类型的 GetString 方法中,直接使用定义在自动生成的 Resources 类的静态属性 ResourceManager (相应类型为 System.Resources.ResourceManager) 来获取给定键值的相应文本资源的内容。

```

using Artech.WcfServices.Service.Interface;
using Artech.WcfServices.Service.Properties;
namespace Artech.WcfServices.Service
{
    public class ResourceService : IResourceService
    {
        public string GetString(string key)
        {
            return Resources.ResourceManager.GetString(key);
        }
    }
}

```

然后,服务 ResourceService 以控制台应用作为宿主进行简单的自我寄宿。下面是服务端配置和客户端配置,由于我们自定义的 CulturePropagationBehaviorAttribute 是以声明的方式作为契约行为应用到契约接口上的,因此配置中并不包含相关的内容。

服务寄宿端配置:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.ResourceService">
        <endpoint address = "http://127.0.0.1:3721/resourceservice"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.
            IResourceService"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

客户端配置:

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name = "resourceservice"
        address = "http://127.0.0.1:3721/resourceservice"
        binding = "ws2007HttpBinding"
        contract = "Artech.WcfServices.Service.Interface.IResource
          Service"/>
    </client>
  </system.serviceModel>
</configuration>
```

下面是进行服务调用的代码。我们一共进行了四次针对 GetString 操作的服务调用,在调用之前对当前线程的 CurrentUICulture (它决定了语言的种类和对资源文件的选择)进行了设置。前面两次和后面两次是在 CurrentUICulture 为 en-US 和 zh-CN 的情况下进行调用的。从输出结果可以清晰地看到,客户端得到的资源文本的语言正好和当前线程的 CurrentUICulture 一致,而这正是应用在契约接口上的 CulturePropagationBehaviorAttribute 特性所致。

```
using (ChannelFactory<IResourceService> channelFactory = new
  ChannelFactory<IResourceService>("resourceservice"))
{
  IResourceService proxy = channelFactory.CreateChannel();
  Thread.CurrentThread.CurrentUICulture = new CultureInfo("en-US");
  Console.WriteLine(proxy.GetString("HappyNewYear"));
  Console.WriteLine(proxy.GetString("MerryChristmas") + "\n");

  Thread.CurrentThread.CurrentUICulture = new CultureInfo("zh-CN");
  Console.WriteLine(proxy.GetString("HappyNewYear"));
  Console.WriteLine(proxy.GetString("MerryChristmas"));
}
```

输出结果:

```
Happy New Year!
Merry Christmas!
```

```
新年快乐!
圣诞快乐!
```

9.4 ServiceHost 对 WCF 的扩展

除了采用自定义特性声明（服务行为、契约行为和操作行为）或配置的方式（服务行为和终结点行为）应用自定义的行为之外，还可以通过自定义 `ServiceHost` 来应用这些自定义的行为。自定义 `ServiceHost` 是对 WCF 的服务端进行扩展的一种常用的方式。

在创建 `ServiceHost` 的时候，WCF 会加载服务相关的配置，并将其作为服务的描述信息附加到 `ServiceHost` 对象上，也可以在开启 `ServiceHost` 之前对其服务描述信息进行相应的修改。`ServiceHost` 在开启之前具有的服务描述信息将会决定在开启之后创建的服务端运行时框架。所以如果我们通过自定义 `ServiceHost` 对象并根据具体应用场景的具体需求对其服务描述进行定制，同样可以起到对 WCF 服务端进行扩展的目的。

9.4.1 自定义 ServiceHost 的本质：对服务描述进行定制

通过前面对 WCF 服务端运行时框架的介绍，我们知道了在初始化 `ServiceHost` 时创建的服务描述是构建服务端运行时框架的基础。服务描述通过类型 `ServiceDescription` 表示，被创建的服务描述可以通过 `ServiceHost` 的只读属性 `Description` 得到。下面的代码片段表示该属性在 `ServiceHost` 的基类 `ServiceHostBase` 中的定义。

```
public abstract class ServiceHostBase : CommunicationObject,
    IExtensibleObject<ServiceHostBase>, IDisposable
{
    //其他成员
    public ServiceDescription Description { get; }
}
```

在服务的众多描述信息中，以前面介绍的四种行为表示的行为信息是最为重要的组成部分。这里的行为信息最终决定了 WCF 服务端框架进行消息分发、实例激活、操作执行、异常处理、元数据发布、事务管理、并发控制、流量限制、传输安全、存取控制等方面的行为。我们通过自定义 `ServiceHost` 实现对 WCF 的扩展，其本质在于对服务的行为描述进行相应的定制。

以上面关于实现语言文化信息自动传播的扩展为例，代表客户端线程 `CurrentUICulture` 和 `CurrentCulture` 的语言文化代码在客户端的发送和服务端的接收与对当前线程语言文化上下文的设置，都是通过自定义行为 `CulturePropagationBehaviorAttribute` 实现的。而该 `CulturePropagationBehaviorAttribute` 特性最终作为契约行为被应用到了契约接口上。如果没有这个特性，对于服务端来说我们也可以通过自定义 `ServiceHost` 的方式直接将 `CulturePropagationBehaviorAttribute` 行为添加到服务描述信息中。

通过自定义 `ServiceHost` 实现对服务描述的定义很简单。只需要重写 `ServiceHost` 的虚方

法 `OnOpening`，并对 `Description` 属性进行相应的修改即可。在下面的代码片段中，我们创建了一个继承自 `ServiceHost` 的 `CulturePropagationServiceHost` 类型，并在重写的 `OnOpening` 方法中将创建的 `CulturePropagationBehaviorAttribute` 对象作为服务行为添加到服务行为列表中。此外还定义了相应的构造函数。（S902）

```
public class CulturePropagationServiceHost: ServiceHost
{
    public CulturePropagationServiceHost(Type serviceType, params Uri[]
        baseAddresses)
        : base(serviceType, baseAddresses)
    { }

    protected override void OnOpening()
    {
        base.OnOpening();
        CulturePropagationBehaviorAttribute behavior =
            this.Description.Behaviors.Find<CulturePropagationBehavior
                Attribute>();
        if(null == behavior)
        {
            this.Description.Behaviors.Add(new
                CulturePropagationBehaviorAttribute());
        }
    }
}
```

在进行自我寄宿的情况下，就可以直接创建 `CulturePropagationServiceHost` 来寄宿相应的服务了。无须再进行基于 `CulturePropagationBehaviorAttribute` 行为的设置，被寄宿的服务就具有“语言文化识别”的能力。

```
using (CulturePropagationServiceHost host = new
    CulturePropagationServiceHost(typeof(ResourceService)))
{
    host.Open();
    Console.Read();
}
```

我们说的基于自定义 `ServiceHost` 的扩展，实际上只需要让我们定义的类继承自 `ServiceHostBase` 即可。但是在绝大部分情况下，可以直接使用定义在 `ServiceHost` 类型中的功能，所以我们一般会通过继承自 `ServiceHost` 来定义我们自己的 `ServiceHost`。

1. ServiceHost 开启后对 Description 的定制无效

基于服务描述的服务端运行时框架式是在 `ServiceHost` 开启过程中被构建出来的，这就意味着只有在 `ServiceHost` 开启之前对服务描述的定义才是有效的。这也是我们需要将对服务描述的定制操作定义在重写的 `OnOpening` 方法中的原因。

比如在下方的代码片段中，对 `CulturePropagationServiceHost` 进行了重新定义，将原本定义在 `OnOpening` 方法中应用 `CulturePropagationBehaviorAttribute` 行为的代码转移到了重写的 `OnOpened` 方法中。其实这样的定义是无意义的，根本起不到任何作用。

```

public class CulturePropagationServiceHost : ServiceHost
{
    //其他成员
    protected override void OnOpened()
    {
        base.OnOpened();
        CulturePropagationBehaviorAttribute behavior =
            this.Description.Behaviors.Find<CulturePropagationBehavior
                Attribute>();
        if (null == behavior)
        {
            this.Description.Behaviors.Add(new
                CulturePropagationBehaviorAttribute());
        }
    }
}

```

2. 通过自定义 ServiceHost 对分发运行时进行定制是无效的

CulturePropagationBehaviorAttribute 针对服务端的意义在于将 CultureReceiver 对象添加到基于终结点的分发运行时 (DispatchRuntime) 所有操作 (DispatchOperation) 的 CallContextInitializer 列表中。而 CultureReceiver 的目的在于从请求消息中获取客户端语言文化上下文, 并对当前线程的语言文化上下文进行相应的设置。有人也许会问这么一个问题: 如果我们在自定义 CulturePropagationServiceHost 的时候, 绕开对服务描述的设置, 直接对分发运行时进行定制, 是否可以起到一样的作用? 照理说, 我们通过下面的方式来重新定义 CulturePropagationServiceHost 也是等效的。

```

public class CulturePropagationServiceHost : ServiceHost
{
    //其他成员
    protected override void OnOpened()
    {
        base.OnOpened();
        foreach (ChannelDispatcher channelDispatcher in this.ChannelDispatchers)
        {
            foreach (EndpointDispatcher endpointDispatcher in
                channelDispatcher.Endpoints)
            {
                foreach (DispatchOperation operation in
                    endpointDispatcher.DispatchRuntime.Operations)
                {
                    operation.CallContextInitializers.Add(new CultureReceiver(new
                        CultureMessageHeaderInfo()));
                }
            }
        }
    }
}

```

但是, 这种在 ServiceHost 开启之后对分发运行时进行的更改是不合法的。如果你使用上面定义的 CulturePropagationServiceHost 进行服务寄宿, 当程序执行到为 DispatchOperation 添加 CallContextInitializer 的地方, 会抛出如图 9-12 所示的 InvalidOperationException 异常,

并且提示“打开 ServiceHost 后，不能更改此值”。

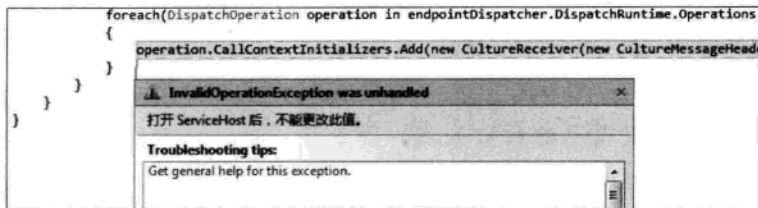


图 9-12 ServiceHost 开启之后更改 DispatchRuntime 导致的异常

在 ServiceHost 被开启的情况下，试图对创建的分发运行时进行改变，都会抛出如图 9-12 所示的异常。其背后的原因在于：在开启 ServiceHost 的过程中，针对终结点创建相应的分发运行时并对其进行初始化之后，会将其标记为只读。

WCF 内部的做法是：基于初始化的 DispatchRuntime 对象创建一个 ImmutableDispatchRuntime 对象，原来的 DispatchRuntime 将不会被使用。ImmutableDispatchRuntime 是一个定义在 System.ServiceModel.Dispatcher 命名空间下的内部类型。从名称上就可以看得出来，ImmutableDispatchRuntime 对象是一个恒定不变的运行时。既然原来的 DispatchRuntime 对象在 ImmutableDispatchRuntime 创建之后就不会被使用，我们针对它的任何修改已经变得没有意义，所以在设计 DispatchRuntime 相关 API 的时候，针对它的属性的修改都会加上 ServiceHost 是否被开启的检验。相同的设计同样应用在 ClientRuntime 上。

9.4.2 自定义 ServiceHost 的创建者：ServiceHostFactory

对于我们自定义的 ServiceHost，可以在自我寄宿的时候直接使用。如果采用 IIS 或者 WAS 寄宿方式，需要为寄宿的服务创建一个.svc 文件（在 WCF 4.0 中这个文件可以借助于相应的配置省掉）。如果读者阅读了上册的第 7 章“服务寄宿（Hosting）”，应该知道自定义 ServiceHost 是通过自定义的 ServiceHostFactory 来创建的。

自定义 ServiceHostFactory 需要继承抽象类 System.ServiceModel.Activation.ServiceHostFactoryBase。下面的代码片段给出了 ServiceHostFactoryBase 的定义，而通过调用 CreateServiceHost 方法得到的类型为 ServiceHostBase 对象用于进行服务的寄宿工作。

```
public abstract class ServiceHostFactoryBase
{
    protected ServiceHostFactoryBase();
    public abstract ServiceHostBase CreateServiceHost(string
        constructorString, Uri[]
        baseAddresses);
}
```

自定义 ServiceHostFactory 的类型通过定义在.svc 文件中的“%@ServiceHost%”指令（Directive）的 Factory 属性来表示。

```
<%@ ServiceHost Service="Artech.WcfServices.Service.ResourceService" Factory
="Artech.WcfServices.Service.CulturePropagationServiceHostFactory"%>
```

除此之外,从上面的代码片段中可以看到,CreateServiceHost 方法中需要传入一个字符串类型的参数 constructorString。从字面上的意思我们知道它代表创建 ServiceHost 时调用相应构造函数以字符串形式表示的参数列表。而这个 constructorString 参数来源于“%@ServiceHost%”指令的 Service 属性值。

也就是说,“%@ServiceHost%”指令的 Service 属性严格来说并不是指寄宿服务的有效类型,而是传递给对应 ServiceHostFactory 的 CreateServiceHost 方法的第一个参数值。之所以在正常的情况下只需要指定寄宿服务的有效类型就可以了,原因在于默认使用的 ServiceHost 为 System.ServiceModel.Activation.ServiceHostFactory,在它通过 CreateServiceHost 方法进行 ServiceHost 的创建时,只需要知道寄宿服务的类型就可以了。

注:ServiceHost 既可以泛指用于进行服务寄宿的继承自 ServiceHostBase 的对象或者类型,又可以具体指 System.ServiceModel.ServiceHost 类型。同理,ServiceHostFactory 既可以泛指继承于 ServiceHostFactoryBase 的对象或者类型,也可以具体指 System.ServiceModel.Activation.ServiceHostFactory 类型。读者应该根据上下文判断这两个词所指为何。

虽然说自定义 ServiceHostFactory 只需要继承 ServiceHostFactoryBase 即可,但是在绝大多数情况下我们会通过继承 System.ServiceModel.Activation.ServiceHostFactory 来创建自定义的 ServiceHostFactory,因为需要借助它提供的自动程序集加载机制。

不知道读者有没有注意这样一个问题:对于“%@ServiceHost%”指令的 Service 属性值,我们仅需要指定寄宿服务的全名(命名空间+类型名称)就可以了,而无须指定具体的程序集名称。如果定义服务类型的程序集没有被加载,服务类型照理说是不能被正确解析的。实际上在 System.ServiceModel.Activation.ServiceHostFactory 调用 CreateServiceHost 方法的时候,如果指定的服务类型不能被解析,它会加载所有被引用的程序集。System.ServiceModel.Activation.ServiceHostFactory 定义如下。

```
public class ServiceHostFactory : ServiceHostFactoryBase
{
    public ServiceHostFactory();
    public override ServiceHostBase CreateServiceHost(string constructorString,
        Uri[] baseAddresses);
    protected virtual ServiceHost CreateServiceHost(Type serviceType, Uri[]
        baseAddresses);
}
```

我们通过继承 System.ServiceModel.Activation.ServiceHostFactory 为 CulturePropagation ServiceHost 创建 ServiceHostFactory。如下面的代码所示,只需要在重写的虚方法 CreateServiceHost 中创建自定义的 ServiceHost 即可。


```

public class CulturePropagationServiceHostFactory : ServiceHostFactory
{
    protected override ServiceHost CreateServiceHost(Type serviceType, Uri[]
        baseAddresses)
    {
        return new CulturePropagationServiceHost(serviceType, baseAddresses);
    }
}

```

9.4.3 实例演示：通过扩展实现基于 IoC 的服务实例的创建 (S903, S904)

所谓控制反转 (IoC, Inversion Of Control) 指应用本身不负责依赖对象的创建和维护, 而交给一个外部容器来负责。这样控制权就由应用转移到了外部 IoC 容器, 控制权就实现了所谓的反转。比如在类型 A 中需要使用类型 B 的实例, 而 B 实例的创建并不由 A 来负责, 而是通过外部容器来创建。

有时我们又将 IoC 称为依赖注入 (DI, Dependency Injection)。所谓依赖注入, 就是由外部容器在运行时动态地将依赖的对象注入组件之中。具体的依赖注入方式又包括如下三种典型的形式。

- 构造器注入 (Constructor Injection): IoC 容器会智能地选择和调用适合的构造函数以创建依赖的对象。如果被选择的构造函数具有相应的参数, IoC 容器在调用构造函数之前会自定义创建相应参数对象。
- 属性注入 (Property Injection): 如果需要使用被依赖对象的某个属性, 在被依赖对象被创建之后, IoC 容器会自动初始化该属性。
- 方法注入 (Method Injection): 如果被依赖对象需要调用某个方法进行相应的初始化, 在该对象创建之后, IoC 容器会自动调用该方法。

在开源社区, 具有很有流行的 IoC 框架, 比如 Castle Windsor、Unity、Spring.NET、StructureMap、Ninject 等。现在就以 Unity 为例, 介绍通过 WCF 的扩展如何实现基于 IoC 的服务实例的创建。

Unity 简介

Unity 是微软 Patterns& Practices 部门开发的一个轻量级的 IoC 框架。该项目在 Codeplex 上的地址为 <http://unity.codeplex.com/>, 可以下载相应的安装包和开发文档。在本书出版之时, Unity 的最新版本为 2.1。限于篇幅, 不可能对 Unity 进行全面的介绍, 但是为了让读者了解 IoC 在 Unity 中的实现, 笔者写了一个简单的程序 (S903)。

首先创建一个控制台程序，定义如下几个接口 (IA、IB、IC 和 ID) 和它们各自的实现类 (A、B、C、D)。在类型 A 中定义了三个属性 B、C 和 D，其类型分别为接口 IB、IC 和 ID。其中属性 B 在构造函数中被初始化；属性 C 上应用了 Microsoft.Practices.Unity.DependencyAttribute 特性，意味着这是一个需要以属性注入方式被初始化的依赖属性；属性 D 则通过方法 Initialize 初始化，该方法上应用了 Microsoft.Practices.Unity.InjectionMethodAttribute，意味着这是一个注入方法，会被自动调用。

```
namespace UnityDemo
{
    public interface IA { }
    public interface IB { }
    public interface IC { }
    public interface ID { }

    public class A : IA
    {
        public IB B { get; set; }
        [Dependency]
        public IC C { get; set; }
        public ID D { get; set; }

        public A(IB b)
        {
            this.B = b;
        }
        [InjectionMethod]
        public void Initialize(ID d)
        {
            this.D = d;
        }
    }
    public class B : IB{}
    public class C : IC{}
    public class D : ID{}
}
```

然后我们为应用添加一个配置文件，并定义如下一段关于 Unity 的配置。这段配置定义了一个名称为 defaultContainer 的 Unity 容器，并在其中完成了上面定义的接口和对应实现类之间映射的类型匹配。

```
<configuration>
  <configSections>
    <section name="unity"
      type="Microsoft.Practices.Unity.Configuration.UnityConfiguration
        Section,
        Microsoft.Practices.Unity.Configuration"/>
  </configSections>
  <unity>
    <containers>
      <container name="defaultContainer">
        <register type="UnityDemo.IA, UnityDemo" mapTo="UnityDemo.A, UnityDemo"/>
        <register type="UnityDemo.IB, UnityDemo" mapTo="UnityDemo.B, UnityDemo"/>
        <register type="UnityDemo.IC, UnityDemo" mapTo="UnityDemo.C, UnityDemo"/>
      </container>
    </containers>
  </unity>
</configuration>
```

```

    <register type="UnityDemo.ID, UnityDemo" mapTo="UnityDemo.D, UnityDemo"/>
  </container>
</containers>
</unity>
</configuration>

```

最后在 Main 方法中创建一个代表 IoC 容器的 UnityContainer 对象，并加载配置信息对其进行初始化。然后调用它的泛型的 Resolve 方法创建一个实现了泛型接口 IA 的对象。最后将返回对象转变成类型 A，并检验其 B、C 和 D 属性是否为空。

```

static void Main(string[] args)
{
    IUnityContainer container = new UnityContainer();
    UnityConfigurationSection configuration =
        ConfigurationManager.GetSection(UnityConfigurationSection.SectionName)
        as UnityConfigurationSection;
    configuration.Configure(container, "defaultContainer");
    A a = container.Resolve<IA>() as A;
    if (null != a)
    {
        Console.WriteLine("a.B == null ? {0}", a.B == null ? "Yes" : "No");
        Console.WriteLine("a.C == null ? {0}", a.C == null ? "Yes" : "No");
        Console.WriteLine("a.D == null ? {0}", a.D == null ? "Yes" : "No");
    }
}

```

从如下给出的执行结果可以得到这样的结论：通过 Resolve<IA>方法返回的是一个类型为 A 的对象，该对象的三个属性被进行了有效的初始化。这个简单的程序分别体现了接口注入（通过相应的接口根据配置解析出相应的实现类型）、构造器注入（属性 B）、属性注入（属性 C）和方法注入（属性 D）。

```

a.B == null ? No
a.C == null ? No
a.D == null ? No

```

接下来演示的实例依然采用我们熟悉的三层结构（Service.Interface、Service 和 Client）。由于我们采用了 IIS/WAS 寄宿方式，所以需要将 Service 项目类型转变成类库项目。由于本实例只涉及针对服务的扩展，所以将自定义组件和相应的行为都定义在 Service 项目中。在这之前，需要添加基于 Unity 的两个程序集引用。可以从 Unity 官方网站上下载 Unity 安装包，也可以直接下载本实例的源代码（其中包含这两个程序集）。

- Microsoft.Practices.Unity.dll
- Microsoft.Practices.Unity.Configuration.dll

步骤一、自定义 InstanceProvider: UnityInstanceProvider

要实现 WCF 和 Unity 之间的集成，最终体现在如何通过 Unity 容器来创建服务实例。从前面介绍的关于服务端运行时框架我们知道最终服务实例的提供落在了一个实例提供者（InstanceProvider）之上。所以本实例的核心就是要自定义一个采用 Unity 实现服务实例提

供机制的自定义实例提供者。我们将之命名为 `UnityInstanceProvider`。

下面的代码给出了实现 `IInstanceProvider` 接口的 `UnityInstanceProvider` 的定义。在构造函数中指定了两个参数，即实现了 `IUnityContainer` 接口的 `Unity` 容器对象和服务契约类型。在真正实现对服务实例创建的 `GetInstance` 方法上，直接调用 `IUnityContainer` 的 `Resolve` 方法传入给定的服务契约类型来创建具体的服务实例。而在 `ReleaseInstance` 方法中则直接调用 `IUnityContainer` 的 `Teardown` 方法进行服务实例的释放。

```
using System;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using Microsoft.Practices.Unity;
namespace Artech.WcfServices.Service
{
    public class UnityInstanceProvider : IInstanceProvider
    {
        public IUnityContainer UnityContainer { get; private set; }
        public Type ContractType { get; private set; }

        public UnityInstanceProvider(IUnityContainer unityContainer, Type
            contractType)
        {
            this.UnityContainer = unityContainer;
            this.ContractType = contractType;
        }

        public object GetInstance(InstanceContext instanceContext, Message
            message)
        {
            return this.UnityContainer.Resolve(this.ContractType);
        }

        public object GetInstance(InstanceContext instanceContext)
        {
            return this.GetInstance(instanceContext, null);
        }

        public void ReleaseInstance(InstanceContext instanceContext, object
            instance)
        {
            this.UnityContainer.Teardown(instance);
        }
    }
}
```

步骤二、创建服务行为: `UnityServiceBehaviorAttribute`

和前面一个关于客户端语言文化上下文自动传播的实例一样，我们自定义的组件最终通过相应的行为应用到 WCF 的运行时中。为此，针对上面自定义的实例提供者定义了一个实现 `IServiceBehavior` 接口的服务行为 `UnityServiceBehaviorAttribute`。为了能让该服务行为以声明的方式直接应用到服务类型上，我们将其定义成特性。

UnityServiceBehaviorAttribute 的所有定义体现在如下所示的代码片段中。构造函数中具有一个字符串类型的参数 containerName 表示配置的 Unity 容器的名称。真正的容器名称在构造函数中被获取，为了避免 UnityContainer 的频繁创建，我们定义了一个静态的以容器名称为键值的字典，保存已经被创建的 Unity 容器。

我们的最终目的是根据给定名称的 Unity 容器创建上面定义的 UnityInstanceProvider，然后将其作为基于终结点的分发运行时的实例提供者，这样的功能定义在 ApplyDispatchBehavior 方法中。创建 UnityInstanceProvider 还需要服务契约的类型，而得到服务契约类型采用了这样的逻辑：首先根据当前 EndpointDispatcher 得到契约名称和命名空间，然后通过 ServiceHostBase 得到表示服务描述的 ServiceDescription 对象，最终根据前面得到的契约名称和命名空间找到对应的表示契约描述的 ContractDescription 对象，而该对象的 ContractType 属性表示服务契约的类型。

如果基于契约类型的注册不存在，ApplyDispatchBehavior 方法还进行了服务契约类型和服务类型之间的类型注册。

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Configuration;
using System.Linq;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;
using Microsoft.Practices.Unity;
using Microsoft.Practices.Unity.Configuration;

namespace Artech.WcfServices.Service
{
    public class UnityServiceBehaviorAttribute : Attribute, IServiceBehavior
    {
        static Dictionary<string, IUnityContainer> containers = new
            Dictionary<string, IUnityContainer>();
        public IUnityContainer UnityContainer { get; private set; }
        public UnityServiceBehaviorAttribute()
            : this(string.Empty)
        { }
        public UnityServiceBehaviorAttribute(string containerName)
        {
            containerName = containerName ?? string.Empty;
            if (containers.ContainsKey(containerName))
            {
                this.UnityContainer = containers[containerName];
            }
            else
            {
                lock (typeof(UnityServiceBehaviorAttribute))
                {
                    IUnityContainer container = new UnityContainer();
                    UnityConfigurationSection configuration =
                        (UnityConfigurationSection)ConfigurationManager.GetSection(
```

```

        UnityConfigurationSection.SectionName);
    if (containerName == string.Empty)
    {
        configuration.Configure(container);
    }
    else
    {
        configuration.Configure(container, containerName);
    }
    containers[containerName] = container;
    this.UnityContainer = container;
}
}
}

public void AddBindingParameters(ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints,
    BindingParameterCollection bindingParameters) {}

public void ApplyDispatchBehavior(ServiceDescription serviceDescription,
    ServiceHostBase serviceHostBase)
{
    foreach (ChannelDispatcher channelDispatcher in
        serviceHostBase.ChannelDispatchers)
    {
        foreach (EndpointDispatcher endpointDispatcher in
            channelDispatcher.Endpoints)
        {
            Type contractType = (from endpoint in
                serviceHostBase.Description.Endpoints
                where endpoint.Contract.Name ==
                    endpointDispatcher.ContractName
                    && endpoint.Contract.Namespace
                        == endpointDispatcher.ContractNamespace
                select
                    endpoint.Contract.ContractType).FirstOrDefault();
            if (null == contractType)
            {
                continue;
            }
            if (!this.UnityContainer.Registrations.Any(registration =>
                registration.RegisteredType == contractType))
            {
                this.UnityContainer.RegisterType(contractType,
                    serviceHostBase.Description.ServiceType);
            }
            endpointDispatcher.DispatchRuntime.InstanceProvider = new
                UnityInstanceProvider(this.UnityContainer, contractType);
        }
    }
}

public void Validate(ServiceDescription serviceDescription, ServiceHostBase
    serviceHostBase) {}
}
}

