# 第 3 章

# 协　　定

WCF 通过 SOAP 消息来传递数据，数据从一端发送到另一端需要经过序列化和反序列化的过程，为了使服务器与客户端都能正确识别 SOAP 消息中传递的内容，WCF 需要一系列的协定。

协定可以为通信的双方构建一种“约定”，服务器和客户端可以重新定义用于通信的类型，但无论怎么定义，都必须遵守协定中的声明。WCF 开发中用到的协定有服务协定、操作协定、数据协定、消息协定。本章将逐一讲述以上各种协定。

## 3.1　服务协定与操作协定

前面在了解 WCF 服务的基本开发步骤时，读者已经接触过服务协定和操作协定了。从类型角度看，服务协定是一个接口，而操作协定就是接口中的一个方法。因此，服务协定包含若干个操作协定（至少需要一个方法声明为操作协定）。

下面代码演示一个简单的服务协定的声明：

```
[ServiceContract]
public interface IWorker
{
    [OperationContract]
    void Run(double k);
    [OperationContract]
    int CheckProgress();
}
```

在接口上应用 ServiceContractAttribute 表示它将作为服务协定公开，接口中要作为操作协定的方法必须应用 OperationContractAttribute，如果方法没有应用 OperationContractAttribute，那么它不会作为服务操作被公开。

再看一个例子：

```
[ServiceContract]
public interface IActions
{
    [OperationContract]
    int Sum(int[] srcs);

    float Avg(float a, float b, float c);
}
```

在上述代码中，只有 Sum 方法才会作为服务操作，而 Avg 方法是不会被服务公开的，因为它没有应用 OperationContractAttribute。

### 3.1.1 服务协定的命名空间与名称

在声明服务协定时，可以指定命名空间（Name Space）和协定名称（Name），这两个属性值都会应用于服务所公开的元数据（WSDL 文档）中。Namespace 属性将作为 WSDL 文档中 portType 元素的命名空间，如果不指定该属性的值，将使用默认值 http://tempuri.org。而 Name 属性则应用于 WSDL 文档中 portType 元素的名称。

假设有这样一个服务协定：

```
[ServiceContract]
interface IDemo
{
    [OperationContract]
    int Add(int x, int y);
}
```

在声明服务协定后，可以用下面的代码来获取协定的命名空间和名称。

```
ContractDescription ctdesc = ContractDescription.GetContract(typeof(IDemo));
string s = $"服务协定名称:{ctdesc.Name}\n";
s += $"服务协定的命名空间:{ctdesc.Namespace}";
Console.WriteLine(s);
```

调用静态的 GetContract 方法可以返回一个 ContractDescription 实例，它包含服务协定相关的信息。执行上述代码后，会输出以下内容：

```
服务协定名称: IDemo
服务协定的命名空间: http://tempuri.org/
```

从以上输出信息可以看到，如果 ServiceContractAttribute 的 Namespace 属性与 Name 属性没有被显式设置，则默认的命名空间为 http://tempuri.org/，默认的协定名称与作为服务协定的接口名称相同（示例中为 IDemo）。

接下来，再声明一个服务协定：

```
[ServiceContract(Namespace = "http://test.com", Name = "MyTask")]
interface ITest
{
    [OperationContract]
    void StartTask();
}
```

依然使用 ContractDescription 类来获取上述服务协定的信息。

```
ContractDescription cd = ContractDescription.GetContract(typeof(ITest));
Console.WriteLine( $ "服务协定名称:{cd.Name}\n 服务协定命名空间:{cd.Namespace}");
```

执行后输出以下结果：

```
服务协定名称: MyTask
服务协定命名空间: http://test.com
```

命名空间可以是任何符合 XML 规范的字符串，由于服务器的 URI 具有唯一性，通常服务协定的命名空间都会使用服务器的 URI。如果不想使用 URI 来定义命名空间，也可以使用其他字符串，例如 my-service、my_service、WorkerService 等，字符串中不要带有空格。

有关本示例的完整代码请参考\第 3 章\Example_1。

### 3.1.2 操作协定的 Action 值

由于操作协定是包含在服务协定中的，因此它不需要声明命名空间，不过，有两个属性值是应该注意的。与服务协定相似，OperationContractAttribute 类也有一个 Name 属性，用于声明操作协定的名称，默认是方法的名字。例如，

```
[ServiceContract]
interface ITest
{
    [OperationContract]
    void Run();
}
```

在上面代码所声明的服务协定中包含一个操作协定，因为 Name 属性未指定，则使用默认值——即方法的名字 Run。当然，根据需要，可以为 Name 属性设置一个字符串值。

操作协定还有一个比较重要的属性 Action，客户端在调用服务时，会把要调用的操作方法的 Action 值加入到 SOAP 消息的 Action 消息头中(作为 Header 元素的子级)，当消息发送到服务器时，服务调度程序会根据消息头 Action 的内容来寻找客户端要调用的服务操

作。也就是说，Action 属性是用来设置操作协定的标识的，调度程序通过客户端传入消息的 Action 头与服务中各个操作方法的 Action 属性值进行匹配，如果找到就调用相关的操作，如果未找到匹配项，就会发生错误。

还可以将 Action 属性的值设置为 *（星号），这表示客户端传入的任意 SOAP 消息都与该操作匹配，不管传入消息的 Action 头的内容是什么，此操作都能被调用。

与 Action 属性对应的，还有一个 ReplyAction 属性，它表示 SOAP 消息中答复消息的 Action 头的内容。通常，客户端调用服务操作后（单向通信除外），不管服务操作是否有数据要返回（包括返回类型为 void），都会向客户端发回一条 SOAP 消息（如果返回值为 void，就返回一条 body 为空的消息），这条从服务器返回的消息的 Action 头的内容就是 ReplyAction 属性所指定的值。

可以使用以下代码来获取服务协定中所包含的操作协定的信息。

```
ContractDescription cd = ContractDescription.GetContract(typeof(ITest));
foreach (OperationDescription op in cd.Operations)
{
    Console.WriteLine( $ "操作名称:{op.Name}");
    Console.WriteLine("-----------------------------------");
    foreach (MessageDescription msg in op.Messages)
    {
        Console.WriteLine( $ "Action = {msg.Action}");
    }
}
```

OperationDescription 类用于描述每个操作协定相关的信息。此处要注意，要获取操作协定的 Action 值，应当先访问其 Messages 集合。因为本示例的操作协定默认是双向通信，所以它会包含两条消息（接收的客户端请求消息和服务器回应给客户端的消息）。

上面代码运行后的输出结果如下：

```
操作名称:Run
-----------------------------------
Action = http://tempuri.org/ITest/Run
Action = http://tempuri.org/ITest/RunResponse
```

服务协定使用的是默认的命名空间，从上面输出可以看到，Action 值是由服务协定命名空间与操作名称组成。对应于 OperationContractAttribute 类，…/Run 就是 Action 属性的值，而…/ RunResponse 就是 ReplyAction 属性的值。

