Broadview
www.broadview.com.cn

下册

WCF全面解析

◎蒋金楠 著

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

下册

WCF全面解析

◎蒋金楠 著

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

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时框架，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。

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：
装 订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 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 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

目 录

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，以及如何对 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。

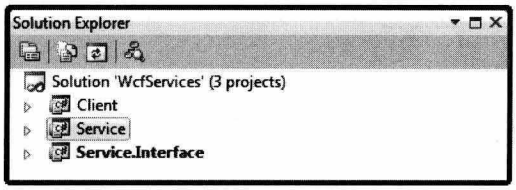

图 1-1 异常抛出实例解决方案结构

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）

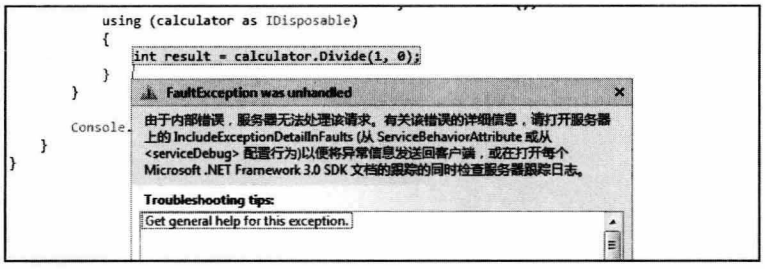

图 1-2 客户端捕获从服务端抛出的异常

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。

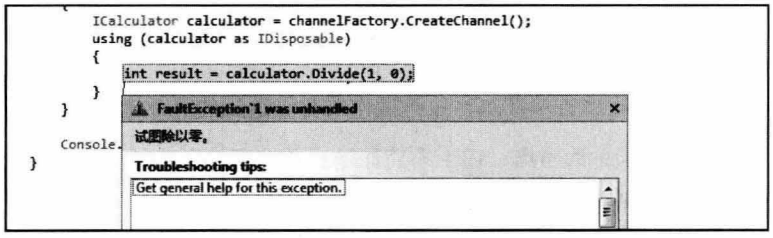

图 1-3 客户端捕获到具有细节信息的异常

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 SerivceDebugBehavior 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，SerivceDebugBehavior 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 SerivceDebugBehavior，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。