```

步骤三、自定义 ServiceHost: UnityServiceHost

接下来需要自定义一个 ServiceHost 来应用上面定义的服务行为 UnityServiceBehavior

Attribute。我们将这个自定义的 ServiceHost 命名为 UnityServiceHost。下面的代码片段给出了整个 UnityServiceHost 的定义。

```
using System;
using System.ServiceModel;
namespace Artech.WcfServices.Service
{
    public class UnityServiceHost : ServiceHost
    {
        private string containerName;
        public UnityServiceHost(Type serviceType, string containerName, params Uri[]
            baseAddresses)
            : base(serviceType, baseAddresses)
        {
            this.containerName = containerName;
        }
        protected override void OnOpening()
        {
            base.OnOpening();
            if (this.Description.Behaviors.Find<UnityServiceBehaviorAttribute>() ==
                null)
            {
                this.Description.Behaviors.Add(new
                    UnityServiceBehaviorAttribute(containerName));
            }
        }
    }
}
```

UnityServiceHost 直接继承自 System.ServiceModel.ServiceHost。构造函数中除了指定服务类型和可选的基地址数组之外，还指定了 Unity 容器的配置名称。在重写的 OnOpening 方法中，服务行为 UnityServiceBehaviorAttribute 被创建出来并被添加到了服务行为列表之中。

步骤四、自定义 ServiceHostFactory: UnityServiceHostFactory

自定义的 ServiceHost 在进行 IIS 或者 WAS 寄宿的时候需要通过相应的 ServiceHostFactory 来创建，我们需要为自定义的 UnityServiceHost 创建对应的 UnityServiceHostFactory。

UnityServiceHost 具有三个参数，即服务类型、Unity 容器名称和基地址数组。通过上面的介绍我们知道最终作为构造函数参数的来源是.svc 文件 %@ServiceHost% 指令的 Service 属性。为了使该属性能够同时包含用于创建自定义 UnityServiceHost 的必需服务类型和 Unity 容器名称，我们希望该属性具有如下的格式，即前半部分代表服务类型，后半部分代表 Unity 容器名称，中间采用分隔符“:”将两者隔开。

```
<%@ ServiceHost Service="{服务类型}: {Unity 容器名称}" ...%>
```

基于这样的 %@ServiceHost% 指令的 Service 属性格式，我们定义了如下的 UnityServiceHostFactory 类型。UnityServiceHostFactory 直接继承自 System.ServiceModel.Activation.ServiceHostFactory，重写的公有 CreateServiceHost 方法中，将 Unity 容器名称从

`constructorString` 参数中提取出来,并传入只包含服务类型名称的字符串作为参数调用基类的 `CreateServiceHost` 方法。而在重写的受保护 `CreateServiceHost` 方法中,则根据之前提取出来的 `Unity` 容器名称创建 `UnityServiceHost` 对象。

```
using System;
using System.ServiceModel;
using System.ServiceModel.Activation;
namespace Artech.WcfServices.Service
{
    public class UnityServiceHostFactory : ServiceHostFactory
    {
        public string ContianerName { get; private set; }
        protected override ServiceHost CreateServiceHost (Type serviceType, Uri[]
            baseAddresses)
        {
            return new UnityServiceHost (serviceType, this.ContianerName,
                baseAddresses);
        }
        public override ServiceHostBase CreateServiceHost (string constructorString,
            Uri[] baseAddresses)
        {
            var split = constructorString.Split(':');
            constructorString = split[0];
            if (split.Length > 1)
            {
                this.ContianerName = split[1].Trim();
            }
            return base.CreateServiceHost (constructorString, baseAddresses);
        }
    }
}
```

步骤五、创建实例程序应用自定义 `ServiceHost`

最后我们创建一个实例程序来演示如何以 `IIS` 寄宿方式使用上面我们自定义的 `ServiceHost`。依然沿用之前演示的资源服务的例子。

在上面演示的例子中,直接通过访问资源文件 (`.resx`) 的方式提供服务的实现。现在从可扩展性的角度对服务进行重新设计以实现对不同资源存储方法的支持。也就是说,可以将资源信息定义在资源文件中,也可以定义在数据库中,或者说访问另一个服务来提供所需要的资源。

为了让我们的资源服务具有这样的可扩展性,将基于不同资源存储方法的功能定义在一个接口中,并将其命名为 `IResourceProvider`。如下面的代码所示, `IResourceProvider` 接口仅具有一个 `GetString` 方法。

```
namespace Artech.WcfServices.Service
{
    public interface IResourceProvider
    {
        string GetString (string key);
    }
}
```


然后将基于.resx 文件的资源提供的功能定义成一个实现了 IResourceProvider 接口的 ResxFileProvider。直接将定义在 ResourceService 的 GetString 方法中的代码移到了 ResxFileProvider 的 GetString 方法中。

```
using Artech.WcfServices.Service.Properties;
namespace Artech.WcfServices.Service
{
    public class ResxFileProvider : IResourceProvider
    {
        public string GetString(string key)
        {
            return Resources.ResourceManager.GetString(key);
        }
    }
}
```

接下来为了创建与具体资源存储无关的资源服务，需要让 ResourceService 仅依赖我们定义的 IResourceProvider，下面是 ResourceService 的定义。

```
public class ResourceService : IResourceService
{
    public IResourceProvider Provider { get; private set; }

    public ResourceService(IResourceProvider provider)
    {
        this.Provider = provider;
    }

    public string GetString(string key)
    {
        return this.Provider.GetString(key);
    }
}
```

为了演示 IIS 服务寄宿方式，我们在 IIS 管理器中创建一个 Web 应用（比如将其命名为 WcfServices），并将其物理路径映射成定义 ResourceService 项目的根目录。同时更改项目的编译输出目录，从默认的\bin\debug 切换成\bin（因为 Web 应用自动加载的是\bin 目录下的程序集）。然后为该项目添加一个 Web.config，并进行如下的配置。通过这个配置文件，我们定义了一个名称为 defaultContainer 的 Unity 容器，并在该容器中定义了从 IResourceProvider 接口到 ResxFileProvider 类型的类型注册。

```
<configuration>
  <configSections>
    <section name="unity"
      type="Microsoft.Practices.Unity.Configuration.UnityConfiguration
        Section, Microsoft.Practices.Unity.Configuration"/>
  </configSections>
  <unity>
    <containers>
      <container name="defaultContainer">
        <register type="Artech.WcfServices.Service.IResourceProvider,
          Artech.WcfServices.Service"
          mapTo="Artech.WcfServices.Service.ResxFileProvider,
            Artech.WcfServices.Service">
```

```

        </register>
    </container>
</containers>
</unity>
<system.serviceModel>
    <services>
        <service name="Artech.WcfServices.Service.ResourceService">
            <endpoint binding="ws2007HttpBinding"
                contract="Artech.WcfServices.Service.Interface.IResource
                Service"/>
        </service>
    </services>
</system.serviceModel>
</configuration>

```

接下来需要在项目的根目录中添加一个文件名为 `ResourceService.svc` 的文件，其 `<%@ServiceHost%>` 指令定义如下。`Service` 属性以 “:” 作为分隔符将服务类型和 `Unity` 容器名称分开，而 `Factory` 属性指定的正是用于创建自定义 `UnityServiceHost` 的 `UnityServiceHostFactory` 的类型。

```

<%@ ServiceHost Service="Artech.WcfServices.Service.ResourceService:
defaultContainer"
Factory="Artech.WcfServices.Service.UnityServiceHostFactory,
Artech.WcfServices.Service"%>

```

现在只需要将客户端配置中的终结点地址替换成 `ResourceService.svc` 的地址 (`http://localhost/wcfservices/ResourceService.svc`)，就可以直接运行客户端程序并得到与之前一样的输出结果。

```

<configuration>
    <system.serviceModel>
        <client>
            <endpoint name ="resourceservice"
                address = "http://localhost/wcfservices/ResourceService.svc"
                binding = "ws2007HttpBinding"
                contract ="Artech.WcfServices.Service.Interface.IResource
                Service"/>
        </client>
    </system.serviceModel>
</configuration>

```

输出结果:

```

Happy New Year!
Merry Christmas!

```

```

新年快乐!
圣诞快乐!

```

第 10 章 WCF 4.0 新特性

(New Features in WCF 4.0)

作为 .NET Framework 的一部分, 几乎每个版本的 .NET Framework 的推出都会为 WCF 带来一些改变。针对最新版本的 .NET Framework 4.0, 一些新的特性被引入到 WCF。对于这些基于 .NET Framework 版本的更替而带来的针对 WCF 的变化, 笔者个人是这么看待的: WCF 在随着 .NET Framework 3.0 发布的时候就具有一个成熟的架构设计, 可扩展性即是一个重要的衡量标准。基于后续版本的 .NET Framework 发布的 WCF 并没有像 WF 一样出现“革新性”的改变, 很多都是利用了这个可扩展性的通信平台开发出来的新特性, WCF 4.0 也不例外。

对于 WCF 4.0 的众多新特性，我们可以按照功能将它们划分为如下几类。

- 简化开发体验：比如支持默认的终结点，让我们可以进行无配置服务部署。提供了标准终结点，让我们无须显式地控制终结点的 ABC 三要素。提供默认绑定和默认行为（服务行为和终结点行为），让最终的配置文件显得更简洁。
- 支持更多的通信场景：提供路由服务使 WCF 能够适应于更复杂的网络环境。提供了对 WS-Discovery 的支持，使服务可以动态注册，也使客户端可以动态地调用可用的服务。
- 与 WF 的集成：实现了声明式的基于长运行时间的工作流服务 (Long-Running Workflow Service)，在框架级别实现了 WCF 和 WF 更好的集成。
- 对 REST 提供更好的支持：WebHttp 服务得到了加强，并赋予了更多的新特性和工具，使 REST 服务的部署变得更加容易。

10.1 简化开发体验

对于很多 WCF 的初学者来说，WCF 复杂的配置令他们望而却步。虽然 Visual Studio 提供了一个易于使用的配置工具，但是也只有在对 WCF 配置的结构有基本认识的情况下才能得心应手地使用它。如果要做到灵活自如地直接对配置文件进行编辑，真的要求读者对 WCF 有比较深入的认识。为了让读者能够“稍微”地从烦琐的配置中解放出来，最新版本的 WCF 支持一种简化的配置方式。为什么说“稍微”呢？主要因为之前支持的配置系统从结构上依然被沿用，新的配置方式仅仅是在此结构的基础上做出一些简化而已，并且原来的那套配置在新的 WCF 中是完全被支持的。

10.1.1 默认终结点

WCF 的很多初学者是从之前的 Web 服务上转移过来的，他们非常怀念 .asmx Web 服务无配置的服务寄宿方式，只需要在定义 Web 服务的时候在服务操作方法上应用 `System.Web.Services.WebMethodAttribute` 特性就可以了，完全可以不需要手工进行相应的配置，因为 Web 服务运行时会自动添加默认的配置。但是对于 WCF 来说，在进行服务寄宿的时候，必须以编程或者配置的方式为服务添加至少一个终结点，而终结点需要具备基本的 ABC 三要素。

对于最新版本 WCF 的编程人员来说，也可以采用无配置的服务寄宿了，这主要得益于 WCF 提供的默认终结点机制。所谓默认终结点，顾名思义，就是在尚未为寄宿的服务添加任何终结点的时候，WCF 会自动根据服务的基地址 (Base Address) 创建一个或者多个默认的终结点。

举个简单的例子，现在有一个叫做 `GreetingService` 的服务，它实现了两个服务契约

IHello 和 IGoodbye。下面的代码片段提供了服务类型和服务契约接口的定义。

```
[ServiceContract]
public interface IHello
{
    [OperationContract]
    void SayHello(string name);
}
[ServiceContract]
public interface IGoodbye
{
    [OperationContract]
    void SayGoodbye(string name);
}
public class GreetingService : IHello, IGoodbye
{
    public void SayHello(string name)
    {
        //省略实现
    }
    public void SayGoodbye(string name)
    {
        //省略实现
    }
}
```

现在创建一个简单的控制台程序作为服务的宿主，并在不提供任何配置文件的情况下调用如下的代码对服务进行自我寄宿。当用于寄宿服务的 `ServiceHost` 被开启之后，打印出其具有的终结点信息。

```
Uri baseHttpAddress = new Uri("http://127.0.0.1/greetingservice ");
Uri baseTcpAddress = new Uri("net.tcp://127.0.0.1/greetingservice ");
ServiceHost host = new ServiceHost(typeof(GreetingService), baseHttpAddress,
baseTcpAddress);
host.Open();
int index = 0;
foreach (ServiceEndpoint endpoint in host.Description.Endpoints)
{
    Console.WriteLine("Endpoint {0}", ++index);
    Console.WriteLine("\tAddress: {0}\n\tBinding: {1}\n\tContract: {2}",
        endpoint.Address, endpoint.Binding, endpoint.Contract.Name);
}
```

输出结果:

```
Endpoint 1
    Address: http://127.0.0.1/greetingservice
    Binding: System.ServiceModel.BasicHttpBinding
    Contract: IHello
Endpoint 2
    Address: http://127.0.0.1/greetingservice
    Binding: System.ServiceModel.BasicHttpBinding
    Contract: IGoodbye
Endpoint 3
    Address: net.tcp://127.0.0.1/greetingservice
    Binding: System.ServiceModel.NetTcpBinding
    Contract: IHello
```

Endpoint 4

```
Address: net.tcp://127.0.0.1/greetingservice
Binding: System.ServiceModel.NetTcpBinding
Contract: IGoodbye
```

从输出的结果不难看出, 虽然我们没有以任何形式为寄宿的服务提供终结点, 但是 WCF 会自动为之添加四个默认的终结点。之所以是四个默认终结点, 原因在于: WCF 会为服务实现的每一个服务契约基于指定的每一个基地址创建一个终结点。在本例中, 服务 GreetingService 实现了两个服务契约 (IHello 和 IGoodbye), 并且在寄宿过程中又为它指定了两个基地址, 所以最终被自动创建的默认终结点是四个。对于自动创建的终结点, 其地址和服务契约分别来源于指定的基地址和服务实现的契约, 那么采用的绑定又是如何确定的呢?

1. 默认终结点的绑定是如何确定的

从上面的例子可以看到, 对于自动创建的四个默认终结点, 如果采用基于 HTTP 协议的地址, 则采用 BasicHttpBinding 作为其终结点绑定。如果地址是基于 TCP 协议的, 作为终结点绑定的则为 NetTcpBinding。所以说定义在基地址中表示传输协议 (Scheme) 的前缀决定了采用的绑定类型。

但是, 为什么基于 HTTP 协议的地址的终结点会自动采用 BasicHttpBinding, 而不是 WSHHttpBinding 或 WS2007HttpBinding 呢? 实际上, 基地址的协议类型和最终作为默认终结点的类型之间的匹配关系是通过配置决定的。在 <system.serviceModel> 配置节中具有一个名为 <protocolMapping> 的子节点。它包含了一系列用于定义传输协议类型 (Scheme) 和绑定类型匹配关系的配置元素。

如果打开基于 .NET Framework 4.0 的配置文件 machine.config.comments (该配置文件所在的目录为 %Windir%\Microsoft.NET\Framework\v4.0.30319\Config), 会发现 <protocolMapping> 配置节具有如下的定义。具体来说, <protocolMapping> 配置节定义了四种传输协议 (HTTP、TCP、Named Pipe 和 MSMQ) 和对应的绑定类型 (BasicHttpBinding、NetTcpBinding、NetNamedPipeBinding 和 NetMsmqBinding) 之间的匹配关系。这实际上代表了默认的协议绑定映射关系, 这也是在上面的例子中基于 HTTP 协议的默认终结点会采用 BasicHttpBinding 作为绑定类型的原因。除了 scheme 和 binding 这两个配置属性之外, <protocolMapping> 的配置元素还具有另外一个额外的配置属性 bindingConfiguration, 表示对具体绑定配置的引用。

```
<system.serviceModel>
  <protocolMapping>
    <add scheme="http" binding="basicHttpBinding" bindingConfiguration="" />
    <add scheme="net.tcp" binding="netTcpBinding" bindingConfiguration="" />
    <add scheme="net.pipe" binding="netNamedPipeBinding" bindingConfiguration="" />
    <add scheme="net.msmq" binding="netMsmqBinding" bindingConfiguration="" />
  </protocolMapping>
  ...
</system.serviceModel>
```

如果默认的协议与绑定映射关系不满足具体应用场景的要求，可以直接修改 `machine.config` 或者基于具体应用的 `App.config` 或 `Web.config`。比如对于上面的例子，如果为之添加一个配置文件并进行如下的配置将基于 HTTP 的绑定类型设置为 `WS2007HttpBinding`，再次运行实例程序，将会发现默认创建的终结点类型发生了相应的改变。

```
<configuration>
  <system.serviceModel>
    <protocolMapping>
      <add scheme="http" binding="ws2007HttpBinding" />
    </protocolMapping>
  </system.serviceModel>
</configuration>
```

输出结果：

```
Endpoint 1
  Address: http://127.0.0.1/greetingservice
  Binding: System.ServiceModel.WS2007HttpBinding
  Contract: IHello
Endpoint 2
  Address: http://127.0.0.1/greetingservice
  Binding: System.ServiceModel.WS2007HttpBinding
  Contract: IGoodbye
Endpoint 3
  Address: net.tcp://127.0.0.1/greetingservice
  Binding: System.ServiceModel.NetTcpBinding
  Contract: IHello
Endpoint 4
  Address: net.tcp://127.0.0.1/greetingservice
  Binding: System.ServiceModel.NetTcpBinding
  Contract: IGoodbye
```

2. 默认终结点是如何被添加的

接下来具体介绍默认终结点机制是如何实现的，具体来讲就是表示默认终结点的 `ServiceEndpoint` 对象是如何被添加到用于表示寄宿服务描述的 `ServiceDescription` 的终结点列表（对应于 `ServiceDescription` 的 `Endpoints` 属性）中的。要了解默认终结点自动添加的原理，需要涉及 WCF 4.0 为 `ServiceHostBase` 添加的一个新的 `AddDefaultEndpoints` 方法。

```
public abstract class ServiceHostBase :
  CommunicationObject, IExtensibleObject<ServiceHostBase>, IDisposable
{
  //其他成员
  public virtual ReadOnlyCollection<ServiceEndpoint> AddDefaultEndpoints();
}
```

从方法名称不难看出，这个方法用于实现为 `ServiceHost` 添加默认终结点。从上面给出的关于这个方法的定义可以知道这是一个公有方法，可以在具体的服务寄宿应用中被直接调

用。当这个方法被调用的时候, WCF 会按照我们之前介绍的策略(为指定的每一个基地址和服务实现的契约组合添加一个终结点, 终结点绑定的类型决定于<protocolMapping>配置)进行默认终结点的添加。方法的返回值表示添加的默认终结点集合。

当 ServiceHost 在开启的时候, WCF 会检验其 Description 属性表示的服务描述是否具有至少一个终结点。如果没有, 会自动调用这个 AddDefaultEndpoints 方法以添加默认的终结点。比如在下面的代码片段中, 在开启 ServiceHost 之前调用 AddServiceEndpoint 方法添加了一个终结点, 最终默认终结点将不会被添加, 所以 ServiceHost 最终只会有一个唯一的终结点。

```
Uri baseHttpAddress = new Uri("http://127.0.0.1/greetingservice ");
Uri baseTcpAddress = new Uri("net.tcp://127.0.0.1/greetingservice ");
ServiceHost host = new ServiceHost(typeof(GreetingService), baseHttpAddress,
baseTcpAddress);
host.AddServiceEndpoint(typeof(Hello), new WSHttpBinding(), "manuallyadded");
host.Open();
...
```

输出结果:

```
Endpoint 1
  Address: http://127.0.0.1/greetingservice/manuallyadded
  Binding: System.ServiceModel.WSHttpBinding
  Contract: Hello
```

由于公有的 AddDefaultEndpoints 方法可以手工被调用, 所以在调用 AddServiceEndpoint 方法之后再调用该方法, ServiceHost 最终将会具有 5 个终结点。

```
Uri baseHttpAddress = new Uri("http://127.0.0.1/greetingservice ");
Uri baseTcpAddress = new Uri("net.tcp://127.0.0.1/greetingservice ");
ServiceHost host = new ServiceHost(typeof(GreetingService), baseHttpAddress,
baseTcpAddress);
host.AddServiceEndpoint(typeof(Hello), new WSHttpBinding(),
"manuallyadded");
host.AddDefaultEndpoints();
host.Open();
...
```

输出结果:

```
Endpoint 1
  Address: http://127.0.0.1/greetingservice/manuallyadded
  Binding: System.ServiceModel.WSHttpBinding
  Contract: Hello
Endpoint 2
  Address: http://127.0.0.1/greetingservice
  Binding: System.ServiceModel.WS2007HttpBinding
  Contract: Hello
Endpoint 3
  Address: http://127.0.0.1/greetingservice
  Binding: System.ServiceModel.WS2007HttpBinding
  Contract: IGoodbye
Endpoint 4
  Address: net.tcp://127.0.0.1/greetingservice
```



```

Binding: System.ServiceModel.NetTcpBinding
Contract: IHello
Endpoint 5
Address: net.tcp://127.0.0.1/greetingservice
Binding: System.ServiceModel.NetTcpBinding
Contract: IGoodbye

```

10.1.2 默认绑定配置

在传统的配置方式下,如果需要对终结点的绑定(无论是系统绑定还是自定义绑定)进行定制,都需要配置一个“具名”的绑定,然后将这个名称指定为终结点配置节的 `bindingConfiguration` 属性,将其显式地应用到终结点绑定上。

比如,我需要采用 `WS2007HttpBinding` 作为终结点绑定,并且需要采用 `Message` 安全模式和用户名密码认证,就需要按照如下的方式进行配置。首先需要在 `<bindings>/<ws2007HttpBinding>` 节点下定义一个具体的 `WS2007HttpBinding`,除了进行所需的安全相关配置之外,这个配置的绑定必须具有一个名字(`name="defaultBinding"`)。然后将绑定的配置名称指定为终结点的配置属性 `bindingConfiguration`,这就意味着终结点采用了配置的绑定。

```

<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding name="defaultBinding">
          <security mode="Message">
            <message clientCredentialType="UserName" />
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    <services>
      <service ...>
        <endpoint bindingConfiguration="defaultBinding"... />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

对于 WCF 传统的配置系统,并不存在“默认绑定”的概念。或者说所谓的“默认绑定”就是所有属性值均采用默认值的绑定,我们并不能显式地设置一个默认绑定。但是在具体的开发场景中,这样的需求是很常见的,因为在同一个应用里面,采用的绝大部分绑定都具有相同的配置。为了解决这样的问题,最新的 WCF 支持对于默认绑定的配置。

在最新的 WCF 中,配置的绑定具有两种类型,一种是传统的具名绑定,即需要显式指定一个名称的绑定。另一种是默认绑定,默认绑定不要指定名称。对于上面给出的针对 `WS2007HttpBinding` 的配置,如果我们将配置属性 `name` 去掉,它就成了一个默认的绑定。对于所有将 `WS2007HttpBinding` 作为绑定的终结点,如果并没有对 `bindingConfiguration` 配置

属性进行显式设置, 这个默认绑定的所有配置就自动应用到了这些终结点的绑定上。

```
<configuration>
  <system.serviceModel>
    <bindings>
      <ws2007HttpBinding>
        <binding>
          <security mode="Message">
            <message clientCredentialType="UserName" />
          </security>
        </binding>
      </ws2007HttpBinding>
    </bindings>
    ...
  </system.serviceModel>
</configuration>
```

10.1.3 默认行为配置

默认行为配置和默认绑定配置的作用类似, 它允许我们在配置中定义不具名的服务行为或者终结点行为。不过在介绍默认行为配置之前, 还是来介绍传统的服务行为和终结点行为采用怎样的配置方式。

在传统的配置系统下, 无论是服务行为还是终结点行为, 我们都必须为之指定一个名称。而服务和终结点的配置节都具有一个 `behaviorConfiguration` 配置属性, 该属性用于设置配置的行为名称。正是通过这样一个配置属性, 配置的服务行为能够应用到目标服务上, 而目标终结点也能够使用配置的终结点行为。

比如下面的一段配置中, 我们配置两个名称为 `defaultBehavior` 的行为。其中一个为终结点行为, 它实际上应用了 `ServiceDebugBehavior` 行为并将 `IncludeExceptionDetailInFaults` 设置为 `True`, 这样可以使服务端抛出的异常的详细信息通过错误消息传播到客户端以利于查错和纠错。另一个为服务行为, 它将默认使用的 `DataContractSerializer` 的 `MaxItemsInObjectGraph` 属性设置为最大值, 以实现大数据对象的序列化和反序列化。这两个行为最终被分别应用到了寄宿的服务及它的终结点上。

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <endpointBehaviors>
        <behavior name="defaultBehavior">
          <dataContractSerializer maxItemsInObjectGraph="2147483647"/>
        </behavior>
      </endpointBehaviors>
      <serviceBehaviors>
        <behavior name="defaultBehavior">
          <serviceDebug includeExceptionDetailInFaults="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

```

    <service behaviorConfiguration="defaultBehavior" ...>
      <endpoint behaviorConfiguration="defaultBehavior".../>
    </service>
  </services>
</system.serviceModel>
</configuration>

```

如果需要将这两个行为定义成默认的服务行为和终结点行为，只需像下面给出的配置一样将它们的 `name` 属性去掉。在这种情况下，对于该配置作用范围内配置的所有服务和终结点，如果并没有对其 `behaviorConfiguration` 进行显式设置，它们将具有对应的默认行为。