有关本示例的完整代码请参考\第 3 章\Example_2。

### 3.1.3 直接把服务类声明为服务协定

在 WCF 服务开发过程中，考虑到服务类不会向客户端公开，一般是先将一个接口类型

声明为服务协定，再单独用一个类来实现服务协定。比如这样：

```
[ServiceContract]
interface ITaskManager
{
    [OperationContract]
    void StartRun();
}

class TaskManagerSvr : ITaskManager
{
    public void StartRun()
    {
        // …
    }
}
```

但是，读者不妨观察一下 ServiceContractAttribute 类的声明，如下：

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface, …)]
public sealed class ServiceContractAttribute : Attribute
{
  …
}
```

从应用的 AttributeTargets 值可以得知，ServiceContractAttribute 类不仅可以应用于接口，还可以应用于类。这说明，类也是可以声明为服务协定的，当然指的是服务器上的代码，因为客户端不需要具体的实现代码，实现代码是在服务端执行的，客户端只是发送基本的调用信息就够了。

下面请看一个示例，参考源代码位于\第 3 章\Example_3。

首先，对于服务器，可以直接将服务类声明为服务协定。代码如下：

```
[ServiceContract(Namespace = "http://someone.net", Name = "demo")]
class MyService
{
    [OperationContract(Name = "add", Action = "add", ReplyAction = "addResponse")]
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

这样一来，就把定义接口和实现接口两个步骤合起来完成了，并直接让 ServiceContractAttribute 应用到 MyService 类上。作为操作协定的方法同样需要应用

OperationContractAttribute。

对于客户端，由于无须具体的执行代码，可以使用接口类型重新声明服务协定。

```
[ServiceContract(Namespace = "http://someone.net", Name = "demo")]
public interface IService
{
    [OperationContract(Name = "add", Action = "add", ReplyAction = "addResponse")]
    int Add(int a, int b);
}
```

读者需要注意的是，接口的名称可以与服务类不同，但是，其应用的 ServiceContract-Attribute 中的各个属性的值，必须与服务类上所声明的属性值一致；操作协定也如此，OperationContractAttribute 上各个属性也要与服务类中声明的一致，如 Action、Name 等。

有了 OperationContractAttribute 的约束，操作方法的名字可以与服务类不同，比如，服务类中的方法名为 Add，客户端中重新定义的操作方法的名字可以叫 Compute。

但是，方法的返回值类型必须一致，如果服务类中方法的返回值为 int，那么客户端上重新声明的操作方法的返回值类型就不能是 string。同理，方法的参数个数、类型、名字也必须一致，服务类中操作方法有两个参数，类型都是 int，那么客户端中的操作方法的参数就不能是三个，类型也不能为 double。参数的个数和类型必须统一，不过参数的名字是可以不一致的。关于参数名字的问题，随后会介绍。

### 3.1.4 约束参数的名字

通过服务协定和操作协定可以规范服务器和客户端之间的类型一致，如前面提到过的，为服务协定统一命名空间和名字，为操作协定统一操作名字和 Action、ReplyAction 的值，并且保证双方的操作方法在参数个数和类型、返回值的类型上匹配，这样通信双方就可以达成协定了。

然而，有一点很容易被忽略，那就是操作方法参数的名字，如果客户端和服务器各自定义的操作协定中，方法参数的名字不匹配，也会造成 WCF 服务调用失败。

不妨通过一个示例来说明问题，参考源代码位于\第 3 章\Example_4。

在服务端，声明一个服务协定，代码如下：

```
[ServiceContract(Namespace = "my-sample", Name = "num_sv")]
public interface IService
{
    [OperationContract(Name = "NumWork", Action = "work-req", ReplyAction = "work-
resp")]
    int Work(int x);
}
```

然后用服务类实现这个协定接口。

```
class MyService : IService
{
    public int Work(int x)
    {
        return x * x;
    }
}
```

这个服务方法的功能是将传入的 int 数值进行二次方运算，然后将结果返回。例如传入的值为 6，那么返回的值为 36。

在客户端上，也按照上述协定重新定义一个服务协定。

```
[ServiceContract(Namespace = "my-sample", Name = "num_sv")]
public interface IComputer
{
    [OperationContract(Name = "NumWork", Action = "work-req", ReplyAction = "work-
resp")]
    int Run(int k);
}
```

服务协定和操作协定已经匹配，可以尝试调用一下该服务。

```
IComputer channel = ChannelFactory<IComputer>.CreateChannel(binding, new EndpointAddress
(uri));
int result = channel.Run(10);
Console.WriteLine("计算结果:{0}", result);
((IClientChannel)channel).Close();
```

上述代码在调用服务时传入的参数值为 10，预期的计算结果为 100。然而，当上述代码执行后，返回的结果是 0。

发生错误的根源是操作方法参数的名字不匹配。WCF 在通道层是使用 SOAP 消息来传输数据的，而传入操作方法的所有参数会被序列化为 XML 内容，然后填充消息的 Body 元素，最后将整条消息发送给服务器。服务器接收到消息后，会读取 Body 元素的子元素，接着将读到的 XML 内容进行反序列化，得到各个传入参数的值，再将参数传递给服务类的实例，进而调用指定的方法(在本例中，将调用 Work 方法)。响应消息也如此，服务器将操作方法的返回值序列化，回传给客户端，客户端再将响应的内容反序列化，从而得到服务方法的返回值。

请读者细心对比一下，服务器上声明的操作方法与客户端上所声明的有什么不同。

```
// 服务器
  [OperationContract(Name = "NumWork", Action = "work-req", ReplyAction = "work-resp")]
  int Work(int x);
```

```
// 客户端
  [OperationContract(Name = "NumWork", Action = "work-req", ReplyAction = "work-resp")]
  int Run(int k);
```

操作协定的 Name、Action 和 ReplyAction 三个属性的值都匹配，唯一不匹配的就是参数的名字，在服务器上，参数的名字为 x，而在客户端上的名字则为 k。在进行 XML 序列化时，会生成以操作协定名称命名的元素（本例中为 NumWork），然后，是参数的名字作为子元素。由于服务器和客户端上参数的名字不同，客户端在序列化时生成的元素名为 k，而服务器在反序列化时要读取的元素名为 x。如此一来，服务器无法读到正确的值，参数只能保留其默认值 0。因此，0×0 的计算结果就是 0。

解决问题的方法就是让通信双方的方法参数名字相同，可以有两种方案：第一种方案是直接使用相同的参数名，即把客户端的 k 改为 x。如果不希望改参数的命名，可以在参数上应用 MessageParameterAttribute，并且让 Name 属性的值一致。

可以把两个协定做以下修改：

```
// 服务器
int Work([MessageParameter(Name = "num")]int x);

// 客户端
int Run([MessageParameter(Name = "num")]int k);
```

此种情况下，尽管参数的名字不同，但是，有了 MessageParameterAttribute 类的约束，参数在序列化和反序列化时所生成的包装参数值的元素名就统一命名为 num，这样就解决了参数名字不匹配的问题了。