```

<configuration>
  <system.serviceModel>
    <behaviors>
      <endpointBehaviors>
        <behavior>
          <dataContractSerializer maxItemsInObjectGraph="2147483647"/>
        </behavior>
      </endpointBehaviors>
      <serviceBehaviors>
        <behavior>
          <serviceDebug includeExceptionDetailInFaults="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    ...
  </system.serviceModel>
</configuration>

```

默认行为配置具有一个默认绑定配置所不具有的属性，那就是配置的继承性。举个简单的例子来说明默认行为配置的继承性代表什么意思。假设创建一个如图 10-1 所示的用于服务寄宿（IIS 寄宿）的 Web 项目。为了对 .svc 文件进行结构化的管理，我们对其进行分类，并将同类的 .svc 文件置于相同的子目录下。在这里我们建立了一个 `Erp` 的子目录用于存放所有关于 ERP 相关服务的 .svc 文件，在这里仅具有一个唯一的基于订单服务的 `OrderService.svc`。

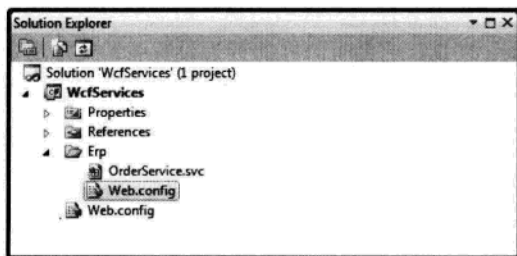


图 10-1 层次化服务配置文件

读者应该注意到了，在 Web 项目的根目录和子目录 `Erp` 中均定义了一个 `Web.config`。这也是很常用的并且我个人推荐的配置方式：将公共的配置定义在外层的 `Web.config` 中，而子目录下 `Web.config` 用于定义在该目录下的所有 .svc 文件对应的服务的配置。现在假设两个

Web.config 分别具有如下的配置。

WcfServices\Web.config:

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <dataContractSerializer ignoreExtensionDataObject="true"
            maxItemsInObjectGraph="65536" />
          <serviceDebug includeExceptionDetailInFaults="true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

WcfServices\Erp\Web.config:

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <dataContractSerializer maxItemsInObjectGraph="2147483647" />
          <serviceTimeouts transactionTimeout="00:01:00" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service name="Artech.WcfServices.Service.OrderService">
        <endpoint binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Contract.IOrderService"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

从上面给出的两段配置可以看到, 两个配置文件中均定义了默认的服务行为。那么对于基于 OrderService.svc 的服务 OrderService 来说, 它具有怎样的行为设置呢? 实际上定义在根目录下 Web.config 的默认服务行为会自动被子目录\Erp 继承, 所以 OrderService 具有的服务行为是两者的“合并”, 它具有的默认服务行为和下面的配置等效。我们将定义在上级目录下默认行为配置被下级目录继承的特性称为默认行为配置的继承性。

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <dataContractSerializer maxItemsInObjectGraph="2147483647"/>
          <serviceTimeouts transactionTimeout="00:01:00" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
```

```

    ...
</system.serviceModel>
</configuration>

```

10.1.4 标准终结点

绑定的本质就是一系列相关绑定元素的有序集合，而系统绑定就是基于若干典型的通信场景对相关绑定元素的整合。WCF 通过系统绑定对绑定元素进行了定制，那么能够在终结点级别对组成该终结点的 ABC（地址、绑定和契约）三要素也进行相应的定制吗？实际上这对于最新版本的 WCF 是可行的，我们将这个机制称为“标准终结点”。

所谓标准终结点，就是基于典型的通信场景选择组成终结点的要素（主要是绑定和契约）进而创建一个标准的终结点。在使用的时候，如果你需要的终结点要素和标准终结点完全一致，就无须进行重复的设置。如果不一致，则只需要单独对此进行重新设置以覆盖定义在标准终结点的默认设置。

比如说，对于用于发布元数据的终结点总是将 `IMetadataExchange` 作为其契约，并且在大部分情况下使用 `MexHttpBinding`。如果我们基于这两个元素创建一个标准的 `MexEndpoint`，那么在为服务配置发布元数据的终结点的时候只需要指定地址就可以了。实际上 WCF 确实为我们创建了这样一个标准的 `MexEndpoint` 终结点。包含 `MexEndpoint` 终结点在内，WCF 总共为我们定义了如表 10-1 所示的 9 个标准终结点。

表 10-1 标准终结点

标准终结点	描 述
<code>mexEndpoint</code>	用于公开服务元数据的标准终结点
<code>dynamicEndpoint</code>	使用 WS-Discovery 在运行时动态查找终结点地址的标准终结点
<code>discoveryEndpoint</code>	发送/接收发现消息的标准终结点
<code>udpDiscoveryEndpoint</code>	通过 UDP 多播方式发送/接收发现消息的标准终结点
<code>announcementEndpoint</code>	由服务用于发送公告消息的标准终结点
<code>udpAnnouncementEndpoint</code>	由服务用于通过 UDP 绑定发送公告消息的标准终结点
<code>workflowControlEndpoint</code>	可用于对 workflow 实例调用控制操作的标准终结点
<code>webHttpEndpoint</code>	带有自动添加 <code>WebHttpBehavior</code> 行为的 <code>WebHttpBinding</code> 绑定的标准终结点
<code>webScriptEndpoint</code>	带有自动添加 <code>WebScriptEnablingBehavior</code> 行为的 <code>WebHttpBinding</code> 绑定的标准终结点

如果希望直接为某个服务配置一个标准终结点，可以借助于 WCF 4.0 为终结点的配置节添加的一个新的配置属性 `kind`，它表示标准终结点名称。在上面的配置中，我为服务配置了一个标准终结点 `mexEndpoint` 以实现基于 MEX 终结点形式的元数据发布。

```

<configuration>
  <system.serviceModel>
    ...
    <services>
      <service ...>
        <endpoint ... />
        <endpoint kind="mexEndpoint" address="http://127.0.0.1:3721/myservice/mex" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

对于系统绑定来说, WCF 允许通过配置的方式对其进行定制, 标准终结点也不例外。如果标准的终结点默认配置不能满足要求, 可以在配置中对其进行相应的定制。在 WCF 配置节下添加了一个新的子节点<standardEndpoints>, 用于对这 9 个标准终结点进行定制。和自定义绑定一样, 需要为自定义的标准终结点起一个名字。如果某个终结点需要使用到自定义的标准终结点, 标准终结点的名称需要设置到终结点配置节的另一个额外的配置属性 endpointConfiguration 上。

在下面的配置中, 自定义了一个基于 WS-Discovery 1.1 的 udpDiscoveryEndpoint, 并取名为 “wsd11”。这个标准终结点通过终结点配置节的两个属性 kind (kind="udpDiscoveryEndpoint") 和 endpointConfiguration (endpointConfiguration="wsd11") 被添加到寄宿服务的终结点列表中。

```

<configuration>
  <system.serviceModel>
    <services>
      <service ...>
        <endpoint ... />
        <endpoint kind="udpDiscoveryEndpoint" endpointConfiguration="wsd11" />
      </service>
    </services>
    <standardEndpoints>
      <udpDiscoveryEndpoint>
        <standardEndpoint name="wsd11" discoveryVersion="WSDiscovery11" />
      </udpDiscoveryEndpoint>
    </standardEndpoints>
  </system.serviceModel>
  ...
</configuration>

```

10.1.5 无.svc 文件服务激活

在采用 IIS/WAS 进行服务寄宿的情况下, 需要为寄宿的服务创建一个.svc 文件。在通常的情况下仅在该.svc 文件中定义基本的<%@ServiceHost%>指令信息。其中最重要的指令信息自然是通过 Service 属性指定的寄宿服务的类型 (实际上调用 ServiceHostFactory 的 CreateServiceHost 方法传入的第一个参数值)。如果采用自定义 ServiceHost, 还需要定义用于创建 ServiceHost 的 ServiceHostFactory 的类型 (通过 Factory 属性)。

在本书的第9章“扩展(Extension)”中,我们介绍了如何通过自定义 ServiceHost 的方式实现 WCF 与 Unity 这个 IoC 框架进行集成。我们为此创建了自定义的 ServiceHost (UnityServiceHost) 和相应的 ServiceHostFactory (UnityServiceHostFactory)。下面就是采用了 UnityServiceHostFactory 这个自定义 ServiceHostFactory 创建的.svc 的内容。

```
<%@ ServiceHost Service="Artech.WcfServices.Services.ResourceService:
defaultContainer" Factory="Artech.WcfExtensions.IoC.UnityServiceHostFactory,
Artech.WcfExtensions"%>
```

从消息交换的角度来说,客户端对 IIS/WAS 寄宿下服务的调用本质上体现在对.svc 这个真实存在的物理文件的访问。如果服务尚未激活,WCF 最终会根据读取请求的物理文件来激活相应的服务。具体来说,就是获取用于创建 ServiceHost 的 ServiceHostFactory 的类型(如果没有通过<%@ServiceHost%>指令的 Factory 进行显式设置,默认使用的 ServiceHostFactory 的类型为 System.ServiceModel.Activation.ServiceHostFactory)。在正确解析出 ServiceHostFactory 类型之后,通过反射创建 ServiceHostFactory 对象,并最终借助它创建寄宿服务的 ServiceHost。

如果 WCF 的服务端能够根据请求正确地创建出基于目标服务的 ServiceHost,就能解决服务的激活问题。进一步说,如果服务端能够维护一个 Service/ServiceHostFactory 与请求地址之间的映射关系,就可以不再需要.svc 文件,因为.svc 对于服务激活来说就是起到了这样一个映射的作用。在最新的 WCF 中,这样一个映射关系可以在配置文件中设置。换言之,在配置对这个映射关系进行了相应设置之后,将不再需要为服务定义.svc 文件了。

在 <system.serviceModel>/<serviceHostingEnvironment> 配置节下,具有一个 <serviceActivations>子节点。上述的关于 Service/ServiceHostFactory 与请求地址之间的映射关系就定义在这个配置节点下。<serviceActivations>配置节下的配置元素具有三个基本的属性,其中 service 和 factory 对应原来定义在.svc 文件中的<%@ServiceHost>指令的 Service 和 Factory 属性,而 relativeAddress 则表示服务相对服务寄宿的 IIS 站点的地址,该地址必须以.svc 为后缀。下面一段配置与上面给出的.svc 文件具有相同的作用,有了这段配置,就不再需要.svc 了。

```
<configuration>
  <system.serviceModel>
    ...
    <serviceHostingEnvironment>
      <serviceActivations>
        <add relativeAddress="ResourceService.svc"
          service="Artech.WcfServices.Services.ResourceService:
            defaultContainer"
          factory="Artech.WcfExtensions.IoC.UnityServiceHostFactory,
            Artech.WcfExtensions"/>
      </serviceActivations>
    </serviceHostingEnvironment>
  </system.serviceModel>
</configuration>
```

如果需要通过 IIS 的方式来寄宿我们熟悉的 CalculatorService, 在不需要定义.svc 的情况下, 下面的 XML 片段代表了所需的最少配置。借助于默认终结点的自动添加机制, WCF 会为寄宿服务实现的每个服务契约针对于每一个基地址(即.svc 地址)添加一个终结点。由于通过配置属性 relativeAddress 定义的地址就是服务的相对基地址, 因此基于这个地址的终结点会自动添加。

```
<configuration>
  <system.serviceModel>
    <serviceHostingEnvironment>
      <serviceActivations>
        <add service="Artech.WcfServices.Service.CalculatorService"
              relativeAddress="OrderService.svc"/>
      </serviceActivations>
    </serviceHostingEnvironment>
  </system.serviceModel>
</configuration>
```

10.2 路由服务(Routing Service)

在一个典型的服务调用场景中, 具有两个基本的角色, 即服务的消费者和服务的提供者。从消息交换的角度讲, 前者一般是消息的最初发送者, 而后者则是消息的最终接收者。在很多情况下, 由于网络环境的局限, 消息的最初发送者和最终接收者不能直接进行消息交换, 需要一个辅助实现消息路由的中介服务, 即我们接下来要介绍的路由服务。为了让读者对路由服务的功能有一个大概的了解, 我们简单地介绍了一些需要消息路由的场景。

- 服务版本策略的实现: 一个服务具有两个版本, 在不希望客户端做任何改动的情况下利用路由服务将原来路由到 Version 1 的服务转移到 Version 2 上。
- 负载均衡: 通过路由服务动态地将客户端的请求路由到服务集群中相应的服务器。
- 异质协议的服务调用: 目标服务以 TCP 协议发布, 客户端可以以 HTTP 协议访问路由服务, 再由路由服务将请求消息以 TCP 协议发送给目标服务。

10.2.1 路由服务就是一个 WCF 服务

路由服务, 就其本质而言就是一个 WCF 服务, 具体的消息路由操作实现在该服务的某个操作之中。在使用路由服务之前, 也需要对其进行寄宿, 并为之指定一个基于某种绑定的终结点。对于需要被路由的服务的客户端, 除了需要将路由服务的地址作为其消息发送的物理地址之外, 它依然像普通的方式一样对目标服务进行调用。

1. 路由服务类型

既然路由服务本质上就是一个 WCF 服务, 它肯定就对应着某个实现相应服务契约的类

型, 这个类型就是具有如下定义的 `System.ServiceModel.Routing.RoutingService`。

```
[AspNetCompatibilityRequirements(
    RequirementsMode=AspNetCompatibilityRequirementsMode.Allowed)]
[ServiceBehavior(AddressFilterMode=AddressFilterMode.Any,
    InstanceContextMode=InstanceContextMode.PerSession,
    UseSynchronizationContext=false,
    ValidateMustUnderstand=false)]
public sealed class RoutingService : ISimplexDatagramRouter,
    ISimplexSessionRouter,
    IRequestReplyRouter,
    IDuplexSessionRouter,
    IDisposable
{
    //ISimplexDatagramRouter
    [OperationBehavior(Impersonation = ImpersonationOption.Allowed)]
    IAsyncResult ISimplexDatagramRouter.BeginProcessMessage(Message message,
        AsyncCallback callback, object state);
    void ISimplexDatagramRouter.EndProcessMessage(IAsyncResult result);

    //ISimplexSessionRouter
    [OperationBehavior(Impersonation = ImpersonationOption.Allowed)]
    IAsyncResult ISimplexSessionRouter.BeginProcessMessage(Message message,
        AsyncCallback callback, object state);
    void ISimplexSessionRouter.EndProcessMessage(IAsyncResult result);

    //IRequestReplyRouter
    [OperationBehavior(Impersonation = ImpersonationOption.Allowed)]
    IAsyncResult IRequestReplyRouter.BeginProcessRequest(Message message,
        AsyncCallback callback, object state);
    Message IRequestReplyRouter.EndProcessRequest(IAsyncResult result);

    //IDuplexSessionRouter
    [OperationBehavior(Impersonation=ImpersonationOption.Allowed)]
    IAsyncResult IDuplexSessionRouter.BeginProcessMessage(Message message,
        AsyncCallback callback, object state);
    void IDuplexSessionRouter.EndProcessMessage(IAsyncResult result);

    void IDisposable.Dispose();
}
```

我们根据其定义来简单地分析一下表示路由服务的 `RoutingService`。首先, `RoutingService` 实现了四个服务契约接口, 分别是 `ISimplexDatagramRouter`、`ISimplexSessionRouter`、`IRequestReplyRouter` 和 `IDuplexSessionRouter`。在这里从 `RoutingService` 对它们的实现不难看出, 这四个契约接口以异步模式定义了一个唯一的操作。其中定义在 `IRequestReplyRouter` 中的叫做 `ProcessRequest`, 定义在其他三个契约接口的叫做 `ProcessMessage`。实际上 `ProcessRequest/ProcessMessage` 就是真正实现消息路由的服务操作。

其次, 实现消息路由的该操作方法的 `BeginProcessMessage/BeginProcessRequest` 方法上应用了 `OperationBehaviorAttribute` 特性并将 `Impersonation` 设置为 `ImpersonationOption.Allowed`, 意味着允许身份模拟 (关于身份模式, 请参阅本书的第 8 章“授权与审核 (Authorization and Auditing)”)。

最后, 在类型级别应用了 `AspNetCompatibilityRequirementsAttribute` 特性并将 `RequirementsMode` 属性设置为 `AspNetCompatibilityRequirementsMode.Allowed`, 意味着路由服务允许与 ASP.NET 兼容 (关于 ASP.NET 兼容模式请参阅上册的第 7 章“服务寄宿 (Hosting)”)。同时应用了 `ServiceBehaviorAttribute` 特性将 `AddressFilterMode` 设置为 `AddressFilterMode.Any`, 意味着 WCF 会关闭基于地址的消息筛选机制。这一点对于路由服务非常重要, 因为它允许路由服务处理携带的目标地址 (WS-Addressing 的 <To> 报头) 与本终结点不一致的请求消息。而在很多情况下, 请求消息携带的目标地址一般都是目标服务的地址, 而不是作为中介的路由服务的地址。

2. 路由服务契约

`RoutingService` 显式实现的四个服务契约接口具有相似的操作方法的定义, 其主要的目的在于对不同消息交换模式和会话的支持。

针对某个服务操作的调用可以通过三种不同的消息交换模式来实现, 即数据报/单向 (`Datagram/One-Way`)、请求/回复 (`Request/Reply`) 和双工 (`Duplex`)。而 `RoutingService` 实现的契约接口 `ISimplexDatagramRouter`/`ISimplexSessionRouter`、`IRequestReplyRouter` 和 `IDuplexSessionRouter` 就是分别基于这三种不同的消息交换模式来定义的。

其中 `ISimplexDatagramRouter` 和 `ISimplexSessionRouter` 的不同之处在于其会话模式的定义。前者采用默认的会话模式 (`SessionMode.Allowed`), 后者则强制使用会话 (`SessionMode.Required`)。而基于双工消息交换模式的 `IDuplexSessionRouter`, 除了也将会话模式设置为 `SessionMode.Required` 之外, 还定义了回调的类型 `IDuplexRouterCallback`。下面的代码片段提供了对这四种契约接口的定义。

```
[ServiceContract(Namespace = "...",
                  SessionMode = SessionMode.Allowed)]
public interface ISimplexDatagramRouter
{
    [OperationContract(AsyncPattern = true, IsOneWay = true, Action = "")]
    IAsyncResult BeginProcessMessage(Message message, AsyncCallback callback,
                                     object state);
    void EndProcessMessage(IAsyncResult result);
}
[ServiceContract(Namespace = "...", SessionMode = SessionMode.Required)]
public interface ISimplexSessionRouter
{
    [OperationContract(AsyncPattern = true, IsOneWay = true, Action = "")]
    IAsyncResult BeginProcessMessage(Message message, AsyncCallback callback,
                                     object state);
    void EndProcessMessage(IAsyncResult result);
}
[ServiceContract(Namespace = "...",
                  SessionMode = SessionMode.Allowed)]
public interface IRequestReplyRouter
{
    [GenericTransactionFlow(TransactionFlowOption.Allowed)]
```

```

[OperationContract(AsyncPattern = true, IsOneWay = false, Action = "*",
    ReplyAction = "**")]
IAsyncResult BeginProcessRequest(Message message, AsyncCallback callback,
    object state);
Message EndProcessRequest(IAsyncResult result);
}
[ServiceContract(Namespace = "...", SessionMode = SessionMode.Required,
    CallbackContract = typeof(IDuplexRouterCallback))]
public interface IDuplexSessionRouter
{
    [GenericTransactionFlow(TransactionFlowOption.Allowed)]
    [OperationContract(AsyncPattern = true, IsOneWay = true, Action = "**")]
    IAsyncResult BeginProcessMessage(Message message, AsyncCallback callback,
        object state);
    void EndProcessMessage(IAsyncResult result);
}

```

四个不同的路由服务契约实际上均定义一个唯一的实现消息路由的操作 `ProcessMessage/ProcessRequest` 操作。只不过这个操作是以异步模式 (`AsyncPattern = true`) 定义的, 所以体现为 `BeginProcessMessage/BeginProcessRequest` 和 `EndProcessMessage/EndProcessRequest` 两个操作方法的组合(关于异步模式操作的定义, 请参阅上册的第4章“契约(Contract)”)。由于操作旨在实现对请求消息(和回复消息)的路由, 所以操作接受一个代表路由消息的类型为 `Message` 的输入参数。而对于 `IRequestReplyRouter` 接口的路由操作具有一个 `Message` 类型的返回值, 代表被路由的回复消息。

由于 `ISimplexDatagramRouter/ISimplexSessionRouter` 专门针对数据报模式的消息交换, 所以应用在操作方法上的 `OperationContractAttribute` 特性将 `IsOneWay` 属性设置为 `True`, 这一点很好理解。

值得一提的是, 对于面向双工消息交换模式的 `IDuplexSessionRouter` 接口来说, 应用在操作方法上的 `OperationContractAttribute` 特性同样将此属性设置为 `True`。我们知道, 所谓的双工消息交换模式实际上可以看做是多次基于简单模式(数据报和请求/回复模式)的消息交换的组合。如果忽略服务端对客户端的回调, 单独来看双工模式下服务调用采用的消息交换模式, 它可以是单向的, 也可以是基于请求/回复模式的。

既然 `IDuplexSessionRouter` 的路由操作是单向的, 那么最终的回复消息是如何被路由到客户端的呢? 实际上, 在这种情况下, 不论是针对服务端回调客户端的消息, 还是最终调用完成后的回复消息, 都是通过路由服务对客户端的回调来实现消息的路由的。

此外, 定义在不同服务契约接口中的路由操作都具有一个相同的特性, 即应用在它们上面的 `OperationContractAttribute` 特性的 `Action` 属性值均被设置为 `*`。通过 `OperationContractAttribute` 特性定义的服务操作的 `Action` 属性最终用于辅助实现对目标操作的选择。服务端运行时正是通过请求消息 `WS-Addressing` 的这个 `<action>` 报头的值来选择当前操作列表中 `Action` 的值与此一致的操作。但是对于路由服务来说, 请求消息的 `<action>` 报头的值一般是决定于真正的目标操作的, 所以路由服务的运行时是不可能根据请求消息正确地选择路由操作来处理该消息的。

如果不能正确地选择出目标操作来处理请求消息, WCF 的服务端运行时就会退而求其次地选择一个“备用”的操作。而在定义服务契约的时候, 通过 `OperationContractAttribute` 特性将 `Action` 属性设为“*”, 这个操作就成为这样一个备用操作。所以说, 将路由操作的 `Action` 属性设置为“*”的最终目的在于: 在操作选择阶段能够正确地选择该操作来处理请求消息以实现对该消息的路由。关于基于 `Action` 的操作选择机制在上册的第 4 章“契约 (Contract)”及本书的第 9 章“扩展 (Extension)”中都有详细的介绍。

10.2.2 基于消息内容的路由策略

路由服务以一个“中介服务”的形式提供消息路由的功能。具体来说, 服务的客户端将原本应该发送给目标服务的基于某个操作的调用消息转发给路由服务, 而路由服务将接收到的消息作为输入, 转而调用目标服务。路由服务对消息的接收和转发机制如图 10-2 所示。

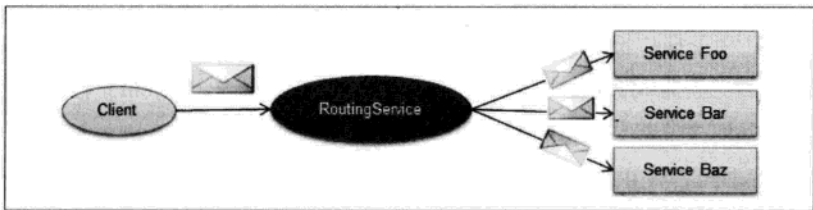


图 10-2 路由服务对消息的接收和转发

所以, 定义在路由服务中的路由操作 `ProcessRequest/ProcessMessage` 需要做的就是根据接收到的消息解析并调用真正的目标服务。至于目标服务的调用以实现对该消息的路由, 和普通的服务调用并没有本质的区别。路由服务的配置中具有用于调用目标服务的所有的客户端终结点的设置。所以需要解决的核心问题是: 如何通过接收到的消息找到用于调用相应服务的客户端终结点。

关于针对请求消息对终结点的选择, 如果你是本书的忠实读者, 或许会具有某种似曾相识之感。在本书的第 9 章“扩展 (Extension)”中介绍服务端运行时框架的构架的时候谈到: 由于不同的终结点可以共享同一个监听地址, 所以一个信道分发器 (一个信道分发器使用一个独立的信道监听器进行请求监听) 具有多个终结点分发器 (一个终结点分发器对应于一个终结点)。当信道分发器接收到请求消息的时候, 也需要根据这个消息选择正确的终结点分发器。

信道分发器对终结点的选择是通过“消息筛选”机制来实现的。每个终结点分发器具有两个消息筛选器 (`MessageFilter`), 其中一个叫做地址筛选器 (`AddressFilter`), 另一个叫做契约筛选器 (`ContractFilter`)。消息筛选器以请求作为输入, 并返回一个布尔类型的值, 如果返回值为 `True`, 则代表对应的终结点分发器适合用于处理接收到的请求消息。关于消

息筛选器，以及针对消息筛选的终结点选择机制，在本书的第 9 章“扩展 (Extension)”中有详细的介绍。

路由服务借用了原本用在信道分发器基于请求消息选择终结点分发器的消息筛选机制，来实现根据被路由的消息选择用于向目标服务路由由消息的客户端终结点。具体的实现是这样的：路由服务维护着一个叫做筛选器表 (FilterTable) 的数据结构，该表的每一个元素代表一个消息筛选器和一个客户端终结点之间的映射关系，而该终结点直接指向某个具体的目标服务。

当路由服务接收到请求消息选择目标服务的时候，只需遍历筛选器表中的每个消息筛选器，并以请求消息作为输入进行评估，最终通过评估结果判断消息对应的终结点是否指向本次路由请求的目标服务。路由服务采用的消息筛选机制大体上如图 10-3 所示。

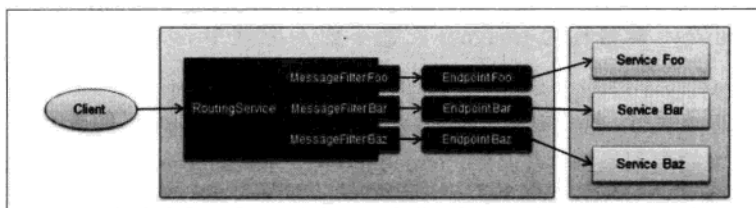


图 10-3 基于消息筛选的终结点选择机制

1. RoutingBehavior 服务行为

上面介绍的这套基于消息筛选的终结点选择机制最终是通过具有如下定义的服务行为 `System.ServiceModel.Routing.RoutingBehavior` 来实现的，所以我们寄宿路由服务 `RoutingService` 的时候，除了为终结点选择适合的绑定和满足相应消息交换模式的服务契约之外，还需在此服务上应用 `RoutingBehavior` 服务行为。

```
public sealed class RoutingBehavior : IServiceBehavior
{
    //其他成员
    public RoutingBehavior(RoutingConfiguration routingConfiguration);
    void IServiceBehavior.AddBindingParameters(ServiceDescription
        serviceDescription, ServiceHostBase serviceHostBase,
        Collection<ServiceEndpoint>endpoints, BindingParameterCollection
        bindingParameters);
    void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription
        serviceDescription, ServiceHostBase serviceHostBase);
    void IServiceBehavior.Validate(ServiceDescription serviceDescription,
        ServiceHostBase serviceHostBase);
}
```

从 `RoutingBehavior` 的定义可以看出，在调用构造函数创建 `RoutingBehavior` 的时候，需要指定一个 `System.ServiceModel.Routing.RoutingConfiguration` 对象作为其参数。该类是对路由策略信息的封装，下面的代码片段列出了 `RoutingConfiguration` 的核心属性的定义。