再次运行示例，就能得到正确的值 100 了。

MessageParameterAttribute 类除了可以应用于输入参数外，还可以应用于输出参数和返回值。在本例中，序列化时封装返回值的 XML 元素名为 NumWorkResult（即在操作名称后面加上 Result），如果想要让操作方法的返回值被统一封装为 outValue，可以在方法上把 MessageParameterAttribute 应用到返回值上。代码如下：

```
[return: MessageParameter(Name = "outValue")]
int Work([MessageParameter(Name = "num")]int x);

[return: MessageParameter(Name = "outValue")]
int Run([MessageParameter(Name = "num")]int k);
```

由于方法的返回值无法在代码中可见，要把某个 Attribute 应用到返回值上，只能把它应用到方法上，并通过“return:”明确指定它是应用于返回值上的。

## 3.2 数据协定

WCF 应用程序通过序列化(通常是 XML 方式)技术来传递数据,对于.NET 中的基本类型(如 string、int、double、byte 等,也包括 DateTime 类型)都可以自动完成序列化,但是,对于自定义的复杂类型,应该声明为数据协定。与服务协定的作用相同,数据协定使得服务器与客户端共享相同的协定,确保双方在通信过程中都能识别自定义类型。自定义复杂类型通常是开发者声明的类或者结构。

### 3.2.1 数据协定与序列化

要让自定义类型成为数据协定,应当将 DataContractAttribute 应用于类型,而且还要把 DataMemberAttribute 应用到类型成员上(字段或属性)。

为了让读者能够更好地理解数据协定,接下来将通过一个实例来阐述。实例步骤如下:

(1) 新建一个控制台应用程序项目。

(2) 为项目添加 System.Runtime.Serialization 程序集的引用。

(3) 定义一个类,命名为 Student,代码如下:

```
class Student
{
    internal string Name { get; set; }
    internal int Age { get; set; }
    internal DateTime Birthday { get; set; }
}
```

在 Student 类上应用 DataContractAttribute。

```
[DataContract]
class Student
{
    …
}
```

虽然上面类和属性成员的可访问性为 internal,但不影响数据协定的序列化与反序列化,哪怕将类型成员声明为 private 也不会影响序列化。

默认情况下,数据协定的名字和类型名字相同,也可以明确指定数据协定的命名空间和名称。例如这样:

```
[DataContract(Namespace = "http://demo", Name = "student")]
class Student
{
```

```
    …
}
```

(4) 为类型的各个成员应用 DataMemberAttribute。

```
[DataContract(Namespace = "http://demo", Name = "student")]
class Student
{
    [DataMember(Name = "name")]
    internal string Name { get; set; }
    [DataMember(Name = "age")]
    internal int Age { get; set; }
    [DataMember(Name = "birthday")]
    internal DateTime Birthday { get; set; }
}
```

如果不设置 DataMemberAttribute 的 Name 属性的值,则序列化时就使用类型成员的名字。比如,对于 Age 属性,如果指定 Name 属性为 age,那么序列化为 XML 文档时,生成的元素会命名为 age,如果不指定 Name 属性,生成的 XML 元素就会命名为 Age。注意,数据协定中的各种命名都是区分大小写的。

(5) 要将数据协定类型实例序列化,应使用 DataContractSerializer 类。下面代码将一个 Student 实例序列化。

```
// 将序列化的内容写入内存流
MemoryStream stream = new MemoryStream();
DataContractSerializer dsz = new DataContractSerializer(typeof(Student));
Student stuobj = new Student
{
    Name = "小张",
    Age = 23,
    Birthday = new DateTime(1988, 7, 12)
};
// 序列化
dsz.WriteObject(stream, stuobj);
```

调用 WriteObject 方法就可以把类实例写入到内存流中。

(6) 下列代码将刚才写进内存流中的 XML 文档显示出来。

```
stream.Position = 0L;
XDocument xmldoc = XDocument.Load(stream);
Console.WriteLine( $ "\n 序列化后的 XML 文档:\n{xmldoc}");
```

由于把 Student 实例写入流后,流的当前指针会处于流的末尾,为了能够从流中读到数

据，必须将 Position 属性设置为 0，即将指针移到流的开始处。

Student 实例序列化后生成的 XML 文档如下：

```
<student xmlns="http://demo" xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <age>23</age>
  <birthday>1988-07-12T00:00:00</birthday>
  <name>小张</name>
</student>
```

（7）下列代码将进行反序列化，并填充新的 Student 实例。

```
// 反序列化
stream.Position = 0L;
Student scobj = (Student)dsz.ReadObject(stream);
```

ReadObject 方法可以从流中读取数据，并填充新的类型实例，最后将新创建的对象实例返回。注意 ReadObject 方法的返回类型为 Object，需要强制转换为数据协定类型。

本例输出的内容如图 3-1 所示。

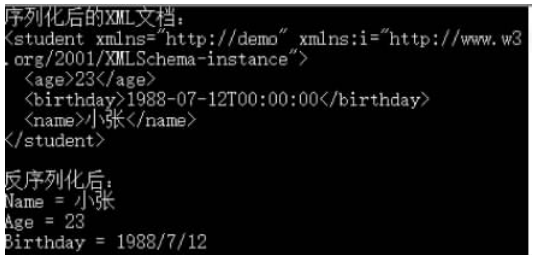


图 3-1 输出序列化后的 XML 文档

有关本示例的完整代码请参考\第 3 章\Example_5。

## 3.2.2 数据成员序列化的顺序

先看示例，下列代码声明了一个数据协定类型：

```
[DataContract]
public class Product
{
    [DataMember]
    public Guid ID { get; set; }
    [DataMember]
    public string DescName { get; set; }
    [DataMember]
    public float Size { get; set; }
}
```

该实例序列化后，会得到类似这样的 XML 文档：

```
<Product…>
  <DescName>产品 A</DescName>
  <ID>0a0c2f5c-150b-4511-b439-96736bf190b0</ID>
  <Size>5.33</Size>
</Product>
```

此时读者会发现，XML 文档中各个子元素是按照名称的首字母来排序的。如果希望让 ID 属性排在第一位，然后是 Size 属性，最后是 DescName，方法也很简单，只要在声明数据协定类型时，明确设置 DataMemberAttribute 类的 Order 属性即可。该属性是一个整数值，一般可以按顺序以连续的整数来进行设置（如 0、1、2、3），也可以不按顺序，只要排在后面的成员的 Order 比其他成员的数值要大即可（如 2、7、11）。

因此，可以对 Product 类做以下修改：

```
[DataContract]
public class Product
{
    [DataMember(Order = 3)]
    public Guid ID { get; set; }
    [DataMember(Order = 8)]
    public string DescName { get; set; }
    [DataMember(Order = 6)]
    public float Size { get; set; }
}
```

虽然数值 3、6、8 并非连续的整数，但 DescName 属性的 Order 要比 Size 属性大，所以序列化之后 DescName 属性会排在 Size 属性后面。

设置 Order 值后序列化生成的 XML 文档如下：

```
<Product…>
  <ID>ae3545d1-e6cc-4d3d-bb49-9bba4f38b0bd</ID>
  <Size>5.33</Size>
  <DescName>产品 A</DescName>
</Product>
```

有关本示例的源代码请参考\第 3 章\Example_6。

### 3.2.3 必需成员与可忽略成员

必需成员要求反序列化过程中必须找到指定的成员。一般情况下，如果 XML 数据中缺少某个成员的值，反序列化程序会为对象实例中的对应成员保留默认值（如果成员类型是 int，默认为 0；如果是 string，则默认为 null）。但是，若是某个成员被标记为必需成员，反序

列化的时候，如果找不到指定的成员的值，就会引发异常。

假设，以下类型是服务器上定义的一个数据协定：

```
[DataContract(Name = "product", Namespace = "sample-data")]
class Product
{
    [DataMember(Name = "id")]
    public int PID { get; set; }
    [DataMember(Name = "name")]
    public string Name { get; set; }
    [DataMember(Name = "color", IsRequired = true)]
    public int Color { get; set; }
}
```

以下类型是在客户端上定义的数据协定：

```
[DataContract(Name = "product", Namespace = "sample-data")]
class ProductInfo
{
    [DataMember(Name = "id")]
    public int ID { get; set;}
    [DataMember(Name = "name")]
    public string ProdName { get; set;}
}
```

尽管类型以及成员名称不同，但是它们有相同的协定信息，可以认为是相同的数据协定。读者会注意到，在服务器上定义的 product 协定存在 Color 属性，而客户端上所定义的数据协定则没有 Color 属性。而且，在服务器上，Color 属性上应用的 DataMemberAttribute 已将 IsRequired 属性设置为 true，表明该成员是必须存在的。

然后把客户端上的 product 协定实例序列化：

```
Client.ProductInfo p1 = new Client.ProductInfo
{
    ID = 1000069,
    ProdName = "测试产品"
};
DataContractSerializer sz = new DataContractSerializer(typeof(Client.ProductInfo));
sz.WriteObject(stream, p1);
```

生成的 XML 文档如下：

```
<product…>
  <id>1000069</id>
```

```
    <name>测试产品</name>
</product>
```

此时生成的 XML 文档中不存在 Color 属性，随后再用刚才的 XML 内容进行反序列化，反序列化的类型将使用服务器上定义的 product 协定。

```
DataContractSerializer dsz = new DataContractSerializer(typeof(Server.Product));
Server.Product p2 = (Server.Product)dsz.ReadObject(stream);
```

这时候会发生异常，如图 3-2 所示。

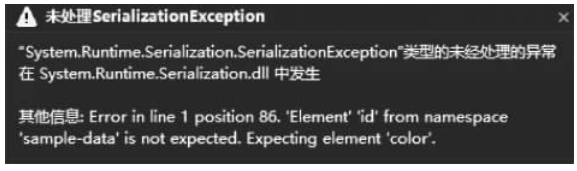


图 3-2　异常信息

异常信息中描述在 id 元素处出现不符合预期的内容，预期的元素应为 color。其实我们知道异常发生的原因是缺少了必需的 Color 属性的值，但为什么会与 ID 属性有关呢？读者不妨回忆一下，前面刚学习过的有关成员序列化顺序的内容，因为 product 协定未指定成员的排列顺序，默认是按字母排序的，即元素 color 应位于元素 id 之前，也就是说，id 元素所在的位置本来应该是 color 元素，但现在缺了 color 元素。

要使上述代码正常运行，客户端上的 product 数据协定就必须声明 color 成员。有关本示例的完整代码请参考\第 3 章\Example_7。

下面讨论另一种情形——在序列化过程中，可以忽略某些成员。在数据协定的成员上应用 IgnoreDataMemberAttribute 后，当序列化程序生成 XML 文档时，会忽略这些成员，即它们不参与序列化。

请看示例，先定义一个名为 car 的数据协定：

```
[DataContract(Namespace = "http://demo", Name = "car")]
public class Car
{
    [DataMember(Name = "speed")]
    public double Speed;
    [DataMember(Name = "color")]
    public int Color;
    [IgnoreDataMember]
    public string Remarks;
}
```

Remarks 字段上应用了 IgnoreDataMemberAttribute，因此在序列化时，它将被忽略。

下面代码将进行序列化：

```
Car car = new Car();
car.Speed = 96.3d;
car.Color = 3224;
car.Remarks = "示例文本";

DataContractSerializer sz = new DataContractSerializer(typeof(Car));
sz.WriteObject(stream, car);
```

序列化后生成的 XML 文档如下：

```
<car …>
  <color>3224</color>
  <speed>96.3</speed>
</car>
```

可以看到，Remarks 字段没有被序列化。

完整的示例代码请参考\第 3 章\Example_8。

### 3.2.4 将枚举类型声明为数据协定

如果数据协定的类型是枚举，其成员就不能应用 DataMemberAttribute，而要改用 EnumMemberAttribute。使用该特性修饰枚举的成员时，可以通过 Value 属性来设置序列化后为成员生成的 XML 元素的名字。

下列代码声明一个 Colors 枚举，其中，成员 R 表示颜色 Red，成员 G 表示颜色 green，成员 B 表示颜色 blue。

```
[DataContract(Namespace = "http://color", Name = "color")]
enum Colors
{
    [EnumMember(Value = "red")]
    R,
    [EnumMember(Value = "green")]
    G,
    [EnumMember(Value = "blue")]
    B
}
```

然后可以尝试把 Colors 枚举的实例进行序列化：

```
Colors c = Colors.B;
DataContractSerializer sz = new DataContractSerializer(typeof(Colors));
sz.WriteObject(stream, c);
```

序列化后生成的 XML 如下：

```
<color xmlns="http://color">blue</color>
```

还可以进行反序列化：

```
Colors c2 = (Colors)sz.ReadObject(stream);
```

完整的示例代码请参考\第 3 章\Example_9。

### 3.2.5 已知类型

下列代码声明了两个数据协定：

```
[DataContract]
public class Student
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public int Age { get; set; }
    [DataMember]
    public object Address { get; set; }
}

[DataContract]
public class AddressInfo
{
    [DataMember]
    public string Province { get; set; }
    [DataMember]
    public string City { get; set; }
    [DataMember]
    public string ZipCode { get; set; }
}
```

注意 Student 类的 Address 属性是 object 类型。在实例化之后，Address 属性会赋值一个 AddressInfo 实例对象。

```
Student stu = new Student();
stu.Name = "小明";
stu.Age = 22;
AddressInfo addr = new AddressInfo();
addr.Province = "山东";
```

```
addr.City = "济南";
addr.ZipCode = "250000";
stu.Address = addr;
```

接着，尝试将 Student 实例序列化。

```
DataContractSerializer sz = new DataContractSerializer(typeof(Student));
sz.WriteObject(stream, stu);
```