```

public sealed class RoutingConfiguration
{
    //其他成员
    public MessageFilterTable<IEnumerable<ServiceEndpoint>> FilterTable { get; }
    public bool RouteOnHeadersOnly { get; set; }
    public bool SoapProcessingEnabled { get; set; }
}

```

路由服务采用基于消息筛选机制的路由策略，而整个路由策略的实施依赖一个筛选器表。作为核心的筛选器表通过 `RoutingConfiguration` 的属性 `FilterTable` 表示。除此之外，`RoutingConfiguration` 还具有两个布尔类型的 `SoapProcessingEnabled` 和 `RouteOnHeadersOnly` 属性。前者表示是否按照 SOAP 消息的方式进行路由处理，后者则表示路由的处理是否仅需要使用报头信息。`SoapProcessingEnabled` 和 `RouteOnHeadersOnly` 两个属性默认为 `True`。

在大部分情况下，我们通过配置的方式将 `RoutingBehavior` 行为应用到路由服务上，该行为对应的行为配置元素名称为 `<routing>`。在下面的配置中，为寄宿的路由服务 `RoutingService` 添加了一个基于 `WS2007HttpBinding` 的终结点，该终结点采用 `IRequestReplyRouter` 作为其服务契约。路由服务应用了一个名称为 `routingBehavior` 的服务行为，而 `RoutingBehavior` 行为的配置就包含在其中。`<routing>`配置元素的 `routeOnHeadersOnly` 和 `soapProcessingEnabled` 分别对应 `RoutingConfiguration` 的同名属性，而 `filterTableName` 则表示配置的筛选器表的名称。

```

<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="routingBehavior">
          <routing filterTableName="greetingFilterTable"
            routeOnHeadersOnly="true"
            soapProcessingEnabled="true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service behaviorConfiguration="routingBehavior"
        name="System.ServiceModel.Routing.RoutingService">
        <endpoint binding="ws2007HttpBinding"
          contract="System.ServiceModel.Routing.Irequest
            ReplyRouter" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

接下来具体地介绍 `RoutingBehavior` 配置中引用的筛选器表，以及具体的筛选器是如何进行配置的。

2. 消息筛选器

关于消息筛选器，在之前的章节中已经进行过相应的说明。为了内容的连贯和读者的阅

读体验，在这里再对消息筛选器的概念和几种典型的消息筛选器进行简单的介绍。

消息筛选器旨在实现对给定消息的评估以判断它是否满足某个预先指定的条件（比如消息携带的 AS-Addressing 报头是否和预先指定的一致）。WCF 中所有的消息筛选器都继承自具有如下定义的抽象类 `System.ServiceModel.Dispatcher.MessageFilter`，而具体的消息评估操作实现在两个 `Match` 方法中。

```
public abstract class MessageFilter
{
    //其他成员
    public abstract bool Match(Message message);
    public abstract bool Match(MessageBuffer buffer);
}
```

WCF 定义了一系列的系统定义的消息筛选器。如果我们在消息筛选过程中需要特殊的消息评估逻辑，还可以通过继承这个 `MessageFilter` 抽象类自定义消息筛选器。常用的包括如下 6 种。

- **ActionMessageFilter**: 该筛选器包含一组预先指定的表示 Action 的字符串，判断给定的消息的<Action>报头是否是其中之一。
- **EndpointAddressMessageFilter**: 预先指定一个 `EndpointAddress` 对象，判断给定消息的<To>报头的值是否与之匹配。
- **PrefixEndpointAddressMessageFilter**: 预先指定一个 `EndpointAddress` 对象，判断给定消息的<To>报头的值与指定的地址是否具有相同的前缀。
- **EndpointNameMessageFilter**: 预先指定一个表示终结点名称的字符串，判断给定消息是否具有一个名称为“`System.ServiceModel.Routing.EndpointNameMessageFilter.Name`”的属性，并且属性值与指定的值一致。
- **XPathMessageFilter**: 预先指定一个 XPath 格式的字符串，判断表示消息的 XML 是否满足基于该 XPath 的查询。
- **MatchAllMessageFilter**: 不管消息的内容是什么，都会匹配成功。

除了以上 6 种常用的消息筛选器外，WCF 还为我们定义了一个特殊的关系型消息筛选器，其对应的类型为 `System.ServiceModel.Dispatcher.StrictAndMessageFilter`。`StrictAndMessageFilter` 本身并不进行具体的消息评估的工作，具体的消息评估工作由它所包含的两个具体的消息筛选器来完成。只要当这两个消息筛选器评估结构均返回 `True` 时，该 `StrictAndMessageFilter` 才返回 `True`，否则返回 `False`。

供路由服务使用的所有消息筛选器配置在 `<system.serviceModel>` 配置节的 `<routing>/<filters>` 节点下。该节点下表示单个消息筛选器配置元素的 `<filter>` 具有三个基本的配置属性，即 `name`、`filterType` 和 `filterData`，分别表示消息筛选器的名称、类型和创建筛选器需要的参数信息。

对于上述 6 种消息筛选器, WCF 为它们的类型定义了别名, 分别是 Action、EndpointAddress、Endpoint、XPath 和 MatchAll。在进行配置的时候只需要对 filterType 属性设置相应的类型别名即可。在下面的配置片段中, 定义了 6 种消息筛选器, 它们分别对应上述的 6 种类型。

```
<configuration>
  <system.serviceModel>
    <routing>
      <filters>
        <filter name="ActionMessageFilter1"
          filterType="Action"
          filterData="http://namespace/contract/operation" />
        <filter name="EndpointAddressMessageFilter1"
          filterType="EndpointAddress"
          filterData="http://host/vdir/s.svc/b" />
        <filter name="PrefixEndpointAddressMessageFilter1"
          filterType="EndpointAddressPrefix"
          filterData="http://host/" />
        <filter name="EndpointNameMessageFilter"
          filterType="Endpoint"
          filterData="SvcEndpoint" />
        <filter name="XPathMessageFilter1"
          filterType="XPath"
          filterData="//ns:element" />
        <filter name="MatchAllMessageFilter1"
          filterType="MatchAll" />
      </filters>
    </routing>
  </system.serviceModel>
</configuration>
```

对于 StrictAndMessageFilter 的配置来说, 它也具有相应的类型别名 And。由于它是通过额外的两个消息筛选器来完成具体的消息评估的, 在配置中这两个消息筛选器通过属性 filter1 和 filter2 来表示。在下面的配置中, 我们定义了三个消息筛选器, 前两个是为第三个 StrictAndMessageFilter1 服务的。

```
<configuration>
  <system.serviceModel>
    <routing>
      <filters>
        <filter name="filter1" ... />
        <filter name="filter2" ... />
        <filter name="StrictAndMessageFilter1"
          filterType="And"
          filter1="filter1"
          filter2="filter2" />
      </filters>
    </routing>
  </system.serviceModel>
</configuration>
```

上面介绍的都是系统定义的消息筛选器的配置, 对于自定义的消息筛选器又该如何配置呢? 比如说, 我定义了如下一个 MyMessageFilter, 并且它具有包含两个字符串参数的构造

函数。

```
namespace Artech.CustomMessageFilters
{
    public class MyMessageFiler : MessageFilter
    {
        public MyMessageFiler(string arg1, string arg2)
        {
            //省略实现
        }
        public override bool Match(Message message)
        {
            //省略实现
        }
        public override bool Match(MessageBuffer buffer)
        {
            //省略实现
        }
    }
}
```

我们需要通过下面的方式对这个自定义消息筛选器进行配置。首先需要将 `filterType` 设置为“Custom”，并通过 `customType` 指定消息筛选器的类型，传入构造函数的参数通过 `filterData` 进行设置，参数值用逗号分隔。

```
<configuration>
  <system.serviceModel>
    <routing>
      <filters>
        <filter name="MyMessageFilter1"
          filterType="Custom"
          customType="Artech.CustomMessageFilters.MyMessageFilter,
            Artech.CustomMessageFilters"
          filterData="argValue1, argValue2"/>
      </filters>
    </routing>
  </system.serviceModel>
</configuration>
```

3. 筛选器表

上面配置的具名消息筛选器最终是为创建用于定义路由策略的筛选器表服务的。筛选器表配置在 `<routing>/<filterTables>` 配置节下。表示具体筛选器配置元素的 `<filterTable>` 具有一个必需的配置属性 `name` 表示筛选器的名称，而之前我们介绍的配置在 `RoutingBehavior` 服务行为上的筛选器表名就是指的这个名称。

`<filterTable>` 下的 `<add>` 节点表示具体的消息筛选器和指向目标服务的客户端中的节点之间的映射关系。其中 `filterName` 和 `endpointName` 属性是对配置的消息筛选器和客户端终结点的引用。筛选器表的具体配置可以参考下面的 XML。

```
<configuration>
  <system.serviceModel>
```

```

<client>
  <endpoint name="endpoint1".../>
  <endpoint name="endpoint2".../>
</client>
<routing>
  <filters>
    <filter name="filter1".../>
    <filter name="filter2".../>
  </filters>
  <filterTables>
    <filterTable name="myFilterTable">
      <add filterName="filter1" endpointName="endpoint1" />
      <add filterName="filter2" endpointName="endpoint2" />
    </filterTable>
  </filterTables>
</routing>
</system.serviceModel>
</configuration>

```

在一般的情况下,定义在筛选器表中的每一个消息筛选器对应不同的客户端终结点。除了这种一对一的路由方式,路由服务还支持一对多的“广播式”的消息路由。对于这种情况,只需要在配置筛选器表的时候为同一个消息筛选器匹配多个终结点即可。比如在下面配置的筛选器表中,同一个消息筛选器 filter1 同时匹配了三个不同的客户端终结点(endpoint1、endpoint2 和 endpoint3)。

```

<configuration>
  <system.serviceModel>
    ...
    <routing>
      <filterTables>
        <filterTable name="myFilterTable">
          <add filterName="filter1" endpointName="endpoint1" />
          <add filterName="filter1" endpointName="endpoint2" />
          <add filterName="filter1" endpointName="endpoint3" />
        </filterTable>
      </filterTables>
    </routing>
  </system.serviceModel>
</configuration>

```

以广播模式实施消息路由的路由服务只能采用单向(One-Way)消息交换模式(实际上“广播”这个词本身就隐含了单向通知的含义)。所以在对路由服务进行寄宿的时候,终结点采用的契约不能是 IRequestReplyRouter。

广播模式的消息路由同时说明了另一个路由策略,即如果路由消息匹配筛选器表中的任何一个消息筛选器,它们对应的终结点都将会被选用。但是如果采用请求/回复消息交换模式,要求匹配路由消息的筛选器最多只能有一个,否则会抛出异常。

WCF 还提供了基于优先级的消息筛选机制。对于筛选器表中的每一个配置元素,除了上述的两个必需的 filterName 和 endpointName 配置属性外,还具有一个可选配置属性 priority。该属性表示进行筛选匹配的优先级,数据类型为整型,数字越大,表示级别越高。在默认的情况下,采用零作为其默认的优先级。

当路由服务在通过消息筛选的方式选择终结点的时候,只会选择能够匹配成功的具有最高优先级的终结点。举个例子,对于下面配置的具有四个消息筛选器的筛选表,如果当前的路由消息同时匹配消息筛选器 filter2、filter3 和 filter4,最终选择的是具有最高优先级的 filter3 和 filter4 对应的终结点 endpoint2 和 endpoint3。

```
<configuration>
  <system.serviceModel>
    ...
    <routing>
      <filterTables>
        <filterTable name="myFilterTable">
          <add filterName="filter1" endpointName="endpoint1" priority="2"/>
          <add filterName="filter2" endpointName="endpoint2" priority="1"/>
          <add filterName="filter3" endpointName="endpoint3" priority="1"/>
          <add filterName="filter4" endpointName="endpoint4" priority="0"/>
        </filterTable>
      </filterTables>
    </routing>
  </system.serviceModel>
</configuration>
```

10.2.3 实例演示：如何使用路由服务（S1001）

下面将通过一个具体的实例来演示如何通过路由服务。在这个例子中,我们会创建两个简单的服务 HelloService 和 GoodbyeService。假设客户端不能直接调用这两个服务,需要使用路由服务作为两者之间的中介。整个消息路由的场景如图 10-4 所示,中间的 GreetingService.svc 就代表路由服务,而两个目标服务则通过 HelloService.svc 和 GoodbyeService.svc 表示。路由服务使用的消息筛选器 EndpointAddressMessageFilter,即根据包含在消息中的目标地址来决定应该将请求消息转发给 HelloService.svc 还是 GoodbyeService.svc。



图 10-4 基于地址筛选的消息路由

步骤一：构建解决方案

首先我们创建一个空的解决方案,并添加三个项目和相应的引用,如图 10-5 所示。其中类库项目 Service.Interface 和 Service 分别用于定义服务契约与服务类型,而控制台项目 Client 用来作为进行服务调用的客户端。

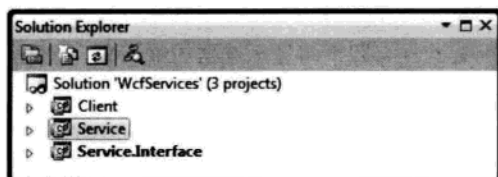


图 10-5 路由服务实例解决方案结构

步骤二：定义服务

我们定义 `HelloService` 和 `GoodbyeService` 两个服务，下面是这两个服务类型和它们实现的契约接口（`IHello` 和 `IGoodbye`）的定义。

服务契约：

```
using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface IHello
    {
        [OperationContract]
        string SayHello(string userName);
    }
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface IGoodbye
    {
        [OperationContract]
        string SayGoodbye(string userName);
    }
}
```

服务类型：

```
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Service
{
    public class HelloService : IHello
    {
        public string SayHello(string userName)
        {
            return string.Format("Hello, {0}", userName);
        }
    }
    public class GoodbyeService : IGoodbye
    {
        public string SayGoodbye(string userName)
        {
            return string.Format("Goodbye, {0}", userName);
        }
    }
}
```

步骤三：寄宿目标服务和路由服务

我们将上面定义的两个服务 HelloService 和 GoodbyeService 及路由服务 RoutingService 寄宿在 IIS 下。为此，我们直接在 IIS 管理器中创建一个 Web 应用（取名为 WcfServices），其物理地址映射为 Service 项目的根目录。然后，不要忘了将该项目编译后的输出目录从默认的 \bin\Debug\ 改为 \bin。接下来在 Service 项目中添加一个 Web.config，并完成如下的配置。

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="routingBehavior">
          <routing filterTableName="greetingFilterTable"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service name="Artech.WcfServices.Service.HelloService">
        <endpoint binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.IHello" />
      </service>
      <service name="Artech.WcfServices.Service.GoodbyeService">
        <endpoint binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.IGoodbye" />
      </service>
      <service name="System.ServiceModel.Routing.RoutingService"
        behaviorConfiguration="routingBehavior">
        <endpoint binding="ws2007HttpBinding"
          contract="System.ServiceModel.Routing.IRequestReplyRouter" />
      </service>
    </services>
    <client>
      <endpoint name="helloService"
        address="http://127.0.0.1/WcfServices/HelloService.svc"
        binding="ws2007HttpBinding"
        contract="*" />
      <endpoint name="goodbyeService"
        address="http://127.0.0.1/WcfServices/GoodbyeService.svc"
        binding="ws2007HttpBinding"
        contract="*" />
    </client>
    <routing>
      <filters>
        <filter name="Address4HelloService"
          filterType="EndpointAddress"
          filterData="http://127.0.0.1/WcfServices/HelloService.svc"/>
        <filter name="Address4GoodbyeService"
          filterType="EndpointAddress"
          filterData="http://127.0.0.1/WcfServices/GoodbyeService.svc"/>
      </filters>
      <filterTables>
        <filterTable name="greetingFilterTable">
          <add filterName="Address4HelloService" endpointName="helloService"/>
          <add filterName="Address4GoodbyeService" endpointName="goodbyeService"/>
        </filterTable>
      </filterTables>
    </routing>
  </system.serviceModel>
</configuration>
```

```

<serviceHostingEnvironment>
  <serviceActivations>
    <add relativeAddress="HelloService.svc"
          service="Artech.WcfServices.Service.HelloService"/>
    <add relativeAddress="GoodbyeService.svc"
          service="Artech.WcfServices.Service.GoodbyeService"/>
    <add relativeAddress="GreetingService.svc"
          service="System.ServiceModel.Routing.RoutingService,
                  System.ServiceModel.Routing,
                  Version=4.0.0.0,
                  Culture=neutral,
                  PublicKeyToken=31bf3856ad364e35"/>
  </serviceActivations>
</serviceHostingEnvironment>
</system.serviceModel>
</configuration>

```

我们对上述的这段配置进行一下简单的分析。首先按照“无.svc 文件服务激活”方式对服务 HelloService、GoodbyeService 和路由服务 RoutingService 进行寄宿，它们的相对地址分别为 HelloService.svc、GoodbyeService.svc 和 GreetingService.svc。它们都具有一个唯一的基于 WS2007HttpBinding 的终结点。由于我们需要路由服务采用请求/回复模式进行消息路由，因此将契约指定为 IRequestReplyRouter。

路由服务上应用了服务行为 RoutingBehavior。配置在该行为上名称为 greetingFilterTable 的筛选器表定义如下。该筛选器表具有两个基于 EndpointAddressMessageFilter 的消息筛选器，配置名称分别为 Address4HelloService 和 Address4GoodbyeService，分别将目标服务 HelloService 和 GoodbyeService 的地址作为筛选条件。而这两个筛选器对应指向目标服务的客户端终结点 helloService 和 goodbyeService。

```

<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="helloService"
                address="http://127.0.0.1/RoutingServiceDemo/HelloService.svc"
                binding="ws2007HttpBinding" contract="*" />
      <endpoint name="goodbyeService"
                address="http://127.0.0.1/RoutingServiceDemo/GoodbyeService.svc"
                binding="ws2007HttpBinding" contract="*" />
    </client>
    <routing>
      <filters>
        <filter name="Address4HelloService"
                filterType="EndpointAddress"
                filterData="http://127.0.0.1/RoutingServiceDemo/HelloService.svc" />
        <filter name="Address4GoodbyeService"
                filterType="EndpointAddress"
                filterData="http://127.0.0.1/RoutingServiceDemo/GoodbyeService.svc" />
      </filters>
      <filterTables>
        <filterTable name="greetingFilterTable">
          <add filterName="Address4HelloService" endpointName="helloService" />
          <add filterName="Address4GoodbyeService" endpointName="goodbyeService" />
        </filterTable>
      </filterTables>
    </routing>
  </system.serviceModel>
</configuration>

```

```

        </filterTable>
    </filterTables>
</routing>
</system.serviceModel>
</configuration>

```

对于上述的配置,细心的读者也许发现一个特殊之处,那就是定义在<client>配置节下的被路由服务使用的终结点的契约被设置成“*”(contract=”*”)。照理说这里的契约应该设置成路由服务实现的服务契约 System.ServiceModel.Routing.IRequestReplyRouter 才对。但是如果真的进行了如此的设置,将会抛出异常,因为将路由服务使用的客户端终结点契约设置成“*”是个强制性的规定。

步骤四: 服务调用

由于调用服务的消息需要通过路由服务这个中介才能抵达真正的目标服务,因此客户端需要将路由服务的地址作为消息发送的目标地址。我们通过 ClientViaBehavior 这个终结点行为实现了物理地址(消息真正发送的目标地址)和逻辑地址(终结点地址)的分离,将消息发送给路由服务的地址(http://127.0.0.1/RoutingServiceDemo/GreetingService.svc)。下面的XML片段代表整个客户端的配置,而 ClientViaBehavior 被定义成默认的终结点行为。

```

<configuration>
  <system.serviceModel>
    <behaviors>
      <endpointBehaviors>
        <behavior>
          <clientVia viaUri="http://127.0.0.1/WcfServices/GreetingService.svc"/>
        </behavior>
      </endpointBehaviors>
    </behaviors>
    <client>
      <endpoint name="helloService"
        address="http://127.0.0.1/WcfServices/HelloService.svc"
        binding="ws2007HttpBinding"
        contract="Artech.WcfServices.Service.Interface.IHello"/>
      <endpoint name="goodbyeService"
        address="http://127.0.0.1/WcfServices/GoodbyeService.svc"
        binding="ws2007HttpBinding"
        contract="Artech.WcfServices.Service.Interface.IGoodbye"/>
    </client>
  </system.serviceModel>
</configuration>

```

借助于这样的配置,可以按照传统的编程方式进行服务的调用,无须再考虑底层消息路由机制的存在。

```

using (ChannelFactory<IHello> channelFactoryHello = new
ChannelFactory<IHello>("helloService"))
using (ChannelFactory<IGoodbye> channelFactoryGoodbye = new
ChannelFactory<IGoodbye>("goodbyeService"))
{
    IHello helloProxy = channelFactoryHello.CreateChannel();
    IGoodbye goodbyeProxy = channelFactoryGoodbye.CreateChannel();
}

```

```

        Console.WriteLine(helloProxy.SayHello("Zhang San"));
        Console.WriteLine(goodbyeProxy.SayGoodbye("Li Si"));
    }

```

输出结果:

```

Hello, Zhang San
Goodbye, Li Si

```

10.2.4 其他路由特性

除了借助于配置的筛选器表提供基于消息筛选的路由之外,路由服务还提供了一些额外的功能和特性,比如异常处理、后备终结点和动态配置等。

1. 异常处理与后备终结点

既然需要使用消息的路由功能,基本上就说明了当时所处的并不是一个简单的网络环境。为了提升路由服务容错能力,路由服务提供基于后备终结点的错误处理机制。对于筛选器表中的每一个条目,除了设置一个与给定消息筛选器相匹配的主终结点(Primary Endpoint)之外,还可以设置一个后备终结点列表。当基于主终结点的服务调用出现网络问题的时候,会转为针对后备终结点的调用,以确保对消息的成功交付。

在<routing>配置节下具有一个<backupLists>子节点,可以在该配置节下配置多个后备终结点列表。除了一个包含后备客户端终结点的配置名称组成的列表之外,每个后备终结点列表必须具有一个名称。而对于筛选器表中的每一个配置元素,还具有一个可选的配置属性 backupList 表示采用的后备终结点列表的名称。

在下面给出的这段配置中,定义了一个仅包含一个消息筛选器(filter1)的筛选器表。这个消息筛选器除了具有一个名称为 primaryEndpoint 的主终结点之外,还具有一个名称为 backup4filter1 的后备终结点列表。这个后备终结点列表包含三个终结点(backupEndpoint1、backupEndpoint2 和 backupEndpoint3)。

```

<configuration>
  <system.serviceModel>
    ...
    <routing>
      <backupLists>
        <backupList name="backup4filter1">
          <add endpointName="backupEndpoint1"/>
          <add endpointName="backupEndpoint2"/>
          <add endpointName="backupEndpoint3"/>
        </backupList>
      </backupLists>
      <filterTables>
        <filterTable name="MyFilterTable">
          <add filterName="filter1" endpointName="primaryEndpoint"
                backupList="backup4filter1"/>
        </filterTable>
      </filterTables>
    </routing>
  </system.serviceModel>
</configuration>

```



```

    </filterTables>
  </routing>
</system.serviceModel>
</configuration>

```

当指向目标服务的主终结点通过消息筛选的方式被解析出来后,路由借此向目标服务发起调用。如果在消息发送过程中,出现 `CommunicationException` (包括其子类,常见的有 `EndpointNotFoundException`、`ServerTooBusyException` 和 `CommunicationObjectFaultedException` 等)或者 `TimeoutException`,路由服务将会转向后备终结点的调用。

此时定义在后备终结点列表中的客户端终结点将会按照配置的顺序依次被调用,直到某个路由成功。以上面的配置来说,如果 `EndpointNotFoundException` 异常在基于主终结点 `primaryEndpoint` 的路由调用中被捕获,将会转向对第一个后备终结点 `backupEndpoint1` 的调用。如果在调用中出现 `TimeoutException` 异常,则选用第二后备终结点 `backupEndpoint2` 继续进行路由调用。如果路由成功,则不再针对后续的后备终结点进行尝试。

有人也许会问, `FaultException` 和其子类 `FaultException<TDetail>` 继承自 `CommunicationException`,那么路由服务会处理在目标服务操作中人为抛出的 `FaultException` 吗?从编程的角度, `FaultException` 异常属于应用级别的异常,应该原封不动地返回给客户端,所以路由服务不应该对此做任何的封装和屏蔽。从实现的角度讲,由于路由操作完全是在消息层面进行的,如果目标服务操作在执行的时候抛出 `FaultException` 异常,路由服务得到的就是一个错误消息,并不会有什么 `FaultException` 异常抛出。

2. 动态配置 (Dynamic Configuration)

当添加其他客户端终结点或需要修改用于路由消息的筛选器时,必须采用某种方法在运行时动态更新配置,以免中断当前正在通过路由服务接收消息的终结点的服务。修改配置文件或宿主应用程序的代码不一定能够始终满足需求,这是因为两种方法都需要回收应用程序,这可能导致丢失当前正在传输的任何消息,并可能在等待服务重新启动时发生停机。

只能以编程方式修改 `RoutingConfiguration`。虽然最初可以使用配置文件来配置服务,但是在运行时只能通过以下方法来修改配置:构造新的 `RoutingConfiguration` 并将其作为参数传递到由 `RoutingExtension` 服务扩展公开的 `ApplyConfiguration` 方法。当前正在传输的所有消息都将采用以前的配置继续路由,而在调用 `ApplyConfiguration` 之后接收的消息将采用新配置。下面的实例演示如何创建路由服务并随后修改配置。

```

RoutingConfiguration routingConfig = new RoutingConfiguration();
routingConfig.RouteOnHeadersOnly = true;
routingConfig.FilterTable.Add(new MatchAllMessageFilter(), endpointList);
RoutingBehavior routing = new RoutingBehavior(routingConfig);
routerHost.Description.Behaviors.Add(routing);
routerHost.Open();
RoutingConfiguration rc2 = new RoutingConfiguration();
ServiceEndpoint clientEndpoint = new ServiceEndpoint();
ServiceEndpoint clientEndpoint2 = new ServiceEndpoint();
rc2.FilterTable.Add(new MatchAllMessageFilter(),

```

```

        new List<ServiceEndpoint>() { clientEndpoint });
    rc2.FilterTable.Add(new MatchAllMessageFilter(),
        new List<ServiceEndpoint>() { clientEndpoint2 });
    rc2.RouteOnHeadersOnly = false;
    routerHost.routerHost.Extensions.Find<RoutingExtension>().ApplyConfigurati
on(rc2);

```

10.3 服务发现 (Service Discovery)

在这之前, 我们进行服务调用的模式都是这样的: 客户端在设计时就预先知道目标服务的地址, 并基于这个地址创建客户端终点对服务进行调用。我们即将介绍的新特性则是你在预先不知道目标服务的地址的情况下, 可以动态地探测可用的服务并调用之。就像我们的无线网卡可以同态地获取周围可用的 WIFI 网络一样。

服务发现解除了客户端和服务端之间的依赖, 允许服务的提供者动态地改变它的地址, 也使新的服务可以很容易地被注册并为人所用。服务发现并不是微软在 .NET 平台下的闭门造车, 而是基于一个开放的标准, 即我们接下来着重介绍的 WS-Discovery。也就是说, 如果 Java 平台的 Web 服务也是基于相同的 WS-Discovery 标准, 那么它们也可以被 WCF 客户端“发现”。

10.3.1 WS-Discovery

WS-Discovery (全称为 Web Services Dynamic Discovery), 是由结构化信息标准促进组织 (OASIS, Organization for the Advancement of Structured Information Standards) 制定的。第一个正式的版本 WS-Discovery 1.0 发布于 2005 年 4 月, 在 2009 年 7 月份 OASIS 发布了 WS-Discovery 1.1, 到目前为止这是最新的版本。

WS-Discovery 定义了两种基本的操作模式, 即 Ad-Hoc 和 Managed。在 Ad-Hoc 模式下, 客户端在一定的网络范围内以广播的形式发送探测 (Probe) 消息以搜寻目标服务。在该探测消息中, 包含相应的搜寻条件。符合该条件的目标服务在接收到探测消息之后将自身相关的信息 (包括地址) 回复给客户端。客户端根据获取到的服务信息, 选择适合的服务进行调用。

对于采用广播形式的 Ad-Hoc 服务发现模式, 可用的目标服务的范围往往只能局限于一个较小的网络。比如对于基于 UDP 的广播的服务探测, 能够被探测到的目标服务只能位于本地子网中。为了解决这个问题, 我们可以采用 Managed 模式。在 Managed 模式下, 一个维护所有可用目标服务的中心发现代理 (Discovery Proxy) 被建立起来, 客户端只需要将探测消息发送到该发现代理, 就可以得到相应的目标服务信息。Ad-Hoc 模式下的广播探测机制在 Managed 模式下被转变成单播形式, 带来的好处就是极大地减轻了网络负载 (Network Traffic)。发现代理不仅用在 Managed 模式下, 在 Ad-Hoc 模式下也可以用到它。

除了上述的这种客户端驱动（客户端主动探测可用的目标服务）模式之外，还可以采用目标服务驱动的模式。在该模式下，客户端开启一个监听程序用于监听上线和离线的服务，而目标服务在上线和离线的时候向监听者发送相应的通知。

要了解 Ad-Hoc 和 Managed 模式下的服务发现机制是如何实现的，就需要了解在整个服务发现模型中各个角色（客户端、目标服务和发现代理）之间是如何协作的。接下来就从消息交换的角度谈谈服务发现模型中各个角色的交互协作问题。

1. Ad-Hoc 模式

图 10-6 所示的序列图揭示了在 Ad-Hoc 模式下客户端和目标服务之间采用的消息交换。目标服务在上线和离线的时候以广播的形式分别发送一个 Hello (1) 和 Bye (6) 消息，而客户端自然是该消息的其中一个接收者。

客户端以广播的形式发送一个 Probe 消息 (2)，该消息包含用于探测的目标服务所满足的条件。对于接收到该广播消息的目标服务，如果自身满足包含在 Probe 消息中的条件，则以单播的形式回复给该客户端一个 Probe Match (简称 PM) 消息 (3)。

如果客户端从 PM 消息中获取的关于目标服务的相关信息足以对其进行调用，则不需要进行后续的消息交换。否则（比如获取的 PM 消息中没有包含目标服务的地址）还需要进行一次旨在实现最终服务调用的服务解析 (Resolution) 的消息交换。客户端以广播的形式发送 Resolve 请求 (4)，该请求中包含被解析目标服务相关的信息。Resolve 和 Probe 广播具有相同的范围。真正的目标服务 (Resolve 消息中用于解析的服务) 将包含自身地址在内的信息以 Resolve Match (简称 RM) 消息的形式回复给客户端 (5)。

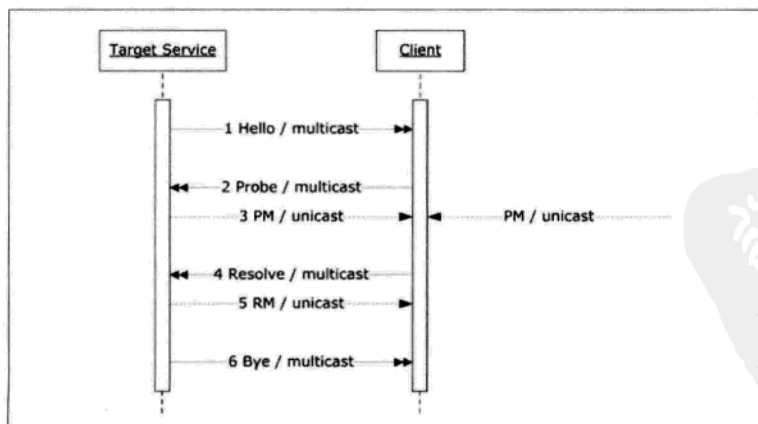


图 10-6 Ad-Hoc 模式下的消息交换

Managed 模式需要一个发现代理对目标服务进行统一管理，但是发现代理本身既可以用 Managed 模式，也可以用于 Ad-Hoc 模式。在 Ad-Hoc 模式下，发现代理对于客户端来说

扮演着目标服务的角色。而对真正的目标服务来说,则扮演着客户端的角色。图 10-7 揭示了在发现代理存在的 Ad-Hoc 模式下,客户端、目标服务和发现代理之间进行的消息交换。

在发现代理存在的情况下,客户端和目标服务之间还是按照上面介绍的方式进行消息交换。比如 Hello/Bye (4)、Probe/PM 和 Resolve/RM (5 和 7)。而发现代理上线和下线的时候,会像真正的目标服务一样发出 Hello/Bye (1) 广播。当接收到客户端发出的 Probe/Resolve 广播的时候 (2),它会像真正的目标服务一样回复以 PM/RM 消息 (3),该回复消息中包含它自身维护的与 Probe/Resolve 请求匹配的目标服务。发现代理同样会接收到真正的目标上下线发出的 Hello/Bye 广播 (4),它可以借此来更新维护的可用目标服务列表。

对于发现代理参与下的 Ad-Hoc 模式,发现代理还提供了一种转换成 Managed 模式的机制。具体的实现是这样的:当发现代理接收到客户端发出的 Probe/Resolve 广播后 (5),会回复给客户端一个 Hello 消息,表明发现代理的存在并可以从 Ad-Hoc 模式转换到 Managed 模式。客户端在接收到该 Hello 消息后 (6),就会将原来以广播的形式发送的 Probe/Resolve 请求转换成指向发现代理的单播形式发送。

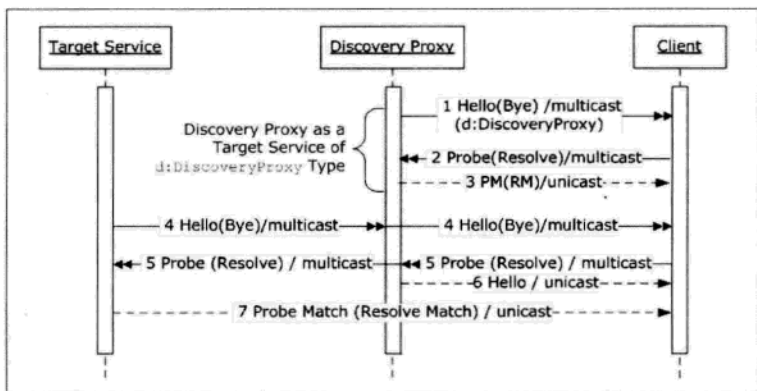


图 10-7 Ad-Hoc 模式下的消息交换 (发现代理)

2. Managed 模式

在 Managed 模式下,由于可用服务都注册到发现代理中,客户端只需要直接和发现代理交互就可以进行可用服务的探测和解析。目标服务也只需要直接和发现代理交互就能实现对自身的注册。在 Managed 模式下,发现代理是真正的核心,而且所有消息交换的方式都是以单播的方式进行的。一来可以解除广播对网络的限制,扩大可用服务的范围,二来也可以避免广播引起对网络的拥堵。

图 10-8 揭示了在 Managed 模式下,客户端、发现代理和目标服务之间进行的消息交换。目标服务上/下线的时候只需要向代理服务发送 Hello (1) /Bye (6) 通知。而客户端用以进行的服务探测和服务解析发送的 Probe (2) /Resolve (4) 也只需要单独发送给发现代理。作为回复,发现代理将 PM(3)/RM(5)返回给客户端。

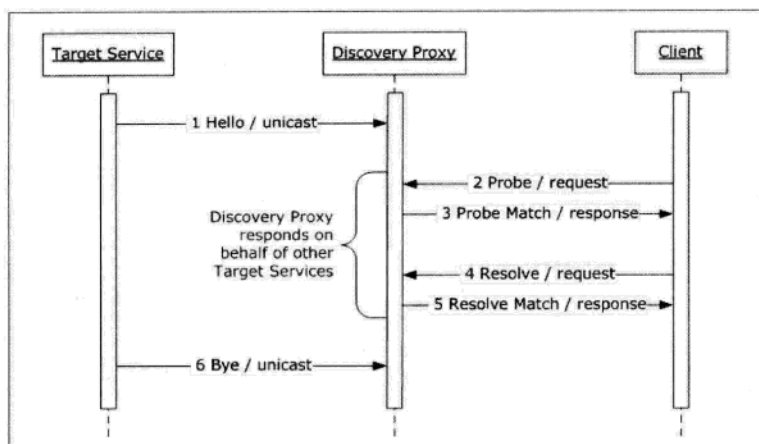


图 10-8 Managed 模式下的消息交换

在对 WS-Discovery 有了基本了解之后，我们将话题拉回 WCF，来继续谈谈 WCF 是如何实现在 WS-Discovery 中定义的服务发现模型的。先来看看 Ad-Hoc 发现模式在 WCF 中的实现。

10.3.2 可被发现的服务 (Discoverable Service)

要让作为服务消费者的客户端能够动态地发现可用的服务，首先要求服务本身具有可被发现的特性。那么一个可被发现的服务和一个一般的服务到底有何不同呢？或者说如何让一个一般的服务在寄宿的时候能够被它潜在的消费者“探测”到呢？

WCF 本质上就是消息交换的通信框架。不论是针对普通的服务操作的调用，还是定义在 WS-Discovery 中的服务的探测 (Probe/PM) 和解析 (Resolve/RM)，本质上都是一种消息的交换。它们并没有本质的不同，或者说唯一不同的就是消息的内容，前者是基于某个服务操作的请求和回复，而后者这是针对服务相关信息的请求和回复。

从消息交换的角度讲，服务发现和元数据的获取比较类似，因为它们交换的消息都是关于服务本身的一些信息。在本书的第 2 章“元数据 (Metadata)”中，我们介绍了元数据的发布具有两种不同的方式：HTTP-GET 和 MEX 终结点。服务发现机制对服务信息交换的实现与基于 MEX 终结点进行服务元数据交换的实现比较类似，因为它也需要一个特殊类型的终结点，即 `DiscoveryEndpoint` (服务发现相关的应用编程接口定义在 `System.ServiceModel.Discovery.dll` 程序集中，对于本节介绍的关于服务发现的类型，如未做特别说明，其命名空间均为 `System.ServiceModel.Discovery`)。

1. DiscoveryEndpoint

在前面介绍“标准终结点”的时候，我们列出的一系列标准终结点列表中就有一个

DiscoveryEndpoint。一个服务必须具有一个 DiscoveryEndpoint 才能成为一个可被发现的服务。而客户端也正是通过 DiscoveryEndpoint 来发现相应的服务的。为了能够更加深刻地认识这个标准终结点，我们不妨先来看看它的定义。

```
public class DiscoveryEndpoint : ServiceEndpoint
{
    //其他成员
    public DiscoveryEndpoint();
    public DiscoveryEndpoint(Binding binding, EndpointAddress endpointAddress);
    public DiscoveryEndpoint(DiscoveryVersion discoveryVersion, ServiceDiscoveryMode
        discoveryMode);
    public DiscoveryEndpoint(DiscoveryVersion discoveryVersion, ServiceDiscoveryMode
        discoveryMode, Binding binding, EndpointAddress endpointAddress);

    public ServiceDiscoveryMode DiscoveryMode { get; }
    public DiscoveryVersion DiscoveryVersion { get; }
    public TimeSpan MaxResponseDelay { get; set; }
}
```

对于一个终结点来说，当然也包括标准终结点，其核心永远是地址、绑定和契约三要素。从上面的代码片段中不难发现，通过构造函数可以为 DiscoveryEndpoint 指定地址和绑定，却无从指定其契约。无须显式指定终结点契约是我们为何要定义这么一个标准终结点的根本目的，而终结点最终采用的契约取决于两个要素：

- 采用的 WS-Discovery 的版本。
- 选择的服务发现的模式 (Ad-Hoc 或者 Managed)。

WS-Discovery 版本通过具有如下定义的 System.ServiceModel.Discovery.DiscoveryVersion 类表示。DiscoveryVersion 具有三个静态的只读属性，分别代表了三个主要的 WS-Discovery 版本。其中 WSDiscoveryApril2005 和 WSDiscovery11 代表两个正式的版本 1.0 和 1.1，而 WSDiscoveryCD1 则代表在 2009 年 1 月份针对 WS-Discovery 1.1 的第一个委员会草案 (CD, Committee Draft)。

```
public sealed class DiscoveryVersion
{
    //其他成员
    public string Name { get; }
    public string Namespace { get; }
    public Uri AdhocAddress { get; }
    public MessageVersion MessageVersion { get; }

    public static DiscoveryVersion WSDiscovery11 { get; }
    public static DiscoveryVersion WSDiscoveryApril2005 { get; }
    public static DiscoveryVersion WSDiscoveryCD1 { get; }
}
```

DiscoveryVersion 具有四个只读属性，Name 表示相应版本的名称。对于表示上述三个版本的 DiscoveryVersion 对象，该属性的值分别为 WSDiscoveryApril2005、WSDiscovery11 和 WSDiscoveryCD1 (与上述的三个静态只读属性的名称一致)。而 Namespace 和 AdhocAddress

则表示具体版本的 WS-Discovery 中采用的命名空间和 Ad-Hoc 模式下各种广播消息中使用的目标地址 (即 <To> 报头携带的地址)。MessageVersion 表示选择的消息版本 (SOAP 版本加上 WS-Addressing 的版本)。默认情况下, DiscoveryEndpoint 的 DiscoveryVersion 属性为 WSDiscovery11。

之前介绍的两种典型的服务发现模式, 即 Ad-Hoc 和 Managed, 则定义在具有如下定义的枚举 System.ServiceModel.Discovery.ServiceDiscoveryMode 中。默认情况下, DiscoveryEndpoint 的 DiscoveryMode 属性值为 Managed。

```
public enum ServiceDiscoveryMode
{
    Adhoc,
    Managed
}
```

DiscoveryEndpoint 采用的契约由 WS-Discovery 的版本和服务发现模式决定, 那么对于这两个要素的不同组合, 最终被选用的终结点契约类型是什么呢? 为此, 我编写了如下一段测试程序。该程序基于不同的 DiscoveryVersion 和 ServiceDiscoveryMode 创建 DiscoveryEndpoint 对象, 最终打印出终结点契约类型的名称。

```
DiscoveryEndpoint endpoint1;
DiscoveryEndpoint endpoint2;
Console.WriteLine("{0,-25}{1,-35}{2,-30}", "", "Ad-Hoc", "Managed");

endpoint1 = new DiscoveryEndpoint(DiscoveryVersion.WSDiscoveryApril2005,
ServiceDiscoveryMode.Adhoc);
endpoint2 = new DiscoveryEndpoint(DiscoveryVersion.WSDiscoveryApril2005,
ServiceDiscoveryMode.Managed);
Console.WriteLine("{0,-25}{1,-35}{2,-30}", "WSDiscoveryApril2005",
endpoint1.Contract.ContractType.Name, endpoint2.Contract.ContractType.Name);

endpoint1 = new DiscoveryEndpoint(DiscoveryVersion.WSDiscovery11,
ServiceDiscoveryMode.Adhoc);
endpoint2 = new DiscoveryEndpoint(DiscoveryVersion.WSDiscovery11,
ServiceDiscoveryMode.Managed);
Console.WriteLine("{0,-25}{1,-35}{2,-30}", "WSDiscovery11",
endpoint1.Contract.ContractType.Name, endpoint2.Contract.ContractType.Name);

endpoint1 = new DiscoveryEndpoint(DiscoveryVersion.WSDiscoveryCD1,
ServiceDiscoveryMode.Adhoc);
endpoint2 = new DiscoveryEndpoint(DiscoveryVersion.WSDiscoveryCD1,
ServiceDiscoveryMode.Managed);
Console.WriteLine("{0,-25}{1,-35}{2,-30}", "WSDiscoveryCD1",
endpoint1.Contract.ContractType.Name, endpoint2.Contract.ContractType.Name);
```

我们将输出的结果通过如表 10-2 所示的表格来表示。从中不难发现, 针对不同的 WS-Discovery 版本和服务发现模式组合, 最终选择的服务契约类型都是不同的。服务契约类型的名称的命名规则为 IDiscoveryContract{Adhoc/Managed}{Discovery Version}。

表 10-2 DiscoveryEndpoint 在不同 WS-Discovery 版本和模式下的契约

	Ad-Hoc	Managed
WSDiscovery April2005	IdiscoveryContract AdhocApril2005	IdiscoveryContract ManagedApril2005
WSDiscovery11	IDiscoveryContractAdhoc11	IDiscoveryContractManaged11
WSDiscoveryCD1	IDiscoveryContractAdhocCD1	IDiscoveryContractManagedCD1

上述的 6 个契约类型对应 6 个接口。不过这些都是内部接口，并不对外公布，但是我们可以通过 Reflector 查看它们的定义。现在就来简单看看 WS-Discovery 1.1 下分别针对 Ad-Hoc 和 Managed 模式的服务契约接口 IDiscoveryContractAdhoc11 和 IDiscoveryContractManaged11 的定义。

IDiscoveryContractAdhoc11:

```
[ServiceContract(Name = "TargetService",
    Namespace = "http://docs.oasis-open.org/ws-dd/ns/
    discovery/2009/01",
    CallbackContract = typeof(IDiscoveryResponseContract11))]
internal interface IDiscoveryContractAdhoc11
{
    [OperationContract(
        Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Probe",
        IsOneWay = true,
        AsyncPattern = true)]
    IAsyncResult BeginProbeOperation(ProbeMessage11 request, AsyncCallback
    callback,
        object state);
    [OperationContract(
        Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Resolve",
        IsOneWay = true,
        AsyncPattern = true)]
    IAsyncResult BeginResolveOperation(ResolveMessage11 request, AsyncCallback
    callback, object state);
    void EndProbeOperation(IAsyncResult result);
    void EndResolveOperation(IAsyncResult result);
    [OperationContract(
        Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Probe",
        IsOneWay = true)]
    void ProbeOperation(ProbeMessage11 request);
    [OperationContract(
        Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Resolve",
        IsOneWay = true)]
    void ResolveOperation(ResolveMessage11 request);
}
```

IDiscoveryContractManaged11:

```
[ServiceContract(Name = "DiscoveryProxy",
    Namespace = "http://docs.oasis-open.org/ws-dd/ns/
    discovery/2009/01")]
internal interface IDiscoveryContractManaged11
{
    [OperationContract(
```



```

        Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Probe",
        ReplyAction = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/ProbeMatches",
        AsyncPattern = true)]
        IAsyncResult BeginProbeOperation(ProbeMessage11 request, AsyncCallback
        callback,
            object state);
        [OperationContract]
            Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Resolve",
            ReplyAction = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/ResolveMatches",
            AsyncPattern = true)]
        IAsyncResult BeginResolveOperation(ResolveMessage11 request, AsyncCallback
        callback, object state);
        ProbeMatchesMessage11 EndProbeOperation(IAsyncResult result);
        ResolveMatchesMessage11 EndResolveOperation(IAsyncResult result);
        [OperationContract]
            Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Probe",
            ReplyAction = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/ProbeMatches"]
        ProbeMatchesMessage11 ProbeOperation(ProbeMessage11 request);
        [OperationContract]
            Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Resolve",
            ReplyAction = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/ResolveMatches"]
        ResolveMatchesMessage11 ResolveOperation(ResolveMessage11 request);
    }

```

服务契约本质上定义了采用的消息交换模式和被交换的消息的格式。对于客户端驱动的服务发现来说, 采用的服务交换不外乎两种类型, 即服务探测 (Probe/PM) 和服务解析 (Resolve/RM)。所以服务契约 `IDiscoveryContractAdhoc11` 和 `IDiscoveryContractManaged11` 实际定义了两组代表这两种消息交换类型的操作 `ProbeOperation` 和 `ResolveOperation`。其中一组是同步操作, 另一组是异步操作。至于契约的名称和命名空间, 以及操作的 `Action`, 与 `ReplyAction` 通过相应的 `ServiceContractAttribute` 和 `OperationContractAttribute` 特性进行相应的定义, 以确保和 WS-Discovery 1.1 规范保持一致。

除了 `DiscoveryMode` 和 `DiscoveryVersion` 这两个只读属性之外, `DiscoveryEndpoint` 还具有一个可读可写的属性 `MaxResponseDelay`, 表示服务在接收到 `Probe` 请求后发送 `PM` 消息延迟发送允许的时间范围。在此 `MaxResponseDelay` 属性规定的时间范围内, 所有目标服务用于响应单个 `Probe` 请求的 `PM` 必须发送出去。如果同时发送所有的 `PM`, 则可能发生网络风暴 (Network Storming)。为了防止发生这种情况, 响应服务在每个 `PM` 发送之间具有一个随机延迟。随机延迟的范围是从 0 到 `MaxResponseDelay`。如果 `MaxResponseDelay` 设置为 0 (默认值), 则在不使用任何延迟的紧凑循环中发送 `PM` 消息。

如果采用编程的方式使用 `DiscoveryEndpoint`, 可以通过在构造函数中传入相应的参数决定采用的 WS-Discovery 版本和服务发现模式, 并通过属性赋值的方式决定 `MaxResponseDelay` 的值。如果采用配置的方式, 这个标准终结点对应的配置元素可以按照如下的方式进行相应的配置。

```
<standardEndpoints>
  <discoveryEndpoint>
    <standardEndpoint name="adhocDiscoveryEndpointConfiguration"
      discoveryVersion="WSDiscovery11" maxResponseDelay="00:00:00.600" />
    </discoveryEndpoint>
  </standardEndpoints>
```

2. UdpDiscoveryEndpoint

由于 DiscoveryEndpoint 需要显式地指定其地址, 所以它只能以单播的方式进行消息交换。WS-Discovery 中的 Ad-Hoc 模式采用广播形式的消息交换, 为此 WCF 为我们创建另一个采用 UDP 广播的标准终结点 UdpDiscoveryEndpoint。如下面的代码片段所示, UdpDiscoveryEndpoint 的 MulticastAddress 属性代表采用的广播地址, 默认值为 “soap.udp://239.255.255.250:3702”。该值也是代表默认 IPV4 广播地址的静态只读属性 DefaultIPv4MulticastAddress 的值。而另一个代表 IPV6 默认广播地址的只读属性 DefaultIPv6MulticastAddress 的值为 “soap.udp://[FF02::C]:3702”。后者代表针对 UDP 传输层的相关设置。

```
public class UdpDiscoveryEndpoint : DiscoveryEndpoint
{
    //其他成员
    public static readonly Uri DefaultIPv4MulticastAddress;
    public static readonly Uri DefaultIPv6MulticastAddress;

    public Uri MulticastAddress { get; set; }
    public UdpTransportSettings TransportSettings { get; }
}
public class UdpTransportSettings
{
    public int DuplicateMessageHistoryLength { get; set; }
    public long MaxBufferPoolSize { get; set; }
    public int MaxMulticastRetransmitCount { get; set; }
    public int MaxPendingMessageCount { get; set; }
    public long MaxReceivedMessageSize { get; set; }
    public int MaxUnicastRetransmitCount { get; set; }
    public string MulticastInterfaceId { get; set; }
    public int SocketReceiveBufferSize { get; set; }
    public int TimeToLive { get; set; }
}
```

UdpDiscoveryEndpoint 的 TransportSettings 用于针对 UDP 传输的相关设置, 其类型为具有如下定义的 System.ServiceModel.Discovery.UdpTransportSettings 类。下面列出了 UdpTransportSettings 的每个属性所代表的含义。

- DuplicateMessageHistoryLength: 传输用于标识重复消息的最大消息哈希数, 默认值为 4112;
- MaxBufferPoolSize: 传输使用的任何缓冲池的最大容量, 默认值为 524288;
- MaxMulticastRetransmitCount: 应重新传输多播消息的最大次数 (第一次发送除外), 默认值为 2;

- **MaxPendingMessageCount**: 已经接收但尚未从每个通道实例的输入队列中移除的消息的最大数量, 默认值为 32;
- **MaxReceivedMessageSize**: 绑定可处理的消息的最大大小, 默认值为 65507;
- **MaxUnicastRetransmitCount**: 应重新传输单播消息的最大次数 (第一次发送除外), 默认值为 1;
- **MulticastInterfaceId**: 该值唯一地标识在发送和接收多播消息时所使用的网络适配器, 默认值为 null;
- **SocketReceiveBufferSize**: 基础 WinSock 套接字上的接收缓冲区的大小, 默认值为 55536;
- **TimeToLive**: 多播数据包可以遍历的网络段跃点数, 默认值为 1。

标准终结点 **UdpDiscoveryEndpoint** 对应的配置元素同样定义了相应的配置属性, 使我们能够按照如下的方式对它采用的广播地址及 UDP 传输层进行自由的配置。