代码执行之后，会发生如图 3-3 所示的异常。


图 3-3 序列化程序找不到所需类型

发生异常的原因是 Student 实例进行序列化时，DataContractSerializer 组件无法识别 AddressInfo 类。在定义 Student 类的时候，Address 属性声明为 object 类型，而实例中实际引用的类型是 AddressInfo。要解决这个问题，可以向序列化提供“已知类型”。

提供已知类型有三种方法，接下来分别介绍。

第一种方法是在调用 DataContractSerializer 类构造函数时，通过一组 Type 列表（或者 Type 数组）来告诉序列化程序，应该识别哪些类型。代码如下：

```
List<Type> types = new List<Type>();
types.Add(typeof(AddressInfo));
DataContractSerializer sz = new DataContractSerializer(typeof(Student), types);
…
```

第二种方法，可以使用一个叫 DataContractSerializerSettings 的类，它可以为序列化程序设置一些参数，然后传递给 DataContractSerializer 类的构造函数。

```
DataContractSerializerSettings setting = new DataContractSerializerSettings();
setting.KnownTypes = new Type[] { typeof(AddressInfo) };
DataContractSerializer sz = new DataContractSerializer(typeof(Student), setting);
```

其中，KnownTypes 属性可以设置已知类型列表。

第三种方法是在 Student 类上应用 KnownTypeAttribute，并指明相关的 Type。

```
[DataContract]
[KnownType(typeof(AddressInfo))]
```

```
public class Student
{
…
}
```

以上三种方法，任选一种即可。设置已知类型后，就可以顺利地将 Student 实例序列化了，序列化生成的 XML 如下：

```
<Student …>
  <Address i:type = "AddressInfo">
    <City>济南</City>
    <Province>山东</Province>
    <ZipCode>250000</ZipCode>
  </Address>
  <Age>22</Age>
  <Name>小明</Name>
</Student>
```

完整的示例代码请参考\第 3 章\Example_10。

### 3.2.6 在 WCF 中使用数据协定

CLR 基础类型（如 string、int、byte 等）在序列化和反序列化的过程中都可以自动识别，因此，基础类型是无须声明为数据协定。在实际开发中，经常会用到自定义的复杂类型（一般是自定义类），如果某个服务操作的参数（或者返回值）是自定义的复杂类型，就应该将其声明为数据协定。

下列代码定义了一个用于发布新闻的服务协定：

```
[ServiceContract(Name = "news_sv", Namespace = "http://xnews.org")]
public interface IService
{
    [OperationContract(Name = "postNews", Action = "post_act", ReplyAction = "post_rpl")]
    void PostNews(News obj);
}
```

其中，PostNews 方法的输入参数是 News 类型，它是一个自定义类，应当声明为数据协定，这样有利于通信的双方共享一个“约定”，作用与服务协定类似。

```
[DataContract]
public class News
{
    [DataMember]
```

```
    public string Title { get; set; }
    [DataMember]
    public string Content { get; set; }
    [DataMember]
    public DateTime PublishDate { get; set; }
}
```

上面所声明的都是协定，协定代码既可以在服务器与客户端之间共享，也可以双方各自定义，只要协定所要求的名称、命名空间、Action 等值相同即可。在本例中，通信双方将共享同一套协定声明。

接下来，在服务器上实现服务协定，代码如下：

```
class MyService : IService
{
    public void PostNews(News obj)
    {
        string msg = $"新闻标题:{obj.Title}\n正文:{obj.Content}\n发布日期:{obj.
PublishDate:d}";
        Console.WriteLine("发布成功,新闻概要如下:\n" + msg);
    }
}
```

创建 ServiceHost，运行服务。

```
Uri svaddress = new Uri("http://localhost:2000/news");

using(ServiceHost host = new ServiceHost(typeof(MyService)))
{
    // 添加终结点
    host.AddServiceEndpoint(typeof(IService), new BasicHttpBinding(), svaddress);
    // 打开服务
    host.Open();
    Console.WriteLine("服务已运行,请等待客户端调用.");

    Console.Read();
}
```

客户端调用代码如下：

```
IService sv = ChannelFactory<IService>.CreateChannel(new BasicHttpBinding(), new
EndpointAddress(svaddress));
News n = new News
{
```

```
    Title = "示例新闻",
    Content = "学习编程是一个循序渐进的过程.",
    PublishDate = new DateTime(2016, 8, 9)
};
sv.PostNews(n);
((IClientChannel)sv).Close();
```

示例运行后,SOAP 消息中 body 元素下生成的 XML 内容如下:

```
<obj …>
  <a:Content>…</a:Content>
  <a:PublishDate>…</a:PublishDate>
  <a:Title>…</a:Title>
</obj>
```

完整的示例代码请参考\第 3 章\Example_11。

### 3.2.7 数据项的最大值

数据项是指在序列化过程中要处理的数据对象,出于服务器的安全性考虑,对于可能被大量调用的服务,都应该为数据协定设置一个数据项的最大值,以防止恶意客户端使用超大型数据来攻击服务器。

下面介绍数据协定实例在序列化过程中是如何计算数据项的,不要求读者掌握计算方法,因为设置数据项的最大值只需要一个合理的值即可,比如 200 或 100,取多大的值要视实际应用情况而定,不需要设定准确的值。

序列化过程中处理的数据项数量通常会按照以下原则来计算:

- 作为数据协定的类型,其自身的实例算作 1 个数据项;
- 基础类型(long、int、double 等),每个实例为 1 个数据项;
- 数据协定类型中,如果属性(或字段)的类型是基础类型,算作 1 个数据项;
- 如果属性(或字段)是复杂类型,则每个属性(或字段)的数据项数量就是该成员的类型实例的数据项数量。比如,属性 X 是 K 类型,而一个 K 类型的实例有 3 个数据项,那么,X 属性就占用 3 个数据项。

请考虑以下数据协定:

```
[DataContract]
public class Goods
{
    [DataMember]
    public string Barcode { get; set;}
    [DataMember]
    public int ID { get; set; }
```

```
    [DataMember]
    public decimal Price { get; set; }
    [DataMember]
    public string Name { get; set; }
}
```

Goods 类自身算作一个数据项，Goods 类有 4 个属性，而所有属性的类型是基础类型，比如，Barcode 属性是 string 类型，ID 属性是 int 类型。因此每个属性均算作一个数据项，最终，计算得知，一个 Goods 实例包含 5 个数据项。

再看一个例子：

```
[DataContract]
public class Data
{
    [DataMember]
    public long Deep { get; set; }
    [DataMember]
    public int[] Raw { get; set; }
}
```

由于类型的 Raw 属性是数组类型，即说明该数据协定包含的数据项是不确定的，具体取决于实例化的过程。下面代码将 Data 类实例化：

```
Data obj = new Data();
obj.Deep = 5000000000L;
obj.Raw = new int[] { 2, 7, 29, 105, 85 };
```