```
<udpDiscoveryEndpoint>
  <standardEndpoint name="adhocDiscoveryEndpointConfiguration"
    discoveryVersion="WSDiscovery11">
    <transportSettings duplicateMessageHistoryLength="2048"
      maxPendingMessageCount="5"
      maxReceivedMessageSize="8192"
      maxBufferPoolSize="262144"/>
    </standardEndpoint>
  </udpDiscoveryEndpoint>
```

3. ServiceDiscoveryBehavior

之前已经说过, 客户端用于获取可用服务发起的请求, 和基于普通服务调用的消息请求并没有本质的不同。匹配的服务在接收到客户端发送的 **Probe/Resolve** 请求后, 会将自己的信息包含在 **PM/RM** 消息中进行回复。现在我们讨论一个核心的问题: 消息的内容如何产生?

对于普通的服务调用, 回复消息的内容最初来源于针对服务实例的操作方法的调用结果。针对服务发现的 **Probe/Resolve** 请求也是一样, 服务端依然存在一个用于返回目标服务信息的“发现服务”, 并且这个服务实现了添加到目标服务的 **DiscoveryEndpoint** 的契约接口。这个服务的类型就是具有如下定义的抽象类 **System.ServiceModel.Discovery.DiscoveryService** 的子类。它实现了 **DiscoveryEndpoint** 基于不同 **WS-Discovery** 版本在 **Ad-Hoc** 和 **Managed** 模式下的 6 个契约接口。

```
public abstract class DiscoveryService :
    IDiscoveryContractAdhocApril2005,
    IDiscoveryContractManagedApril2005,
    IDiscoveryContractAdhoc11,
    IDiscoveryContractManaged11,
    IDiscoveryContractAdhocCD1,
    IDiscoveryContractManagedCD1, ...
{
    //省略成员
}
```

知道了真正用于实现服务发现的服务，我们需要考虑另一个问题：这个继承自 `DiscoveryService` 的发现服务在接收到服务发现请求后是如何被激活的？

如果读者阅读过本书的第9章“扩展 (Extension)”，相信对服务对象的激活机制有了一定的了解。当用于寄宿服务的 `ServiceHost` 对象被开启之后，服务的每个终结点都会转换成一个终结点分发器，这当然也包括上述的 `DiscoveryEndpoint`。激活的服务实例被封装在一个实例上下文中，而服务实例和用于封装服务实例的实例上下文分别通过针对分发运行时的两个特殊的组件来提供，即实例提供者和实例上下文提供者。

如果我们能够自定义用于激活发现服务的实例提供者和实例上下文提供者，并且通过扩展将其应用到针对 `DiscoveryEndpoint` 的分发运行时上，就能够彻底解决发现服务的激活问题。图 10-9 大体上揭示了整个发现服务的激活机制。

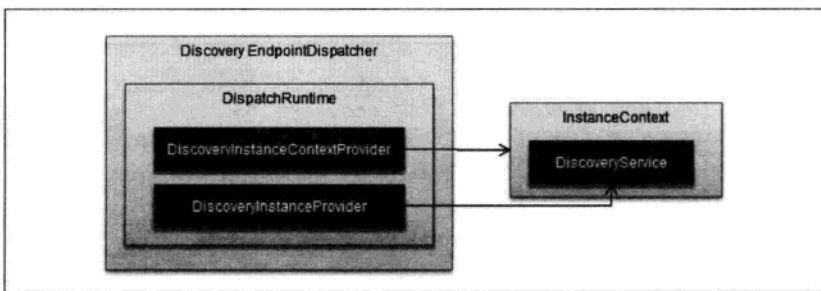


图 10-9 发现服务的激活机制

在 WCF 的具体实现中，这个自定义的实例提供者和实例上下文提供者类型是一个内部类 `ServiceDiscoveryInstanceContextProvider`（它同时实现了 `IInstanceProvider` 和 `IInstanceContextProvider` 两个接口）。而最终将它应用到 `DiscoveryEndpoint` 对应的分发运行时则是通过一个服务行为来实现的，这个服务行为的类型是 `System.ServiceModel.Discovery.ServiceDiscoveryBehavior`。所以一个服务要能够成为一个可被发现的服务，除了要具有一个 `DiscoveryEndpoint` 之外，还必须应用这个 `ServiceDiscoveryBehavior` 服务行为。

10.3.3 目标服务的探测和解析

当应用了 `ServiceDiscoveryBehavior` 行为的服务通过标准终结点 `DiscoveryEndpoint` 被发布出来之后，客户端就可以按照 WS-Discovery 中定义的方式对可用的目标方式进行探测和解析了。这个过程本质上就是一次普通的服务调用，具体来说是针对发布发现服务（非目标服务）的标准终结点 `DiscoveryEndpoint` 的调用，所以客户端也需要具有这样一个匹配的终结点。

1. DiscoveryClient

客户端针对可用目标服务的探测与解析可以通过具有如下定义的 `System.ServiceModel.Discovery.DiscoveryClient` 类型来实现。我们可以直接通过一个 `DiscoveryEndpoint` 对象，或者是 `DiscoveryEndpoint` 的配置名称来创建 `DiscoveryClient` 对象。

```
public sealed class DiscoveryClient : ICommunicationObject, IDisposable, ...
{
    //事件
    public event EventHandler<FindCompletedEventArgs> FindCompleted;
    public event EventHandler<ResolveCompletedEventArgs> ResolveCompleted;

    //构造函数
    public DiscoveryClient(DiscoveryEndpoint discoveryEndpoint);
    public DiscoveryClient(string endpointConfigurationName);

    //Find (Probe)
    public FindResponse Find(FindCriteria criteria);
    public void FindAsync(FindCriteria criteria);
    public void FindAsync(FindCriteria criteria, object userState);

    //Resolve
    public ResolveResponse Resolve(ResolveCriteria criteria);
    public void ResolveAsync(ResolveCriteria criteria);
    public void ResolveAsync(ResolveCriteria criteria, object userState);
}
```

`DiscoveryClient` 定义了两套方法，一套是 `Find/FindAsync`，另一套则是 `Resolve` 和 `Resolve/ResolveAsync`。实际上 `Find` 相当于是 `WS-Discovery` 中的 `Probe`，而 `Resolve` 自然就是 `WS-Discovery` 中的 `Resolve` 了。其中 `Find/Resolve` 采用同步的调用方式，而 `FindAsync/ResolveAsync` 则采用异步调用，异步调用完成的时候会触发事件 `FindCompleted/ResolveCompleted`。

不论是用于可用服务探测的 `Find/FindAsync`，还是用于目标服务解析的 `Resolve/ResolveAsync`，都需要指定相应的匹配条件。前者对应的匹配条件通过类型 `System.ServiceModel.Discovery.FindCriteria` 来表示，而后者对应的匹配条件的类型则是 `System.ServiceModel.Discovery.ResolveCriteria`。

同步方法 `Find/Resolve` 的返回类型分别为 `System.ServiceModel.Discovery.FindResponse` 和 `System.ServiceModel.Discovery.ResolveResponse`。而对于异步调用，则可以通过注册的 `FindCompleted/ResolveCompleted` 事件参数获取类型为 `FindResponse/ResolveResponse` 的返回值。这两个事件的参数类型分别为 `FindCompletedEventArgs` 和 `ResolveCompletedEventArgs`，这两个类型和它们的基类 `System.ComponentModel.AsyncCompletedEventArgs` 定义如下。

```
public class FindCompletedEventArgs : AsyncCompletedEventArgs
{
    //其他成员
    public FindResponse Result { get; }
}
```

```

public class ResolveCompletedEventArgs : AsyncCompletedEventArgs
{
    //其他成员
    public ResolveResponse Result { get; }
}
public class AsyncCompletedEventArgs : EventArgs
{
    //其他成员
    public bool Cancelled { get; }
    public Exception Error { get; }
    public object UserState { get; }
}

```

2. FindCriteria/FindResponse

代表 Probe 请求的 Find 方法接受一个 FindCriteria 类的输入参数作为进行探测可用目标的匹配条件, 该类型的主要属性成员定义如下。其中 ContractTypeNames 代表探测的目标服务实现的契约类型列表, 而 Scopes 和 ScopeMatchBy 则分别代表了探测目标的范围和对范围进行匹配的方式。

```

public class FindCriteria
{
    //其他成员
    public static readonly Uri ScopeMatchByExact;
    public static readonly Uri ScopeMatchByLdap;
    public static readonly Uri ScopeMatchByNone;
    public static readonly Uri ScopeMatchByPrefix;
    public static readonly Uri ScopeMatchByUuid;

    public Collection<XmlQualifiedName> ContractTypeNames { get; }
    public Collection<Uri> Scopes { get; }
    public Uri ScopeMatchBy { get; set; }
}

```

目标的探测范围通过一个 Uri 集合表示。客户端要通过范围进行目标服务的探测, 前提是目标服务预先得与表示范围的 Uri 相关联。服务 (实际上是指服务的某个终结点) 的范围关联通过终结点行为 System.ServiceModel.Discovery.EndpointDiscoveryBehavior 来指定。如下面的代码片段所示, 和 FindCriteria 一样, EndpointDiscoveryBehavior 同样具有一个 Uri 集合类型的 Scopes 属性。

```

public class EndpointDiscoveryBehavior : IEndpointBehavior
{
    //其他成员
    public Collection<Uri> Scopes { get; }
}

```

在服务寄宿的时候, 我们将表示服务范围的 Uri 列表定义在 EndpointDiscoveryBehavior 终结点行为中, 并通过将此行为应用在寄宿服务相应的终结点上, 实现了服务 (终结点) 与范围的关联。在下面的配置中, 我定义了一个名为 scopeMatch 的终结点行为将表示服务范围的两个 Uri 应用到了服务的终结点上。

```

<configuration>
  <system.serviceModel>
    <services>
      <service ...>
        <endpoint behaviorConfiguration="scopeMatch" .../>
        ...
      </service>
    </services>
    <behaviors>
      <endpointBehaviors>
        <behavior name="scopeMatch">
          <endpointDiscovery>
            <scopes>
              <add scope="http://www.example.com/calculator"/>
              <add scope="ldap:///ou=engineering,o=examplecom,c=us"/>
            </scopes>
          </endpointDiscovery>
        </behavior>
      </endpointBehaviors>
    </system.serviceModel>
  </configuration>

```

当服务接收到带有服务范围列表作为匹配条件的探测请求时,在进行匹配判断的时候就验证相应终结点关联的范围是否在指定的范围之内。至于具体采用的范围匹配的逻辑,则取决于 FindCriteria 的 ScopeMatchBy 属性。

FindCriteria 的 ScopeMatchBy 属性类型依然是 Uri。WCF 预选定义了 5 个 Uri 代表相应的进行范围匹配的 5 种算法,它们对应定义在 FindCriteria 中的 5 个静态只读属性 (ScopeMatchByExact、ScopeMatchByLdap、ScopeMatchByNone、ScopeMatchByPrefix 和 ScopeMatchByUuid)。

```

public class FindCriteria
{
    //其他成员
    public static readonly Uri ScopeMatchByExact;
    public static readonly Uri ScopeMatchByLdap;
    public static readonly Uri ScopeMatchByNone;
    public static readonly Uri ScopeMatchByPrefix;
    public static readonly Uri ScopeMatchByUuid;

    public Uri ScopeMatchBy { get; set; }
}

```

下面的列表列出了这 5 个静态字段分别代表了何种服务范围匹配算法,以及各自具有怎样的 Uri 值。实际上这些代表服务范围匹配算法的 Uri 也定义在 WS-Discovery 规范之中,但是为了避免为实现对不同版本的 WS-Discovery 的支持而采用不同的 Uri, WCF 在这里并没有真正地采用定义在相应版本的 WS-Discovery 中的 Uri,而是定义了自己的常量。在对 Probe 消息进行序列化的时候,会转换成相应版本 WS-Discovery 支持的 Uri。

- ScopeMatchByExact: 对 Uri 进行精确匹配, Uri 为 [http://schemas.microsoft.com/ws/2008/06/discovery/strcmp0](http://schemas.microsoft.com/ws/2008/06/discovery/strcmp0;);

- **ScopeMatchByPrefix**: 将指定的 Uri 作为服务范围的前缀进行匹配, Uri 为 `http://schemas.microsoft.com/ws/2008/06/discovery/rfc`;
- **ScopeMatchByLdap**: 按使用 LDAP URL 的段来匹配范围, Uri 为 `http://schemas.microsoft.com/ws/2008/06/discovery/ldap`;
- **ScopeMatchByUuid**: 通过使用 UUID 字符串来完全匹配范围, Uri 为 `http://schemas.microsoft.com/ws/2008/06/discovery/uuid`;
- **ScopeMatchByNone**: 仅匹配那些未指定范围的服务, Uri 为 `http://schemas.microsoft.com/ws/2008/06/discovery/none`。

如果采用 **ScopeMatchByExact**, 进行精确匹配是区分大小写的。而基于前缀匹配的 **ScopeMatchByPrefix**, 实际上是按以 “/” 分隔的段进行匹配的, 比如 `http://contoso/building1` 与范围为 `http://contoso/building/floor1` 的服务相匹配, 但与 `http://contoso/building100` 不匹配, 因为最后两个段不匹配。**ScopeMatchBy** 的值必须指定为上述的 5 种 Uri 之一, 其他各式的 Uri 是无效的。如果未指定范围匹配规则, 则使用 **ScopeMatchByPrefix**。

按照 WS-Discovery 定义的消息交换模式来看, 客户端针对 **Find/FindAsync** 的方法调用实际上就是发送 Probe 请求。符合匹配条件的目标服务会回复以 PM 消息, 该消息中会包含服务相关的元数据信息。最终这些 PM 消息中的内容会被提取出来, 被封装成 **FindResponse** 对象并作为 **Find** 方法的返回值 (或者事件参数 **FindCompletedEventArgs** 的 **Result** 属性)。接下来, 我们将重点介绍 **FindResponse** 这个类型。

```
public class FindResponse
{
    //其他成员
    public DiscoveryMessageSequence GetMessageSequence(EndpointDiscoveryMetadata
        endpointDiscoveryMetadata);
    public Collection<EndpointDiscoveryMetadata> Endpoints { get; }
}
```

如上面的代码所示, **FindResponse** 具有一个核心的只读属性 **Endpoints**, 其类型是一个元素类型为 **System.ServiceModel.Discovery.EndpointDiscoveryMetadata** 的集合。**EndpointDiscoveryMetadata** 就是代表用于描述获取到的满足匹配条件的服务终结点的元数据。**EndpointDiscoveryMetadata** 定义如下, 通过相应的属性可以得到代表目标服务终结点的地址、契约类型列表、监听地址、服务范围和扩展和版本相关信息。

```
public class EndpointDiscoveryMetadata
{
    //其他成员
    public EndpointAddress Address { get; set; }
    public Collection<XmlQualifiedName> ContractTypeNames { get; }
    public Collection<XElement> Extensions { get; }
    public Collection<Uri> ListenUris { get; }
    public Collection<Uri> Scopes { get; }
    public int Version { get; set; }
}
```

FindResponse 除了具有一个表示目标服务终结点元数据的 Endpoints 属性之外, 还具有一个 GetMessageSequence 方法, 该方法以 EndpointDiscoveryMetadata 对象作为输入, 返回一个 System.ServiceModel.Discovery.DiscoveryMessageSequence 对象。DiscoveryMessageSequence 被称为消息序列, 涉及定义在 WS-Discovery 中的一个重要的概念应用序列 (Application Sequence/AppSequence)。

简单起见, 我们可以这样来理解: 采用广播模式的服务发现无法确保消息的有序接收, 即不能确保消息按照它被发送的顺序被接收 (先发先至), 所以需要相应的序号封装在一个被称为 AppSequence 的报头中发送。DiscoveryMessageSequence 类型定义如下, 它的三个只读属性分别对应 AppSequence 报头的相应属性 (Attribute), 具体的含义请参考 WS-Discovery 规范。

```
public class DiscoveryMessageSequence : ...
{
    //其他成员
    public long InstanceId {get; }
    public long MessageNumber {get; }
    public Uri SequenceId {get; }
}
```

3. ResolveCriteria/ResolveResponse

前面介绍了用于进行可用服务探测的 Find/FindAsync 操作的输入和输出, 接下来我们按照相同的方式来分析用于进行服务解析的 Resolve/ResolveAsync 操作的输入和输出。首先来介绍一下用于封装匹配条件的 ResolveCriteria 类型, 下面给出了它核心的属性定义。

```
public class ResolveCriteria
{
    //其他成员
    public EndpointAddress Address { get; set; }
    public Collection<XElement> Extensions { get; }
    public TimeSpan Duration { get; set; }
}
```

其中 Address 属性表示被解析的服务的终结点地址, 而 Exntesions 代表以 XElement 集合表示的扩展信息。Duration 属性表示 Resolve 操作执行的超时时限, 即要求对于 Resolve 请求, 在规定的时限内必须得到回复。如果没有进行显式设置, Durarion 属性采用默认值 20 秒。

作为 Resolve/ResolveAsync 输出的 ResolveResponse 类型定义很简单。它具有两个核心的只读属性, 代表被解析后的服务终结点元数据的 EndpointDiscoveryMetadata 属性和代表消息序列的 MessageSequence 属性。

```
public class ResolveResponse
{
    //其他成员
    public EndpointDiscoveryMetadata EndpointDiscoveryMetadata { get;}
    public DiscoveryMessageSequence MessageSequence { get;}
}
```

10.3.4 实例演示：如何利用服务发现机制实现服务的“动态”调用 (S1002)

接下来我们通过一个简单的例子来演示如何创建和发布一个可被发现的服务，客户端如何在不知道服务终结点地址的情况下动态探测可用的服务并调用之。该实例的解决方案采用如图 10-5 所示的结构，即包含项目 **Service.Interface**（类库）、**Client**（控制台应用）和 **Service**（控制台应用）分别定义服务契约、服务（包括服务寄宿）和客户端程序。

步骤一：创建服务契约和服务

第一个步骤自然是在 **Service.Interface** 项目中定义代表服务契约的接口。还是采用计算服务的例子，下面是服务契约接口和服务类型的定义。

服务契约：

```
using System.ServiceModel;
namespace Artech.WcfServices.Service.Interface
{
    [ServiceContract(Namespace = "http://www.artech.com/")]
    public interface ICalculator
    {
        [OperationContract]
        double Add(double x, double y);
    }
}
```

服务类型：

```
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Service
{
    public class CalculatorService : ICalculator
    {
        public double Add(double x, double y)
        {
            return x + y;
        }
    }
}
```

步骤二：寄宿服务

接下来需要以 **Service** 这个控制台应用作为宿主，对上面定义的 **CalculatorService** 服务进行寄宿，下面是为此添加的配置。

```
<configuration>
  <system.serviceModel>
    <behaviors>
```

```

<serviceBehaviors>
  <behavior>
    <serviceDiscovery />
  </behavior>
</serviceBehaviors>
<endpointBehaviors>
  <behavior name="scopeMapping">
    <endpointDiscovery enabled="true">
      <scopes>
        <add scope="http://www.artech.com/calculatorservice"/>
      </scopes>
    </endpointDiscovery>
  </behavior>
</endpointBehaviors>
</behaviors>
<services>
  <service name="Artech.WcfServices.Service.CalculatorService">
    <endpoint address="http://127.0.0.1:3721/calculatorservice"
      binding="ws2007HttpBinding"
      contract="Artech.WcfServices.Service.Interface.ICalculator"
      behaviorConfiguration="scopeMapping" />
    <endpoint kind="udpDiscoveryEndpoint" />
  </service>
</services>
</system.serviceModel>
</configuration>

```

在上面这段配置中,被寄宿服务的终结点除了有一个基于 `WS2007HttpBinding` 的终结点外,还具有一个 `UdpDiscoveryEndpoint` 标准终结点。我还定义了一个名称为 `scopeMapping` 的终结点行为,该行为通过 `EndpointDiscoveryBehavior` 行为定义了一个代表服务范围的 Uri (`http://www.artech.com/calculatorservice`)。这个终结点行为最终被应用到了第一个终结点,意味着该终结点将此 Uri 作为它的服务范围。最后我还定义了一个包含 `ServiceDiscoveryBehavior` 的默认服务行为。

现在被寄宿的服务具有了 `ServiceDiscoveryBehavior` 行为和一个 `UdpDiscoveryEndpoint`,所以它是一个可被发现的服务了。该服务通过如下一段简单的程序进行自我寄宿。

```

using System;
using System.ServiceModel;
namespace Artech.WcfServices.Service
{
    class Program
    {
        static void Main(string[] args)
        {
            using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
            {
                host.Open();
                Console.Read();
            }
        }
    }
}

```

步骤三：服务的“动态”调用

现在来编写客户端服务调用的程序。假设客户端不知道服务的终结点地址，需要通过服务发现机制进行动态的探测。最终通过探测返回的终结点地址动态地创建服务代理对服务发起调用。我们不需要对客户端程序添加任何配置，可用服务的探测和调用完全通过如下的代码来实现。

```
using System;
using System.ServiceModel;
using System.ServiceModel.Discovery;
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Client
{
    class Program
    {
        static void Main(string[] args)
        {
            DiscoveryClient discoveryClient = new DiscoveryClient(new
                UdpDiscoveryEndpoint());
            FindCriteria criteria = new FindCriteria(typeof(ICalculator));
            criteria.Scopes.Add(new Uri("http://www.artech.com/"));
            FindResponse response = discoveryClient.Find(criteria);

            if (response.Endpoints.Count > 0)
            {
                EndpointAddress address = response.Endpoints[0].Address;
                using(ChannelFactory<ICalculator> channelFactory = new
                    ChannelFactory<ICalculator>(new WS2007HttpBinding(), address))
                {
                    ICalculator calculator = channelFactory.CreateChannel();
                    Console.WriteLine("x + y = {2} when x = {0} and y = {1}", 1, 2,
                        calculator.Add(1, 2));
                }
            }
            Console.Read();
        }
    }
}
```

整段程序分为两个部分，分别实现可用服务的探测和对目标服务的调用。首先基于创建的标准终结点 `UdpDiscoveryEndpoint` 创建 `DiscoveryClient` 对象。然后基于服务契约接口的类型 (`ICalculator`) 创建 `FindCriteria` 对象，并在它的 `Scopes` 集合中添加了一个 `Uri` (`http://www.artech.com/`)。由于我们不曾指定 `FindCriteria` 的 `MatchBy` 属性，默认采用基于前缀的服务范围匹配方式，所以这个 `Uri` 和我们的目标服务是可以匹配的。将此 `FindCriteria` 对象作为输入调用 `Find` 方法，并从返回的 `FindResponse` 中得到目标服务的终结点地址。最后用此终结点地址创建服务代理并进行服务调用。

整个实例程序编写完毕，在启动服务寄宿程序 `Service` 的前提下启动客户端程序 `Client`，定义在 `Client` 中的服务调用能够顺利完成，并得到如下的输出结果。不过中间有一个服务探测过程，需要等待一段时间后才能看到执行结果。

$$x + y = 3 \text{ when } x = 1 \text{ and } y = 2$$

10.3.5 DynamicEndpoint

在上面的例子中我们演示了客户端在不知道目标服务地址的情况下如何通过服务发现机制进行服务的动态调用。从我们的演示来看，这需要两个基本的步骤：

- 借助于 `DiscoveryClient` 通过服务探测（或者解析）获取进行服务调用必需的元数据（主要是目标服务终结点地址）；
- 根据获取的元数据信息创建服务代理进行服务调用。

那么是否有一种方式能够将这两个步骤合二为一呢？答案是肯定的，这就涉及对另一个标准终结点，即 `DynamicEndpoint` 的使用。

为了对 `DynamicEndpoint` 这个标准终结点的作用有一个感性的认识，我们借助于 `DynamicEndpoint` 对上面例子中的服务调用方式进行相应的更改。先为控制台应用 `Client` 添加一个配置文件，然后定义如下一段简单的配置。（S1003）

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="calculatorservice"
        kind="dynamicEndpoint"
        binding="ws2007HttpBinding"
        contract="Artech.WcfServices.Service.Interface.ICalculator"/>
    </client>
  </system.serviceModel>
</configuration>
```

在这段配置中，我定义了一个客户端终结点。不过和我们之前的终结点配置有点不同，因为我们并没有对地址进行相应的设置。之所以可以省略掉对目标服务终结点地址的设置，原因在于我们定义的是一个 `DynamicEndpoint` (`kind="dynamicEndpoint"`)。而我们进行服务调用的程序和基于普通终结点的调用方式完全一样。运行修改后的程序，你会得到一样的执行结果。

```
using (ChannelFactory<ICalculator> channelFactory = new
ChannelFactory<ICalculator>("calculatorservice"))
{
    ICalculator calculator = channelFactory.CreateChannel();
    Console.WriteLine("x + y = {2} when x = {0} and y = {1}", 1, 2,
        calculator.Add(1, 2));
}
```

`DynamicEndpoint` 之所以能够将服务探测和调用这两个步骤统一起来，其本质在于 `DynamicEndpoint` 是由两个终结点组合而成的。其中一个为用于进行服务探测的 `DiscoveryEndpoint`，另一个用于真正服务调用的终结点，该终结点使用 `DynamicEndpoint` 的绑定和契约，而使用 `DiscoveryEndpoint` 探测的地址。关于 `DynamicEndpoint` 的组合性，也可以通过其定义看出来。

```
public class DynamicEndpoint : ServiceEndpoint
{
    //其他成员
    public DynamicEndpoint(ContractDescription contract, Binding binding);
    public DiscoveryEndpointProvider DiscoveryEndpointProvider { get; set; }
    public FindCriteria FindCriteria { get; set; }
}
```

从 `DynamicEndpoint` 的定义可以看出,只需要通过指定终结点 ABC 三要素的绑定和契约就能够构建 `DynamicEndpoint` 这个标准终结点,而地址正是通过 `DiscoveryEndpoint` 终结点动态探测获得的。而具体负责创建这个 `DiscoveryEndpoint` 是通过属性 `DiscoveryEndpointProvider` 表示的 `DiscoveryEndpointProvider` 对象。至于 `FindCriteria` 属性,自然就是在进行服务探测时指定匹配条件。

`DiscoveryEndpointProvider` 是一个抽象类, `DiscoveryEndpoint` 终结点的创建通过定义在该类上的唯一的抽象方法 `GetDiscoveryEndpoint` 实现。而 WCF 定义了两个具体的 `DiscoveryEndpointProvider`, 一个是 `UdpDiscoveryEndpointProvider`, 它会创建一个 `UdpDiscoveryEndpoint`。另外一个为 `ConfigurationDiscoveryEndpointProvider`, 它会根据我们的配置进行 `DiscoveryEndpoint` 的创建。

下面的代码给出了 `DiscoveryEndpointProvider`、`UdpDiscoveryEndpointProvider` 和 `ConfigurationDiscoveryEndpointProvider` 的简单定义,从中可以看出后两个具体的 `DiscoveryEndpointProvider` 类型都是内部类型。

```
public abstract class DiscoveryEndpointProvider
{
    public abstract DiscoveryEndpoint GetDiscoveryEndpoint();
}
internal class UdpDiscoveryEndpointProvider : DiscoveryEndpointProvider
{
    //省略成员
}
internal class ConfigurationDiscoveryEndpointProvider :
DiscoveryEndpointProvider
{
    //省略成员
}
```