下面演示一下计算过程：

(1) Data 实例自身是一个数据项；

(2) Deep 属性是基础类型，算作一个数据项；

(3) Raw 属性是 int 数组类型，数组实例自身算作一个数据项，而数组中包含 5 个元素，int 是基础类型，故每个元素都算作一个数据项。

最终，计算出 Data 实例包含 8 个数据项。

但是，有一个类型的数据项比较特殊，那就是——byte[](字节数据组)。如果某个作为数据协定的类型中存在 byte[]类型的成员，那么它将被视为一个数据项。比如，

```
[DataContract]
public class Demo
{
    [DataMember]
    public byte[] Icon { get; set; }
```

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

然后实例化 Demo 类：

```
Demo dm = new Demo();
dm.FileName = "56.ico";
dm.Icon = new byte[] { 0x09, 0x33, 0xD1, 0x6E, 0x7F, 0x81, 0xA2, 0xCB };
```

不管 Icon 属性中有多少个字节，byte 数组都被视为一个数据项，因为字节数组在序列化时会被转化为字符串(Base64 字符串)，上面的 Demo 实例在序列化后生成的 XML 文档如下：

```
<Demo>
  <FileName>56.ico</FileName>
  <Icon>CTPRbn+Boss=</Icon>
</Demo>
```

如果一个数据协定中引用了另一个数据协定，也应该把另一个数据协定实例的数据项数量也计算进去。请考虑下面代码：

```
[DataContract]
public class AddressInfo
{
    [DataMember]
    public string Province { get; set; }
    [DataMember]
    public string City { get; set; }
}

[DataContract]
[KnownType(typeof(AddressInfo))]
public class Employee
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public int Age { get; set; }
    [DataMember]
    public AddressInfo Address { get; set; }
}
```

在上述代码中，Employee 类将作为主要的数据协定，其中，它的 Address 属性引用了

AddressInfo 协定的实例。Employee 类型中没有声明集合类型的成员，因此它的数据项数量是可以确定的，计算过程如下：

（1）Employee 实例自身算作一个数据项；

（2）Name 属性是基础类型，算作一个数据项；

（3）Age 属性是基础类型，算作一个数据项；

（4）Address 属性引用的是 AddressInfo 类型的实例，其数据项数量就是 AddressInfo 类所包含数据项数量，AddressInfo 类的 Province 属性与 City 属性都是基础类型，共 2 个数据项，AddressInfo 类自身算一个，合计为 3 个数据项。

最后算得 Employee 类在序列化过程中需要处理 6 个数据项。

至此，相信读者已经对如何计算对象的数据项有一定的了解，接下来，就要重点学习如何在 WCF 中设置序列化数据项的最大值。

下例简单演示一下如何设置序列化的最大数据量。

首先声明服务协定与数据协定，如下：

```
[ServiceContract]
public interface IDemo
{
    [OperationContract]
    void SetData(TestData[] items);
}

[DataContract]
public class TestData
{
    [DataMember]
    public double ItemA;
    [DataMember]
    public double ItemB;
    [DataMember]
    public double ItemC;
}
```

其中，服务操作 SetData 的输入参数为 TestData 类的数组，TestData 数据协定包含三个公共字段。

接下来在服务器端实现服务协定，代码如下：

```
class MyService : IDemo
{
    public void SetData(TestData[] items)
    {
        string msg = $"已收到，数组长度为：{items.Length}";
```

```
            Console.WriteLine(msg);
        }
    }
```

要限制对象序列化时产生数据项的最大值，需要在服务实现类上应用ServiceBehaviorAttribute，然后给 MaxItemsInObjectGraph 属性赋值。下面代码将设置数据项的最大值为 100：

```
[ServiceBehavior(MaxItemsInObjectGraph = 100)]
class MyService : IDemo
{
    …
}
```

随后，启动服务并侦听客户端调用：

```
using (ServiceHost host = new ServiceHost(typeof(MyService)))
{
     host.AddServiceEndpoint(typeof(IDemo), new BasicHttpBinding(), "http://localhost:
700");
    host.Open();
    …
}
```

编写客户端调用代码：

```
IDemo c = ChannelFactory<IDemo>.CreateChannel(new BasicHttpBinding(), new EndpointAddress
("http://localhost:700"));
Random rand = new Random();
List<TestData> items = new List<TestData>();
for (int i = 0; i<50; i++)
{
    TestData dt = new TestData();
    dt.ItemA = rand.NextDouble();
    dt.ItemB = rand.NextDouble();
    dt.ItemC = rand.NextDouble();
    items.Add(dt);
}
c.SetData(items.ToArray());
((IClientChannel)c).Close();
```

每个 TestData 实例需要处理 4 个数据项，上面调用中产生了 50 个 TestData 实例，总共需要 200 个数据项。由于前面代码中已为 WCF 服务设置了数据项的最大值为 100，显然，客户端发送的数据量已超出服务器的要求，所以上面的调用会引发如图 3-4 所示的

异常。

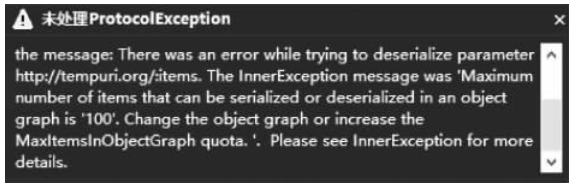


图 3-4 发生异常

现在再把 MaxItemsInObjectGraph 属性改为 300，代码如下：

```
[ServiceBehavior(MaxItemsInObjectGraph = 300)]
class MyService : IDemo
{
    …
}
```

修改后就可以正常调用 WCF 服务了。

其实在代码中设置数据项的最大值，在实际开发中不太实用，因为这个值可能会根据不同情况会被频繁改动，如果写在代码中，则每次修改后都要重新生成应用程序，显得不太方便，因此，最合理的做法是使用配置文件来进行设置，当要修改时，只需改一下配置文件就行了。比如，本示例可以在配置文件中做以下配置：

```
<configuration>
  …
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <dataContractSerializer maxItemsInObjectGraph="300"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

WCF 相关的配置都会写到 system. serviceModel 节点下，在配置文件中，是通过 dataContractSerializer 元素的 maxItemsInObjectGraph 特性来设置序列化的数据项最大值的，该配置元素最终会以 behavior 的形式应用到 WCF 服务上。有关使用配置文件配置 WCF 的内容，在后续的章节中介绍，此处只是略提一下。

读者要注意的是，使用代码配置与使用配置文件来配置，两种方案任选其一即可，不需要同时使用。

完整的示例代码请参考\第 3 章\Example_12。

### 3.2.8 版本兼容

当服务器与客户端各自定义的数据协定不一致时，就要考虑版本兼容的问题。这里所说的“不一致”，指的是数据成员在数量上不匹配的情形。如果数据协定在通信过程中存在一个往返过程，就会造成数据的丢失（缺少的成员不会进行序列化）。

所谓往返过程，就是指参与通信的 A 将数据发送给 B，而 B 在处理后又把数据发回给 A，在这个来回过程中，如果 A 和 B 上所定义的数据协定的成员不对称，就会出现数据的丢失。比如，A 上面的数据协定类有一个 City 属性，A 将数据发送给 B 时，会把 City 属性进行序列化；可是，B 上面定义的数据协定没有 City 属性。因此，当 B 把数据发回给 A 时，序列化过程中会把 City 属性的值丢弃，最终，A 收到的数据中不存在 City 属性的值，就发生了数据丢失。

读者不妨通过下面的示例来理解这一问题的具体含义。

假设，服务器和客户端分别定义了以下数据协定。