在默认的情况下 `DynamicEndpoint` 采用的 `DiscoveryEndpointProvider` 是 `UdpDiscoveryEndpointProvider`, 也就意味着 `DiscoveryEndpoint` 在进行真正的服务调用之前会先创建一个 `UdpDiscoveryEndpoint` 来探测可用调用的服务的终结点地址。从这个意义上讲,我们采用 `DynamicEndpoint` 进行的服务调用,和之前先创建一个基于 `UdpDiscoveryEndpoint` 的 `DiscoveryClient` 对象探测出目标服务的终结点地址,再使用该地址创建服务代理进行服务调用的方式从本质上是一致的。

如果我们不需要采用 `UdpDiscoveryEndpoint` 作为 `DynamicEndpoint` 默认使用的 `DiscoveryEndpoint`,或者需要对 `UdpDiscoveryEndpoint` 进行相应的设置,都可以通过配置来完成。此外可供配置的还有表示服务探测匹配条件的 `FindCriteria`。在下面的配置中,针对

DynamicEndpoint 采用的 UdpDiscoveryEndpoint 进行了相应的设置, 并为 FindCriteria 添加了一个表示服务范围的 Uri。

```
<configuration>
  <system.serviceModel>
    <standardEndpoints>
      <dynamicEndpoint>
        <standardEndpoint name="dynamicEndpointWithScope">
          <discoveryClientSettings>
            <endpoint kind="udpDiscoveryEndpoint"
              endpointConfiguration="adhocDiscoveryEndpointConfiguration"/>
            <findCriteria>
              <scopes>
                <add scope="http://www.artech.com/" />
              </scopes>
            </findCriteria>
          </discoveryClientSettings>
        </standardEndpoint>
      </dynamicEndpoint>
    <udpDiscoveryEndpoint>
      <standardEndpoint name="adhocDiscoveryEndpointConfiguration"
        discoveryVersion="WSDiscovery11">
        <transportSettings duplicateMessageHistoryLength="2048"
          maxPendingMessageCount="5"
          maxReceivedMessageSize="8192"
          maxBufferPoolSize="262144"/>
      </standardEndpoint>
    </udpDiscoveryEndpoint>
  </standardEndpoints>
</system.serviceModel>
</configuration>
```

10.3.6 服务上/下线通知

到目前为止, 我们所介绍的都是基于客户端驱动的服务发现模式, 也就是说客户端主动发出请求以探测和解析可用的目标服务。在介绍 WS-Discovery 的时候, 我们还谈到另外一种服务驱动的模式, 即服务在上线和下线的时候主动对外发出 Hello/Bye 通知。服务上/下线通知机制依赖另外一个 AnnouncementEndpoint 标准终结点。

1. AnnouncementEndpoint

在采用服务端驱动的情况下, 目标服务通过 AnnouncementEndpoint 终结点发送上下线通知, 而客户端通过相同的终结点接收通知。和 DiscoveryEndpoint 终结点一样, AnnouncementEndpoint 终结点也是基于 WS-Discovery 标准的。具体来说, 前者完成 Probe/PM 和 Resolve/RM 消息交换, 而后者则实现 Hello/Bye 消息交换。

下面的代码片段给出了 AnnouncementEndpoint 定义。和 DiscoveryEndpoint 终结点一样, 只需要指定终结点 ABC 三要素的地址和绑定来创建 AnnouncementEndpoint, 而契约决定于采用的 WS-Discovery 的版本 (与采用 Ad-Hoc/Managed 模式无关)。WS-Discovery 的版本通

过属性 `DiscoveryVersion` 表示, 在默认的情况下采用的版本为 `WS-Discovery 1.1`。

```
public class AnnouncementEndpoint : ServiceEndpoint
{
    //其他成员
    public AnnouncementEndpoint(DiscoveryVersion discoveryVersion);
    public AnnouncementEndpoint(Binding binding, EndpointAddress address);
    public AnnouncementEndpoint(DiscoveryVersion discoveryVersion, Binding
        binding,
        EndpointAddress address);

    public DiscoveryVersion DiscoveryVersion { get; }
    public TimeSpan MaxAnnouncementDelay { get; set; }
}
```

另一个属性 `MaxAnnouncementDelay` 与 `DiscoveryEndpoint` 的 `MaxResponseDelay` 属性的作用类似, 通过它设置一个最大允许的通知延迟发送的时间跨度, 以防止在网络出现故障后所有服务同时重新联机所造成的网络风暴。`MaxAnnouncementDelay` 属性的默认值为“00:00:00”, 意味着通知在服务上/下线的时候会被立即发送出去。

我们不妨采用之前分析 `DiscoveryEndpoint` 终结点的方式来分析一下 `AnnouncementEndpoint` 终结点在选择不同的 `WS-Discovery` 版本的情况下具有怎样的契约。为此我们编写了如下一段测试程序, 该程序基于三种不同的版本 (`WSDiscoveryApril2005`、`WSDiscovery11` 和 `WSDiscoveryCD1`) 分别创建 `AnnouncementEndpoint`, 最终将该终结地契约的类型名称输出。

```
AnnouncementEndpoint endpoint;
endpoint = new AnnouncementEndpoint(DiscoveryVersion.WSDiscoveryApril2005);
Console.WriteLine("{0,-20}: {1}", "WSDiscoveryApril2005",
    endpoint.Contract.ContractType.Name);

endpoint = new AnnouncementEndpoint(DiscoveryVersion.WSDiscovery11);
Console.WriteLine("{0,-20}: {1}", "WSDiscovery11",
    endpoint.Contract.ContractType.Name);

endpoint = new AnnouncementEndpoint(DiscoveryVersion.WSDiscoveryCD1);
Console.WriteLine("{0,-20}: {1}", "WSDiscoveryCD1",
    endpoint.Contract.ContractType.Name);
```

输出结果:

```
WSDiscoveryApril2005 : IAnnouncementContractApril2005
WSDiscovery11       : IAnnouncementContract11
WSDiscoveryCD1      : IAnnouncementContractCD1
```

从输出结果可以看到, 基于指定的三种不同的 `WS-Discovery` 版本, `AnnouncementEndpoint` 终结点契约类型分别为 `IAnnouncementContractApril2005`、`IAnnouncementContract11` 和 `IAnnouncementContractCD1`。它们实际上对应着相应的内部接口, 并不对外公布。不过为了更加深刻地认识 `AnnouncementEndpoint` 终结点, 我们不妨来看看基于 `WS-Discovery 1.1` 的契约接口 `IAnnouncementContract11` 的定义。


```

[ServiceContract(Name = "Client",
    Namespace = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01")]
internal interface IAnnouncementContract11
{
    //Hello
    [OperationContract(IsOneWay = true,
        Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Hello")]
    void HelloOperation(HelloMessage11 message);
    [OperationContract(IsOneWay = true,
        Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Hello",
        AsyncPattern = true)]
    IAsyncResult BeginHelloOperation(HelloMessage11 message, AsyncCallback
        callback, object state);
    void EndHelloOperation(IAsyncResult result);
    //Bye
    [OperationContract(IsOneWay = true,
        Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Bye")]
    void ByeOperation(ByeMessage11 message);
    [OperationContract(IsOneWay = true,
        Action = "http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01/Bye",
        AsyncPattern = true)]
    IAsyncResult BeginByeOperation(ByeMessage11 message, AsyncCallback callback,
        object state);
    void EndByeOperation(IAsyncResult result);
}

```

由于 `AnnouncementEndpoint` 终结点旨在实现定义在 WS-Discovery 消息交换模型中的 Hello/Bye 部分, 所以 `IAnnouncementContract11` 仅包含两套操作。其中 `ByeOperation` (同步) 和 `BeginHelloOperation/EndHelloOperation` (异步) 基于服务上线的 Hello 通知, 而 `HelloOperation` (同步) 和 `BeginByeOperation/EndByeOperation` (异步) 基于服务离线的 Bye 通知。由于通知都是单向的, 所以两个操作的 `IsOneWay` 属性为 `True`。服务契约的命名空间、操作的 `Action` 的值都与 WS-Discovery 1.1 规范一致。

`AnnouncementEndpoint` 的两个属性 `DiscoveryVersion` 和 `MaxAnnouncementDelay` 都可以通过它的标准终结点配置元素相应的属性 (`discoveryVersion` 和 `maxAnnouncementDelay`) 来指定。在下面的配置中, 我们定义了一个名称为 `endpoint4April2005` 的 `AnnouncementEndpoint` 终结点, 它基于 `WS-DiscoveryApril2005`, 最大通知延迟的时间为 20 秒。

```

<configuration>
  <system.serviceModel>
    <standardEndpoints>
      <announcementEndpoint>
        <standardEndpoint name="endpoint4April2005"
            discoveryVersion="WSDiscoveryApril2005"
            maxAnnouncementDelay="00:00:20" />
      </announcementEndpoint>
    </standardEndpoints>
  </system.serviceModel>
</configuration>

```

2. UdpAnnouncementEndpoint

由于 `AnnouncementEndpoint` 终结点需要指定具体的地址, 所以它采用的是单播的模式。

为了支持广播模式的通知, WCF 为 `AnnouncementEndpoint` 设计了基于 UDP 的版本, 即 `UdpAnnouncementEndpoint` 标准终结点。

从下面的代码片段中不难发现, `UdpAnnouncementEndpoint` 和之前介绍的 `UdpDiscoveryEndpoint` 具有相同的属性定义。它们都包括具有一个表示广播地址的 `MulticastAddress` 属性, 而另一个属性 `TransportSettings` 用于进行 UDP 传输层设置。正因如此, `UdpAnnouncementEndpoint` 和 `UdpDiscoveryEndpoint` 这两个基于 UDP 传输协议的标准终结点具有相同结构的配置。

```
public class UdpAnnouncementEndpoint : AnnouncementEndpoint
{
    //其他成员
    public Uri MulticastAddress { get; set; }
    public UdpTransportSettings TransportSettings {get; }
}
```

3. 上/下线通知的发送

接下来我们关注另外一个主题: 如何让服务在上/下线的时候具有发送消息的能力? 这自然要用到上面我们介绍的 `AnnouncementEndpoint` 终结点。但是, 我们是否只需要在服务寄宿的时候为寄宿的服务添加这样一个标准终结点就可以了呢? 实则不然。

最终使服务具有通知发送功能的 `AnnouncementEndpoint` 终结点实际上是通过一个服务行为被应用到目标服务上的, 而这个服务行为就是我们之前介绍过的 `ServiceDiscoveryBehavior`。它除了通过实现发现服务的激活以使目标服务可以被探测和解析之外, 还可以为目标服务添加一到多个 `AnnouncementEndpoint` 终结点使得在上/下线的时候对外发出通知。从下面给出的代码片段来看, `ServiceDiscoveryBehavior` 具有一个类型为 `AnnouncementEndpoint` 集合的只读属性 `AnnouncementEndpoints`。

```
public class ServiceDiscoveryBehavior : IServiceBehavior
{
    //其他成员
    public Collection<AnnouncementEndpoint> AnnouncementEndpoints { get; }
}
```

如果采用编程的方式来应用这个服务行为, 可以手工地将创建的 `AnnouncementEndpoint` 终结点添加到 `ServiceDiscoveryBehavior` 的 `AnnouncementEndpoints` 集合中。我们还是推荐采用配置的方式来为服务添加 `AnnouncementEndpoint` 终结点。在 `ServiceDiscoveryBehavior` 对应的配置节下具有一个 `<announcementEndpoints>` 子节点用于配置 `AnnouncementEndpoint` 列表。在下面的配置中, 我们定义了一个包含 `ServiceDiscoveryBehavior` 的默认服务行为, 它的 `AnnouncementEndpoints` 集合中包含一个自定义的 `UdpAnnouncementEndpoint` 终结点。

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
```

```

    <behavior>
      <serviceDiscovery>
        <announcementEndpoints>
          <endpoint kind="udpAnnouncementEndpoint"
            endpointConfiguration="endpoint4April2005"/>
        </announcementEndpoints>
      </serviceDiscovery>
    </behavior>
  </serviceBehaviors>
</behaviors>
<standardEndpoints>
  <announcementEndpoint>
    <standardEndpoint name="endpoint4April2005"
      discoveryVersion="WSDiscoveryApril2005"
      maxAnnouncementDelay="00:00:20" />
  </announcementEndpoint>
</standardEndpoints>
</system.serviceModel>
</configuration>

```

通过 ServiceDiscoveryBehavior 为目标服务添加相应的 AnnouncementEndpoint 终结点使它具有了自动发送上/下线通知的能力。实际上除了这种自动的方式之外,还可以“手动”地进行通知的发送,这就需要用到另外一个具有如下定义的 AnnouncementClient。

```

public sealed class AnnouncementClient : ICommunicationObject, IDisposable
{
    //其他成员
    public event EventHandler<AsyncCompletedEventArgs> AnnounceOnlineCompleted;
    public event EventHandler<AsyncCompletedEventArgs> AnnounceOfflineCompleted;

    public AnnouncementClient();
    public AnnouncementClient(AnnouncementEndpoint announcementEndpoint);
    public AnnouncementClient(string endpointConfigurationName);

    //AnnounceOnline
    public void AnnounceOnline(EndpointDiscoveryMetadata discoveryMetadata);
    public void AnnounceOnlineAsync(EndpointDiscoveryMetadata discoveryMetadata);
    public void AnnounceOnlineAsync(EndpointDiscoveryMetadata discoveryMetadata,
        object userState);
    public IAsyncResult BeginAnnounceOnline(EndpointDiscoveryMetadata
        discoveryMetadata, AsyncCallback callback, object state);
    public void EndAnnounceOnline(IAsyncResult result);

    //AnnounceOffline
    public void AnnounceOffline(EndpointDiscoveryMetadata discoveryMetadata);
    public void AnnounceOfflineAsync(EndpointDiscoveryMetadata discoveryMetadata);
    public void AnnounceOfflineAsync(EndpointDiscoveryMetadata discoveryMetadata,
        object userState);
    public IAsyncResult BeginAnnounceOffline(EndpointDiscoveryMetadata
        discoveryMetadata, AsyncCallback callback, object state);
    public void EndAnnounceOffline(IAsyncResult result);
}

```

AnnouncementClient 最终还是借助于 AnnouncementEndpoint 实现上/下线通知的发送,所以我们在调用构造函数来创建 AnnouncementClient 的时候需要指定一个具体的 AnnouncementEndpoint 对象或者置名称。如果调用无参构造函数,则要求默认的客户端终结

点必须是 `AnnouncementEndpoint` 终结点。

`AnnouncementClient` 具有两套分别用于发送上线和离线通知的方法,方法的输入都是包含被通知服务相关元数据的 `EndpointDiscoveryMetadata` 对象。其中 `AnnounceOnline/AnnounceOffline` 通过同步的方式实现上/下线通知的发送,而异步方式则具有两个方式:一种是传统的 `Beging/End` 的方式,另一种是通过调用 `AnnounceOnlineAsync/AnnounceOfflineAsync` 方法。通知发送结束之后会分别触发 `AnnounceOnlineCompleted/AnnounceOfflineCompleted` 事件。

4. 上/下线通知的接收

前面我们介绍了目标服务在上/下线的时候如何发送通知,接下来我们站在客户端的角度,谈谈如何监听和接收通知。可以在客户端开启一个服务来监听目标服务发送的上/下线通知,而 WCF 已经定义了这样一个服务,即具有如下定义的 `AnnouncementService`。

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single,
                  ConcurrencyMode=ConcurrencyMode.Multiple)]
public class AnnouncementService :
    IAnnouncementContractApril2005,
    IAnnouncementContract11,
    IAnnouncementContractCD1,...
{
    //其他成员
    public event EventHandler<AnnouncementEventArgs> OnlineAnnouncementReceived;
    public event EventHandler<AnnouncementEventArgs> OfflineAnnouncementReceived;
}
public class AnnouncementEventArgs : EventArgs
{
    //其他成员
    public EndpointDiscoveryMetadata EndpointDiscoveryMetadata { get; }
    public DiscoveryMessageSequence MessageSequence { get; }
}
```

对于这个 `AnnouncementService` 服务类型,首先需要关注的是它实现的服务契约。从上面的定义可以看到, `AnnouncementService` 实现了三个契约 `IAnnouncementContractApril2005`、`IAnnouncementContract11` 和 `IAnnouncementContractCD1`, 它们正是 `AnnouncementEndpoint` 针对不同 WS-Discovery 版本所采用的契约。

其次,在 `AnnouncementService` 类型上应用了 `ServiceBehaviorAttribute` 特性,并将 `InstanceContextMode` 属性设置成 `InstanceContextMode.Single`, 所以 `AnnouncementService` 会以单例模式被寄宿。同时 `ConcurrencyMode` 属性被设置为 `ConcurrencyMode.Multiple`, 所以该服务支持并发。

`AnnouncementService` 服务被寄宿的时候所添加的终结点必须是 `AnnouncementEndpoint` 终结点,因为它需要这样的终结点进行通知的监听与接收。当服务上/下线通知被接收之后,事件 `OnlineAnnouncementReceived/OfflineAnnouncementReceived` 分别被触发。通过类型为 `AnnouncementEventArgs` 的事件参数,你可以获得封装目标服务元数据的 `EndpointDiscovery`

Metadata 对象和代表消息序列的 DiscoveryMessageSequence 对象。

实际上我们所说的对目标服务上/下线通知的监听与接收就是在客户端基于寄宿这样一个 AnnouncementService 服务，并通过注册 OnlineAnnouncementReceived/OfflineAnnouncementReceived 这两个事件获得通知。

5. 实例演示：目标服务上/下线通知的发送与接收 (S1004)

现在通过一个简单的实例演示如何通过 ServiceDiscoveryBehavior 服务行为为寄宿的服务添加一个实现上/下线通知的 AnnouncementEndpoint 终结点，以及客户端如何通过对 AnnouncementService 服务的寄宿实现对通知的监听和接收。

我们依然沿用上一个演示实例的解决方案结构，并且直接使用定义好的 CalculatorService。现在为该服务的寄宿定义如下一段配置。我们定义了一个包含 ServiceDiscoveryBehavior 行为的默认服务行为，并且一个 UdpAnnouncementEndpoint 终结点被添加到了 ServiceDiscoveryBehavior 的 AnnouncementEndpoints 集合之中。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.CalculatorService">
        <endpoint address="http://127.0.0.1:3721/calculatorservice"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.ICalculator" />
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceDiscovery>
            <announcementEndpoints>
              <endpoint kind="udpAnnouncementEndpoint" />
            </announcementEndpoints>
          </serviceDiscovery>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

然后通过如下一段简短的代码对 CalculatorService 进行寄宿，与之前的不同之处在于输出了一段程序退出的提示性文字。当服务寄宿程序启动后输入任何字符，服务将会被关闭(离线)。

```
using System;
using System.ServiceModel;
namespace Artech.WcfServices.Service
{
    class Program
    {
        static void Main(string[] args)
```

```

    {
        using (ServiceHost host = new ServiceHost(typeof(CalculatorService)))
        {
            host.Open();
            Console.WriteLine("Enter any key to exit.");
            Console.Read();
        }
    }
}

```

接下来我们需要在客户端通过寄宿 `AnnouncementService` 服务来监听 `CalculatorService` 服务启动和关闭自动发出的通知。先通过如下所示的一段配置为寄宿的 `AnnouncementService` 添加一个 `UdpAnnouncementEndpoint` 终结点。

```

<configuration>
  <system.serviceModel>
    <services>
      <service name="System.ServiceModel.Discovery.AnnouncementService">
        <endpoint kind="udpAnnouncementEndpoint" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

`AnnouncementService` 服务通过如下的代码进行寄宿。由于 `AnnouncementService` 被定义成一个单例服务，我们就可以直接针对一个预先创建好的 `AnnouncementService` 实例来创建用于服务寄宿的 `ServiceHost`。在服务开启之前，我们注册了 `AnnouncementService` 的 `OnlineAnnouncementReceived` 和 `OfflineAnnouncementReceived` 两个事件，在它接收到目标服务上/下线通知的时候会输出目标服务终结点的地址和契约名称。

```

using System;
using System.ServiceModel;
using System.ServiceModel.Discovery;
namespace Artech.WcfServices.Client
{
    class Program
    {
        static void Main(string[] args)
        {
            AnnouncementService announcementService = new AnnouncementService();
            announcementService.OnlineAnnouncementReceived += (sender, e) =>
            {
                string contractTypes = string.Empty;
                Console.WriteLine("Receive Service Online Announcement.");
                Console.WriteLine("\tAddress: {0}",
                    e.EndpointDiscoveryMetadata.Address.Uri);
                Console.WriteLine("\tContract: {0}",
                    e.EndpointDiscoveryMetadata.ContractTypeNames[0]);
            };
            announcementService.OfflineAnnouncementReceived += (sender, e) =>
            {
                string contractTypes = string.Empty;
                Console.WriteLine("Receive Service Offline Announcement.");
                Console.WriteLine("\tAddress: {0}",

```

```

        e.EndpointDiscoveryMetadata.Address.Uri);
        Console.WriteLine("\tContract: {0}",
            e.EndpointDiscoveryMetadata.ContractTypeNames[0]);
    };
    using (ServiceHost host = new ServiceHost(announcementService))
    {
        host.Open();
        Console.Read();
    }
}
}
}
}

```

客户端和服务端所有的配置和编码工作完成之后,先启动客户端开启通知监听服务。然后开启服务端启动服务 `CalculatorService`,最后输入任意键(不要直接关闭控制台窗口)退出服务端程序。此时你会发现客户端的控制台上具有如下的输出。这段文字的前一部分代表 `CalculatorService` 启动的时候(`ServiceHost` 的 `Open` 方法被执行之后)发出的上线通知,后一部分代表服务关闭(`ServiceHost` 的 `Dispose` 方法执行之后)发出的离线通知。

```

Receive Service Online Announcement.
Address: http://127.0.0.1:3721/calculatorservice
Contract: http://www.artech.com/:ICalculator
Receive Service Offline Announcement.
Address: http://127.0.0.1:3721/calculatorservice
Contract: http://www.artech.com/:ICalculator

```

10.3.7 发现代理 (Discovery Proxy)

上面的内容大部分是围绕着 `Ad-Hoc` 模式展开介绍的。`Managed` 模式和 `Ad-Hoc` 的不同之处在于可用服务的终结点通过发现代理来统一管理。客户端在进行可用目标服务探测和解析的时候不再需要发送广播请求,而是直接向发现代理进行探测和解析请求就可以了。

至于发现服务如何进行可用服务的实时维护,则是具体实现上的选择问题。不过 `WS-Discovery` 通过目标服务的通知机制来解决发现代理维护的服务的实时可用性。具体来说就是赋予了发现代理监听服务上/下线通知的能力,并根据接收到的通知来进行可用服务的动态注册和注销。不过与 `Ad-Hoc` 模式下采用广播模式的通知不同,在 `Managed` 模式下,目标服务只需要专门针对发现代理发送通知就可以了。

在 `Ad-Hoc` 模式下,我们采用 `UdpAnnouncementEndpoint` 实现了广播式的通知,而在 `Managed` 模式下则直接使用 `AnnouncementEndpoint` 终结点进行单播式的通知。该终结点的地址就是发现代理的地址。同理,在 `Ad-Hoc` 模式下我们进行广播式服务探测和解析是通过 `UdpDiscoveryEndpoint` 终结点来进行的,在 `Managed` 模式下可以直接使用 `DiscoveryEndpoint` 终结点实现客户端向发现代理单方面的可用服务的探测和解析请求。

注:在本章中提到的 `AnnouncementEndpoint` 终结点,有的地方是泛指包含 `UdpAnnouncementEndpoint` 在内的可用于实现服务上/下线通知的“通知终结点”,有时候则专门指在

Managed 模式下进行单播式通知的 `System.ServiceModel.Discovery.AnnouncementEndpoint`, 望读者能够根据上下文加以区分。相同的情况同样发生在 `DiscoveryEndpoint` 上。

发现代理部仅局限于 Managed 模式, 同样可以使用在 Ad-Hoc 模式下。在 Ad-Hoc 模式下, 发现代理可以像目标服务一样监听来自客户端发出的广播式的 Probe/Resolve 请求, 也可以像客户端一样监听来自服务端发出的广播式的 Hello/Bye 通知。所以 `UdpDiscoveryEndpoint` 和 `UdpAnnouncementEndpoint` 同样可以应用在发现代理上。

发现代理本质上就是一个服务, 它的核心功能就是接收客户端发送的针对可用服务探测和解析的 Probe/Resolve 请求, 并回复以相应的 PM 和 RM 消息。至于上面提到的对目标服务上/下线的通知监听能力, 只是具体实现对可用服务维护的一种方式而已。

1. 通过继承 `DiscoveryProxy` 创建发现代理

发现服务本质上就是一个 WCF 服务, 并且这个服务实现的服务契约定义的操作应该基于定义在 WS-Discovery 中的几种基本的消息交换: Probe/PM、Resolve/RM 和 Hello/Bye。交换的消息针对不同版本的 WS-Discovery (`WSDiscoveryApril2005`、`WSDiscovery11` 和 `WSDiscoveryCD1`) 又具有不同的要求。即使针对某个具体版本的 WS-Discovery, Probe/PM 和 Resolve/RM 的消息也会因采用 Ad-Hoc 或 Managed 模式又有所不同。如果需要创建一个同时支持不同版本 WS-Discovery 的发现代理服务, 就应该实现 `DiscoveryEndpoint` 和 `AnnouncementEndpoint` 终结点所实现的所有服务契约。

所以说要自己从头到尾去定义这样一个发现代理服务并不是一件容易的事情。为了使开发人员可以无须关注具体的消息交换的细节, 帮助他们很容易地定义发现代理, WCF 提供了一个抽象类 `DiscoveryProxy`。我们只需要将我们自定义的发现代理服务类型继承该类并且重写相应的方法就可以了。

下面的代码给出了 `DiscoveryProxy` 的核心方法的定义。正如上面的分析, 作为一个完备的发现代理服务, 应该实现 `DiscoveryEndpoint` 和 `AnnouncementEndpoint` 终结点所实现的所有服务契约, 在这里得到了证实。`DiscoveryProxy` 定义了 4 组抽象的 `OnBegingXxx/OnEndXxx` 方法, 分别针对四个基本的服务发现操作 (消息交换): 服务探测 (Probe/PM)、服务解析 (Resolve/RM)、上线通知 (Hello) 和离线通知 (Bye)。作为继承自 `DiscoveryProxy` 的自定义发现代理服务, 只需要重写这些抽象方法即可。

```
public abstract class DiscoveryProxy :
    IAnnouncementContractApril2005,
    IAnnouncementContract11,
    IAnnouncementContractCD1,
    IDiscoveryContractAdhocApril2005,
    IDiscoveryContractManagedApril2005,
    IDiscoveryContractApril2005,
    IDiscoveryContractAdhoc11,
```



```

IDiscoveryContractManaged11,
IDiscoveryContractAdhocCD1,
IDiscoveryContractManagedCD1, ...
{
    //Find(Probe)
    protected abstract IAsyncResult OnBeginFind(FindRequestContext
        findRequestContext, AsyncCallback callback, object state);
    protected abstract void OnEndFind(IAsyncResult result);

    //Resolve
    protected abstract IAsyncResult OnBeginResolve(ResolveCriteria resolveCriteria,
        AsyncCallback callback, object state);
    protected abstract EndpointDiscoveryMetadata OnEndResolve(IAsyncResult
        result);

    //Online Announcement(Hello)
    protected abstract IAsyncResult OnBeginOnlineAnnouncement(
        DiscoveryMessageSequence messageSequence, EndpointDiscoveryMetadata
        endpointDiscoveryMetadata, AsyncCallback callback, object state);
    protected abstract void OnEndOnlineAnnouncement(IAsyncResult result);

    //Offline Announcement(Bye)
    protected abstract IAsyncResult OnBeginOfflineAnnouncement(
        DiscoveryMessageSequence messageSequence, EndpointDiscoveryMetadata
        endpointDiscoveryMetadata, AsyncCallback callback, object state);
    protected abstract void OnEndOfflineAnnouncement(IAsyncResult result);

    //其他成员
}

```

2. 实例演示：自定义发现代理服务 (S1005)

接下来将通过一个简单的实例演示如何自定义发现代理服务，以及如何利用这个发现代理构建一个基于 Managed 模式的服务发现环境以实现服务的自动注册和服务的动态调用。实例解决方法依然采用前两个演示实例的结构，并且直接使用定义好的 CalculatorService 作为目标服务。

步骤一：创建自定义发现代理服务

我们首先通过继承 DiscoveryProxy 创建一个自定义的发现代理服务，将它取名为 DiscoveryProxyService。由于我们要重写的方法都是异步模式的，OnBeginXxx 的输出和 OnEndXxx 的输入都是一个 IAsyncResult 类型的对象，所以先要定义一个实现 IAsyncResult 接口的类型。为了简单起见，我们在 Service 项目中定义如下一个最为简单的 DiscoveryAsyncResult（其实它根本起不到异步执行的目的）。

```

using System;
using System.ServiceModel.Discovery;
using System.Threading;
namespace Artech.WcfServices.Service
{
    public class DiscoveryAsyncResult : IAsyncResult
    {

```

```

public object AsyncState { get; private set; }
public WaitHandle AsyncWaitHandle { get; private set; }
public bool CompletedSynchronously { get; private set; }
public bool IsCompleted { get; private set; }
public EndpointDiscoveryMetadata Endpoint { get; private set; }

public DiscoveryAsyncResult(AsyncCallback callback, object asyncState)
{
    this.AsyncState = asyncState;
    this.AsyncWaitHandle = new ManualResetEvent(true);
    this.CompletedSynchronously = this.IsCompleted = true;
    if (callback != null)
    {
        callback(this);
    }
}

public DiscoveryAsyncResult(AsyncCallback callback, object asyncState,
    EndpointDiscoveryMetadata Endpoint)
    : this(callback, asyncState)
{
    this.Endpoint = Endpoint;
}
}
}

```

下面自定义如下一个发现代理服务 **DiscoveryProxyService**，我们通过在类型上应用 **ServiceBehaviorAttribute** 特性将 **DiscoveryProxyService** 定义成一个单例服务，并且支持并发。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.ServiceModel;
using System.ServiceModel.Discovery;
namespace Artech.WcfServices.Service
{
    [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
        ConcurrencyMode = ConcurrencyMode.Multiple)]
    public class DiscoveryProxyService : DiscoveryProxy
    {
        public IDictionary<EndpointAddress, EndpointDiscoveryMetadata> Endpoints
        { get; private set; }
        public DiscoveryProxyService()
        {
            this.Endpoints = new Dictionary<EndpointAddress,
                EndpointDiscoveryMetadata>();
        }

        //Find(Probe)
        protected override IAsyncResult OnBeginFind(FindRequestContext
            findRequestContext, AsyncCallback callback, object state)
        {
            var endpoints = from item in this.Endpoints
                where findRequestContext.Criteria.IsMatch(item.
                    Value)
                select item.Value;
            foreach (var endpoint in endpoints)
            {
                findRequestContext.AddMatchingEndpoint(endpoint);
            }
        }
    }
}

```

```

        return new DiscoveryAsyncResult(callback, state);
    }
    protected override void OnEndFind(IAsyncResult result) {}

    //Resolve
    protected override IAsyncResult OnBeginResolve(ResolveCriteria
        resolveCriteria, AsyncCallback callback, object state)
    {
        EndpointDiscoveryMetadata endpoint = null;
        if (this.Endpoints.ContainsKey(resolveCriteria.Address))
        {
            endpoint = this.Endpoints[resolveCriteria.Address];
        }
        return new DiscoveryAsyncResult(callback, endpoint);
    }
    protected override EndpointDiscoveryMetadata OnEndResolve(IAsyncResult
        result)
    {
        return ((DiscoveryAsyncResult)result).Endpoint;
    }

    //OnlineAnnouncement
    protected override IAsyncResult OnBeginOnlineAnnouncement(
        DiscoveryMessageSequence messageSequence, EndpointDiscoveryMetadata
        endpointDiscoveryMetadata, AsyncCallback callback, object state)
    {
        this.Endpoints[endpointDiscoveryMetadata.Address] =
            endpointDiscoveryMetadata;
        return new DiscoveryAsyncResult(callback, state);
    }
    protected override void OnEndOnlineAnnouncement(IAsyncResult result)
    {}

    //OfflineAnnouncement
    protected override IAsyncResult OnBeginOfflineAnnouncement(
        DiscoveryMessageSequence messageSequence, EndpointDiscoveryMetadata
        endpointDiscoveryMetadata, AsyncCallback callback, object state)
    {
        if (this.Endpoints.ContainsKey(endpointDiscoveryMetadata.Address))
        {
            this.Endpoints.Remove(endpointDiscoveryMetadata.Address);
        }
        return new DiscoveryAsyncResult(callback, state);
    }
    protected override void OnEndOfflineAnnouncement(IAsyncResult result) {}
}

```

DiscoveryProxyService 具有一个 `IDictionary<EndpointAddress, EndpointDiscoveryMetadata>` 类型的属性 `Endpoints` 表示可用的目标服务列表。在处理服务上线通知的 `OnBeginOnlineAnnouncement/OnEndOnlineAnnouncement` 方法中将代表上线服务的 `EndpointDiscoveryMetadata` 添加到 `Endpoints` 列表中。而在处理服务离线通知的 `OnBeginOfflineAnnouncement/OnEndOfflineAnnouncement` 方法中则将代表离线服务的 `EndpointDiscoveryMetadata` 从 `Endpoints` 列表中移除。

在处理客户端服务探测请求的 `OnBeginFind/OnEndFind` 方法中，从传入的

FindRequestContext 中获得代表匹配条件的 FindCriteria 对象, 并通过它从 Endpoints 列表中找到匹配的 EndpointDiscoveryMetadata, 最终通过调用的 AddMatchingEndpoint 方法将它们添加到 FindRequestContext 之中。至于用于处理服务解析请求的 OnBeginResolve/OnEndResolve, 则只需要从 Endpoints 列表中将给定的终结点地址一致的 EndpointDiscoveryMetadata 返回就可以了。

步骤二: 寄宿发现代理服务 and 目标服务

现在我们需要寄宿上面创建的自定义发现代理服务 DiscoveryProxyService 和代表目标服务的 CalculatorService, 我们把所有的设置都定义在如下的配置中。

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="Artech.WcfServices.Service.DiscoveryProxyService">
        <endpoint address="net.tcp://127.0.0.1:8888/discoveryproxy/probe"
          binding="netTcpBinding"
          kind="discoveryEndpoint"
          isSystemEndpoint="false" />
        <endpoint address="net.tcp://127.0.0.1:9999/discoveryproxy/announcement"
          binding="netTcpBinding"
          kind="announcementEndpoint" />
      </service>
      <service name="Artech.WcfServices.Service.CalculatorService"
        behaviorConfiguration="serviceAnnouncement">
        <endpoint address="http://127.0.0.1:3721/calculatorservice"
          binding="ws2007HttpBinding"
          contract="Artech.WcfServices.Service.Interface.ICalculator" />
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceDiscovery/>
        </behavior>
        <behavior name="serviceAnnouncement">
          <serviceDiscovery>
            <announcementEndpoints>
              <endpoint kind="announcementEndpoint"
                address="net.tcp://127.0.0.1:9999/discoveryproxy/announcement"
                binding="netTcpBinding" />
            </announcementEndpoints>
          </serviceDiscovery>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

首先, 一个包含 ServiceDiscoveryBehavior 的默认服务行为被定义, 它将会自动应用到寄宿的两个服务上。对于发现代理服务 DiscoveryProxyService, 它具有两个采用 NetTcpBinding 绑定的标准终结点。其中一个地址为 “net.tcp://127.0.0.1:8888/

discoveryproxy/probe”，isSystemEndpoint 属性被设置成 False（这个设置是必需的）的 DiscoveryEndpoint 终结点。另一个则是地址为 “net.tcp://127.0.0.1:9999/discoveryproxy/announcement” 的 AnnouncementEndpoint 终结点。

至于目标服务 CalculatorService，应用了一个名称为 serviceAnnouncement 的服务行为。通过这个服务行为为它添加了一个 AnnouncementEndpoint 终结点。该终结点采用 NetTcpBinding，而地址则是发现代理服务 AnnouncementEndpoint 终结点的地址 “net.tcp://127.0.0.1:9999/discoveryproxy/announcement”。

然后通过如下一段简单的代码来同时寄宿发现代理服务 DiscoveryProxyService 和目标服务 CalculatorService。由于目标服务 CalculatorService 是在发现代理服务之后开启的，所以在它开启之后会自动向发现服务发送一个上线的通知，而发现代理在接收到通知之后会将目标服务的 EndpointDiscoveryMetadata 添加到 Endpoints 列表中。

```
using System;
using System.ServiceModel;
namespace Artech.WcfServices.Service
{
    class Program
    {
        static void Main(string[] args)
        {
            using (ServiceHost discoveryProxyService = new
                ServiceHost(typeof(DiscoveryProxyService)))
            using (ServiceHost calculatorService = new
                ServiceHost(typeof(CalculatorService)))
            {
                discoveryProxyService.Open();
                calculatorService.Open();
                Console.Read();
            }
        }
    }
}
```

步骤三：服务的动态调用

现在我们需要让客户端在不知道目标服务终结点地址的情况下进行服务的动态调用。我们直接使用 DynamicEndpoint 标准终结点。下面的 XML 片段代表客户端程序的配置，在这段配置中定义了唯一一个用于调用 CalculatorService 的 DynamicEndpoint 终结点。为了让这个 DynamicEndpoint 终结点通过请求我们寄宿的发现代理服务进行可用服务的探测，我们为它添加了一个采用 NetTcpBindg 的 DiscoveryEndpoint 终结点，该终结点的地址为 “net.tcp://127.0.0.1:8888/discoveryproxy/probe”。

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint name="calculatorService"
        kind="dynamicEndpoint"
```

```

        endpointConfiguration="unicastEndpoint"
        binding="ws2007HttpBinding"
        contract="Artech.WcfServices.Service.Interface.ICalculator"/>
</client>
<standardEndpoints>
  <dynamicEndpoint>
    <standardEndpoint name="unicastEndpoint">
      <discoveryClientSettings>
        <endpoint kind="discoveryEndpoint"
          address="net.tcp://127.0.0.1:8888/discoveryproxy/probe"
          binding="netTcpBinding"/>
      </discoveryClientSettings>
    </standardEndpoint>
  </dynamicEndpoint>
</standardEndpoints>
</system.serviceModel>
</configuration>

```

真正进行服务调用的代码和调用普通服务没有两样。完成所有的配置和编码工作之后,先后启动服务和客户端程序,你会发现客户端控制台具有如下的输出结果,表示服务调用成功完成。

```

using System;
using System.ServiceModel;
using Artech.WcfServices.Service.Interface;
namespace Artech.WcfServices.Client
{
    class Program
    {
        static void Main(string[] args)
        {
            using (ChannelFactory<ICalculator> channelFactory = new
                ChannelFactory<ICalculator>("calculatorService"))
            {
                ICalculator calculator = channelFactory.CreateChannel();
                Console.WriteLine("x + y = {2} when x = {0} and y = {1}", 1, 2,
                    calculator.Add(1, 2));
            }
            Console.Read();
        }
    }
}

```

输出结果:

x + y = 3 when x = 1 and y = 2

附录 A 实例列表

第 1 章	S101	直接在服务操作中抛出异常
	S102	异常细节的传播
	S103	通过 FaultException 直接指定错误信息
	S104	抛出 FaultException<TDetail>异常
	S105	通过 WCF 扩展实现与 EntLib 的集成
第 2 章	S201	如何通过 WsdlExporter 导出元数据
	S202	模拟 ServiceMetadataBehavior 实现基于元数据发布 (WS-MEX)
	S203	模拟 ServiceMetadataBehavior 实现基于元数据发布 (HTTP-GET)
	S204	手工获取以 WS-MEX 形式发布的元数据
	S205	手工获取以 HTTP-GET 形式发布的元数据
	S206	通过 MetadataExchangeClient 获取元数据
	S207	通过 WsdlImporter 创建终结点进行服务调用
第 3 章	S301	创建事务型服务 (服务端操作进行多次服务调用)
	S302	创建事务型服务 (客户端进行多次服务调用)
	S303	通过 TransactionFormatter 进行事务的写入
第 4 章	S401	创建 WCF 监控程序
	S402	单调实例上下文模式下的并发 (不同的客户端)
	S403	单调实例上下文模式下的并发 (相同的客户端 + 服务代理未预先开启)
	S404	单调实例上下文模式下的并发 (相同的客户端 + 服务代理预先开启)
	S405	并发模式对回调的影响 (Reentrant 并发模式)
	S406	并发模式对回调的影响 (Multiple 并发模式)
	S407	UseSynchronizationContext 的意义何在
第 5 章	S501	通过图片传输演示可靠会话的作用
第 6 章	S601	如何进行消息的批量接收处理
	S602	会话中的消息交换
	S603	让会话服务操作 TransactionScopeRequired=True 就足够了吗

续表

第 6 章	S604	会话队列服务应该如何定义
	S605	接收重试 (单批次)
	S606	接收重试 (多批次)
第 7 章	S701	创建基于 TLS/SSL 的服务 (自我寄宿 + SSL Over Tcp + 服务证书不被信任)
	S702	创建基于 TLS/SSL 的服务 (自我寄宿 + SSL Over Tcp + 改变服务证书验证方式)
	S703	创建基于 TLS/SSL 的服务 (自我寄宿 + HTTPS + 服务证书不被信任)
	S704	创建基于 TLS/SSL 的服务 (自我寄宿 + HTTPS + 改变服务证书验证方式)
	S705	创建基于 TLS/SSL 的服务 (IIS 寄宿 + HTTPS)
	S706	通过 MembershipProvider 进行用户名/密码的认证
	S707	验证安全会话是否可以避免重复认证 (关闭安全会话)
	S708	验证安全会话是否可以避免重复认证 (开启安全会话)
第 8 章	S801	基于 Windows 用户组的声明式授权
	S802	通过身份模拟的方式读取文件
	S803	在 Windows 认证下使用 ASP.NET Roles 授权
	S804	在 X.509 证书认证下使用 ASP.NET Roles 授权
	S805	通过 WCF 扩展实现授权
	S806	通过自定义 AuthorizationPolicy 和 ServiceAuthorizationManager 实现授权
	S807	基于认证的安全审核
	S808	基于授权的安全审核 (默认 AuthorizationManager)
	S809	基于授权的安全审核 (自定义 AuthorizationManager)
第 9 章	S901	通过扩展确保语言文化一致性
	S902	自定义 ServiceHost
	S903	IoC 在 Unity 中的实现
	S904	通过扩展实现基于 IoC 的服务实例的创建
第 10 章	S1001	如果使用路由服务
	S1002	如何利用服务发现机制实现服务的“动态”调用 (DiscoveryClient)
	S1003	如何利用服务发现机制实现服务的“动态”调用 (DynamicEndpoint)
	S1004	目标服务上/下线通知的发送与接收
	S1005	自定义发现代理服务

参考文献

- [1] Juval Lowy. Programming WCF Services. O'Reilly, 2007
- [2] Scott Klein. Professional WCF Programming: .NET Development with the Windows Communication Foundation. Wiley Publish Inc, 2007
- [3] Chris Peiris, Dennis Mulder, Shawn Cicoria, Amit Bahree, Nishith Pathak. Pro WCF: Practical Microsoft SOA Implementation. Apress, 2007
- [4] Steve Resnick, Richard Crane, Chris Bowen. Essential Windows Communication Foundation for .NET Framework 3.5. Pearson Education, 2008
- [5] Justin Smith. Inside Windows Communication Foundation. Microsoft Press, 2007
- [6] Thomas Erl. 王满红 陈荣华 译. SOA 概念、技术与设计. 北京: 机械工业出版社, 2007
- [7] W.Richard Stevens. 范建华 译. TCP/IP 详解 (卷 1: 协议). 北京: 机械工业出版社, 2000
- [8] Matthew MacDonald. 戢中东, 周长青, 张晔, 常小红 译. .NET 分布式应用程序: 集成 XML Web 服务与 .NET 远程处理. 北京: 清华大学出版社, 2005
- [9] MSDN WCF Library: <http://msdn.microsoft.com/en-gb/library/ms735119.aspx>
- [10] Nicholas Allen's Indigo Blog: <http://blogs.msdn.com/dnrick/default.aspx>
- [11] 张玉彬. WCF 技术博客: <http://www.cnblogs.com/jillzhang/category/121346.html>
- [12] 李会军. WCF 后传系列: <http://www.cnblogs.com/Terrylee/category/36734.html>
- [13] 张逸. WCF&SOA 博客文章系列: <http://www.cnblogs.com/wayfarer/category/24807.html>
- [14] 徐宁. Web Service & SOA 博客文章系列: <http://www.cnblogs.com/idior/category/47114.html>
- [15] Aaron Skonnard . Patterns For High Availability, Scalability, And Computing Power With Windows Azure: <http://msdn.microsoft.com/en-gb/magazine/dd727504.aspx>
- [16] Juval Lowy. Easily Apply Transactions To Services:
<http://msdn.microsoft.com/engb/magazine/2009.01.foundations.aspx>
- [17] Juval Lowy. Managing State With Durable Services:
<http://msdn.microsoft.com/engb/magazine/cc947881.aspx>

- [18] Juval Lowy. What's New for WCF in Visual Studio 2008:
<http://msdn.microsoft.com/engb/magazine/cc163289.aspx>
- [19] Juval Lowy. Synchronization Contexts in WCF:
<http://msdn.microsoft.com/en-gb/magazine/cc163321.aspx>
- [20] Introduction to IIS 7.0 Architecture:
<http://learn.iis.net/page.aspx/101/introduction-to-iis7-architecture/>
- [21] Michele Leroux Bustamante. Building a WCF Router, Part 1:
<http://msdn.microsoft.com/enus/magazine/cc500646.aspx>
- [22] Michele Leroux Bustamante. Building a WCF Router, Part 2:
<http://msdn.microsoft.com/enus/magazine/cc546553.aspx>
- [23] SOAP Version 1.2 Spec: <http://www.w3.org/TR/soap/>
- [24] Web Services Addressing 1.0 Spec: <http://www.w3.org/TR/ws-addr-core/>
- [25] J.D Meier's Blog: <http://blogs.msdn.com/jmeier/default.aspx>
- [26] Aaron Skonnard's Blog: <http://www.pluralsight.com/community/blogs/aaron/default.aspx>
- [27] Buddhike's Weblog: <http://blogs.thinktecture.com/buddhike/>
- [28] Carlos' blog: <http://blogs.msdn.com/carlosfigueira/default.aspx>
- [29] Mehran Nikoo's Notes:
http://mehranikoo.net/CS/archive/2008/05/31/WCF_5F00_Best_5F00_Practices.aspx
- [30] Matevz Gacnik's Weblog:
<http://www.request-response.com/blog/CategoryView,category,NET30WCF.aspx>
- [31] SOAP 1.1 Specification: <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>
- [32] SOAP 1.2 Specification: <http://www.w3.org/TR/soap/>
- [33] EntLib in Codeplex: <http://entlib.codeplex.com/>
- [34] WS-Policy 1.2 Specification: <http://www.w3.org/Submission/WS-Policy/>
- [35] WS-Policy 1.5 Specification: <http://www.w3.org/TR/ws-policy/>
- [36] WS-Transfer (2006.03) Specification:
<http://www.w3.org/Submission/2006/SUBM-WS-Transfer-20060315/>
- [37] WS-Transfer (2006.09) Specification: <http://www.w3.org/Submission/WS-Transfer/>
- [38] WSDL 1.1 Specification: <http://www.w3.org/TR/wsdl>
- [39] WSDL 1.2 Specification: <http://www.w3.org/TR/2003/WD-wsdl12-20030303/>
- [40] WS-MEX Specification: <http://www.w3.org/TR/ws-metadata-exchange/>

- [41] WS-Coordination 1.2 Specification:
<http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html>
- [42] WS-AT 1.1 Specification:
<http://docs.oasis-open.org/ws-tx/wstx-wsat-1.1-spec-os/wstx-wsat-1.1-spec-os.html>
- [43] WS-AT Specification:
<http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os/wstx-wsat-1.2-spec-os.html>
- [44] MSDTC in Wikipedia: http://en.wikipedia.org/wiki/Distributed_Transaction_Coordinator
- [45] MSDTC in Technet: [http://technet.microsoft.com/en-us/library/dd337629\(v=ws.10\).aspx](http://technet.microsoft.com/en-us/library/dd337629(v=ws.10).aspx)
- [46] Introducing System.Transactions in the .NET Framework 2.0:
<http://msdn.microsoft.com/en-us/library/ms973865.aspx>
- [47] [MS-DTCO]: MSDTC Connection Manager: OleTx Transaction Protocol Specification:
[http://msdn.microsoft.com/en-us/library/cc229116\(v=prot.10\).aspx](http://msdn.microsoft.com/en-us/library/cc229116(v=prot.10).aspx)
- [48] WS-RM 1.0 Specification: <http://msdn.microsoft.com/en-us/library/bb924595.aspx>
- [49] WS-RM 1.1 Specification:
<http://docs.oasis-open.org/ws-rx/wsrw/200608/wsrw-1.1-spec-cd-04.html>
- [50] WCF Reliable Messaging Demo: http://thejoyofcode.com/WCF_Reliable_Messaging_Demo.aspx
- [51] Best Practices for Reliable Sessions: <http://msdn.microsoft.com/en-us/library/ms733795.aspx>
- [52] Direct Format Names:
[http://msdn.microsoft.com/en-us/library/windows/desktop/ms700996\(v=VS.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms700996(v=VS.85).aspx)
- [53] Queues in Windows Communication Foundation:
<http://msdn.microsoft.com/en-us/library/ms731089.aspx>
- [54] WS-Security 1.0 Specification:
<http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- [55] WS-Security 1.1 Specification:
<http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>
- [56] WS-Trust 1.3 Specification: <http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html>
- [57] WS-Trust 1.4 Specification: <http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html>
- [58] WS-SecureConversation 1.3 Specification:
<http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.3/ws-secureconversation.html>
- [59] WS-SecureConversation 1.4 Specification:
<http://docs.oasis-open.org/ws-sx/ws-secureconversation/v1.4/ws-secureconversation.html>
- [60] WS-SecurityPolicy 1.2 Specification:

- <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html>
- [61] WS-SecurityPolicy 1.3 Specification: <http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200802>
- [62] Improving Web Services Security: Scenarios and Implementation Guidance for WCF:
<http://msdn.microsoft.com/en-us/library/ff650794.aspx>
- [63] Configuring HTTP and HTTPS: <http://msdn.microsoft.com/en-us/library/ms733768.aspx>
- [64] How Service Publication and Service Principal Names Work:
[http://technet.microsoft.com/en-us/library/cc755804\(v=ws.10\).aspx](http://technet.microsoft.com/en-us/library/cc755804(v=ws.10).aspx)
- [65] Mutual Authentication Using Kerberos:
[http://msdn.microsoft.com/en-us/library/ms677600\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms677600(v=vs.85).aspx)
- [66] NTLM: <http://en.wikipedia.org/wiki/NTLM>
- [67] Security Support Provider Interface: <http://en.wikipedia.org/wiki/SSPI>
- [68] Stream Upgrades, Part 1:
http://blogs.msdn.com/b/drnick/archive/2006/09/07/stream-upgrades_2c00_-part-1.aspx
- [69] Stream Upgrades, Part 2:
http://blogs.msdn.com/b/drnick/archive/2006/09/07/stream-upgrades_2c00_-part-2.aspx
- [70] Stream Upgrades, Part 3:
http://blogs.msdn.com/b/drnick/archive/2006/09/07/stream-upgrades_2c00_-part-3.aspx
- [71] User and computer accounts: [http://technet.microsoft.com/en-us/library/cc759279\(v=ws.10\).aspx](http://technet.microsoft.com/en-us/library/cc759279(v=ws.10).aspx)
- [72] ASP.NET 2.0 Security Guidelines - Impersonation/Delegation:
http://www.guidanceshare.com/wiki/ASP.NET_2.0_Security_Guidelines_-_Impersonation/Delegation#Know_Your_Tradeoffs_with_Impersonation
- [73] Configuring Many-to-One Client Certificate Mappings for IIS 7/7.5:
<http://blogs.iis.net/webtopics/archive/2010/04/27/configuring-many-to-one-client-certificate-mappings-for-iis-7-7-5.aspx>
- [74] Delegation and Impersonation with WCF: <http://msdn.microsoft.com/en-us/library/ms730088.aspx>
- [75] FindPrivateKey: <http://msdn.microsoft.com/en-us/library/aa717039.aspx>
- [76] WCF Security Guidance Project: <http://wcfsecurity.codeplex.com/>
- [77] Manual Addressing: <http://blogs.msdn.com/b/drnick/archive/2007/01/09/manual-addressing.aspx>
- [78] Discovery Announcements and Announcement Client:
<http://msdn.microsoft.com/en-us/library/ee620626.aspx>
- [79] WS-Discovery 1.1 Specification:
<http://docs.oasis-open.org/ws-dd/discovery/1.1/pr-01/wsdd-discovery-1.1-spec-pr-01.html>

定做电子书，海量电子书，
各科电子书，代寻电子书。

Q Q：1759560190

[General Information]

书名=WCF全面解析：下册

作者=蒋金楠著

页数=576

出版社=北京市：电子工业出版社

出版日期=2012.04

SS号=12981499

DX号=000007802158

URL=<http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000007802158&d=A945BC664758998D08A51C7B7E222302>