```
// 服务器
[DataContract(Namespace = "sample-data")]
public class StudentForSV
{
    [DataMember(Name = "name")]
    public string Name { get; set; }
    [DataMember(Name = "age")]
    public int Age { get; set; }
}
…

// 客户端
[DataContract(Namespace = "sample-data")]
public class StudentForCL
{
    [DataMember(Name = "name")]
    public string Name { get; set; }
}
```

尽管类的名字不同，但是它们是同一个数据协定。读者容易看出，服务器上定义的数据协定有一个 Age 属性（协定名为 age），而客户端上定义的数据协定是没有 Age 属性的。也就是说，它们虽然是同一个协定，可它们的成员数量不对等。

当服务器将数据发送给客户端时，会把 Name 和 Age 两个属性序列化，客户端收到数据后，可以进行修改，然后再发回给服务器。但是，由于客户端上的数据对象没有 Age 属性，客户端在序列化时会忽略 Age 属性，因此数据传回给服务器后，Age 属性就会被丢弃。

假如服务器将以下数据发到客户端：

```
StudentForSV stuobj = new StudentForSV();
stuobj.Name = "小赵";
stuobj.Age = 21;
return stuobj;
```

请读者先记住：服务器传到客户端的数据中，Name 属性的值是“小赵”，Age 属性的值是 21。现在，客户端会对来自服务器的数据进行编辑，把 Name 属性改为“小刘”，然后又传回给服务器。

```
// 获取来自服务器的数据
StudentForCL retobj = channel.GetStudent();
…
// 修改数据对象
retobj.Name = "小刘";
// 传回服务器
channel.SetStudent(retobj);
```

然而，数据传回服务器后，Age 属性就变成了 0(本应是 21)，如图 3-5 所示。

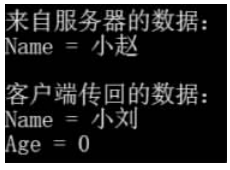


图 3-5 Age 属性的值被丢弃

int 类型的默认值是 0，这表明，Age 属性的值在发回服务器时被丢弃了。

在讲述解决方法之前，先介绍一个接口——IExtensibleDataObject。它的定义如下：

```
public interface IExtensibleDataObject
{
    ExtensionDataObject ExtensionData { get; set; }
}
```

实现这个接口很简单，只需要在实现类型中安排一个 ExtensionDataObject 类型的属性即可，开发者不必做额外的处理，WCF 运行库会自动完成其余工作。

要解决上述版本兼容性导致数据丢失，可以让数据协定类实现 IExtensibleDataObject 接口，既可以让服务器与客户端上的数据协定类都实现这个接口，也可以仅让缺少成员的数据协定类实现接口。

在本示例中，客户端的数据协定缺少了 Age 属性，因此可以将客户端上的数据协定(StudentForCL 类)做如下修改：

```
[DataContract(Namespace = "sample-data")]
public class StudentForCL : IExtensibleDataObject
{
    public ExtensionDataObject ExtensionData { get; set; }

    ...
}
```

经过以上代码的简单调整，就可以得到正确的运行结果，如图 3-6 所示。

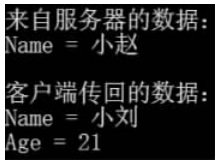


图 3-6　Age 属性可以正确回传

实现 IExtensibleDataObject 接口后，当客户端从服务器接收到数据后，会把缺少的 Age 属性的值作为额外数据存放到 ExtensionData 属性中；当该对象重新回发给服务器时，就会连同 ExtensionData 属性中缓存的内容一同发送，如此一来，服务器就可以收到完整的回传数据，Age 属性的值就不再丢失。

完整的示例代码请参考\第 3 章\Example_13。

## 3.3　使用 XML 序列化

除了使用 DataContractSerializer 类将数据对象序列化为 XML 文档外，在 WCF 中，还支持传统的 XML 序列化模型，即可以使用 XmlElementAttribute、XmlAttributeAttribute 等特性来修饰类型的成员。

首先定义一个测试类型：

```
public class AudioTrack
{
    [XmlAttribute(AttributeName = "title")]
    public string Title { get; set; }
    [XmlAttribute(AttributeName = "artist")]
    public string Artist { get; set; }
    [XmlAttribute(AttributeName = "no")]
    public int TrackNo { get; set; }
    [XmlAttribute(AttributeName = "album")]
    public string Album { get; set; }
}
```

注意该类没有定义为数据协定，而是在其成员上应用了 XmlAttributeAttribute。该特性使得类型成员被序列化为 XML 的特性值，而不是一个 XML 元素。

然后，声明服务协定，AudioTrack 类将作为服务操作的一个输入参数传递。

```
[ServiceContract]
interface IDemo
{
    [OperationContract]
    void PostMusic(AudioTrack track);
}
```

由于 WCF 默认是使用 DataContractSerializer 类来进行序列化的，如果要将参数内容以传统 XML 的方式来序列化，就要使用 XmlSerializer 类。要实现这一目的，必须在服务协定或者操作协定上应用 XmlSerializerFormatAttribute。如果特性应用到服务协定的接口上，那么该协定中的所有操作方法都会使用 XmlSerializer 类来序列化；如果特性只应用到操作方法上，那么只有这个操作方法才会使用 XmlSerializer 类来序列化，其他操作方法将使用默认的 DataContractSerializer 类来进行序列化。

对于本示例而言，只有一个操作方法，因此，XmlSerializerFormatAttribute 既可以应用到协定接口上，也可以应用到操作方法上。下面代码将 XmlSerializerFormatAttribute 应用到协定接口上：

```
[ServiceContract, XmlSerializerFormat]
interface IDemo
{
    …
}
```

下面代码测试服务调用：

```
IDemo channel = ChannelFactory<IDemo>.CreateChannel(binding, epaddr);
AudioTrack trackobj = new AudioTrack();
trackobj.TrackNo = 1;
trackobj.Title = "test song";
trackobj.Artist = "Tom";
trackobj.Album = "test album";
channel.PostMusic(trackobj);
```

通信时生成的 XML 内容如下：

```
<track title="test song" artist="Tom" no="1" album="test album"/>
```

读者会看到，AudioTrack 实例的所有属性值都放到一个 track 元素中，并且属性会映

射到 XML 元素的特性列表中。

传统 XML 序列化与数据协定序列化有一个明显的区别：数据协定可以将非公共类型进行序列化，而 XML 序列化只能作用于公共类型。读者可以尝试把 AudioTrack 类的可访问性改为 internal，如下面代码所示：

```
class AudioTrack
{
    [XmlAttribute(AttributeName = "title")]
    public string Title { get; set; }
    [XmlAttribute(AttributeName = "artist")]
    public string Artist { get; set; }
    [XmlAttribute(AttributeName = "no")]
    public int TrackNo { get; set; }
    [XmlAttribute(AttributeName = "album")]
    public string Album { get; set; }
}
```

当再次运行示例时，就会发生如图 3-7 所示的异常。


图 3-7　无法序列化非公共类型

XmlSerializerFormatAttribute 类有一个比较关键的属性——Use，如果属性设置为 Encoded，那么在序列化数据时会强制按照 SOAP 消息正文的架构规范来进行，而不管类型成员是否应用了 XmlAttributeAttribute。

比如，把本示例中的 XmlSerializerFormatAttribute 对象的 Use 属性做以下修改：

```
[ServiceContract, XmlSerializerFormat(Use = OperationFormatUse.Encoded)]
interface IDemo
{
    …
}
```

改动后再次运行示例，序列化得到的 XML 内容会变为：

```
<q2:AudioTrack id="id1" xsi:type="q2:AudioTrack" xmlns:q2="http://tempuri.org/encoded">
  <Title xsi:type="xsd:string">test song</Title>
  <Artist xsi:type="xsd:string">Tom</Artist>
  <TrackNo xsi:type="xsd:int">1</TrackNo>
```

```
    <Album xsi:type = "xsd:string"> test album </Album >
</q2:AudioTrack >
```

因此，若希望自定义的数据类型能够按照自定义的方式生成 XML 内容，就要把 Use 属性设置为 Literal。

完整的示例源代码请参考\第 3 章\Example_14。

## 3.4 消息协定

消息协定与数据协定相似，但它们的用途不同。数据协定是面向自定义类型而声明的对象及其成员的约定；消息协定则是将类型成员映射到 SOAP 消息的各个部分中，约定哪些成员会被放到消息头，哪些成员会被放到消息正文中。

下列代码声明了一个 Data 类，它包含四个属性，并且将它定义为消息协定。

```
[MessageContract]
public class Data
{
    [MessageHeader]
    public int ID { get; set; }
    [MessageHeader]
    public string Token { get; set; }
    [MessageBodyMember]
    public string Description { get; set; }
    [MessageBodyMember]
    public long ByteLen { get; set; }
}
```

要将某个类型声明为消息协定，需要在类型上应用 MessageContractAttribute，在类型成员上，应用 MessageBodyMemberAttribute 表明该成员在序列化之后会被放置在 SOAP 消息的 Body 元素下(即作为消息正文)，而应用了 MessageHeaderAttribute 的成员就会被放进 SOAP 消息的 Header 元素下(即作为消息头)。在 Data 类中，ID 属性和 Token 属性将被插入到消息头中，而 Description 属性和 ByteLen 属性将作为消息的正文部分。

消息协定允许开发者以面向对象的方式来安排 SOAP 消息的内容，从而使消息的结构更加灵活。

### 3.4.1 消息协定的基本用法

下面通过一个简单的示例来演示消息协定的使用方法，具体实现步骤如下：

(1) 新建一个控制台应用程序项目。

(2) 声明消息协定，其名称为 ProductItem。

```
[MessageContract]
public class ProductItem
{
    [MessageBodyMember]
    public string Name { get; set; }
    [MessageBodyMember]
    public string Color { get; set; }
    [MessageBodyMember]
    public int PID { get; set; }
    [MessageHeader]
    public DateTime PostTime { get; set; }
}
```

ProductItem 类的前三个属性都位于消息正文，PostTime 属性将被单独放到消息头中。

(3) 声明服务协定，代码如下：

```
[ServiceContract]
public interface IService
{
    [OperationContract]
    void NewProduct(ProductItem item);
}
```

其中，NewProduct 方法的参数类型就是刚才定义的 ProductItem 消息协定。

(4) 定义服务类，实现服务协定。

```
class MyService : IService
{
    public void NewProduct(ProductItem item)
    {
        string str = $"{nameof(ProductItem.PostTime)} = {item.PostTime:D}";
        str += $"\n{nameof(ProductItem.PID)} = {item.PID}";
        str += $"\n{nameof(ProductItem.Name)} = {item.Name}";
        str += $"\n{nameof(ProductItem.Color)} = {item.Color}";
        Console.WriteLine("收到来自客户端的数据:\n{0}", str);
    }
}
```

(5) 实例化服务主机，寄宿并运行服务。

```
ServiceHost host = new ServiceHost(typeof(MyService), new Uri("http://localhost:800"));
// 打开服务
host.Open();
Console.WriteLine("服务已启动.");
```

```
…
Console.ReadKey();
host.Close();
```

（6）调用服务。

```
EndpointAddress epaddress = new EndpointAddress("http://localhost:800");
BasicHttpBinding binding = new BasicHttpBinding();
IService svobj = ChannelFactory<IService>.CreateChannel(binding, epaddress);
ProductItem p = new ProductItem
{
    PID = 100001,
    Name = "test product",
    Color = "black",
    PostTime = DateTime.Now
};
svobj.NewProduct(p);
```

运行示例程序后，客户端所提交的 SOAP 消息如下：

```
<s:Envelope xmlns:s="……">
  <s:Header>
    <h:PostTime xmlns:h="http://tempuri.org/">2016-11-28T10:52:45.1842021+08:00</h:
PostTime>
  </s:Header>
  <s:Body>
    <ProductItem xmlns="http://tempuri.org/">
      <Color>black</Color>
      <Name>test product</Name>
      <PID>100001</PID>
    </ProductItem>
  </s:Body>
</s:Envelope>
```

其中，http://tempuri.org/是服务协定生成的默认命名空间，从上面消息可知，不管是消息头还是消息正文，当消息协定没明确指定命名空间的时候，默认情况下会引用服务协定的命名空间。

PostTime 属性已经被放到 Header 元素下，另外三个属性都位于 Body 元素下，并且包装消息正文的元素名称与消息协定的名称相同（即 ProductItem）。

由于一个消息协定代表着是整条 SOAP 消息，因此，如果操作方法使用了消息协定，那么，对于方法的输入参数，只能有一个参数，而且返回值必须是 void 类型或者消息协定类型。假设 A、B 是两个消息协定类型，那么下面的操作方法声明是可行的：

```
void TestMethod(A pa);
B TestMethod(A p);
```

而下面的声明是不允许的：

```
B TestMethod(A p, int x);
void TestMethod(string title, A p);
long TestMethod(A p);
```

每一轮通信过程中，客户端只能向服务器发送一条 SOAP 消息，而消息协定会填充整条消息。如果操作方法除了消息协定类型的参数外还带有其他参数，那其他参数将无法存储，所以，使用消息协定的方法不允许出现多个输入参数。对于返回值，要么使用 void 类型，因为 void 类型会使服务调用返回空的消息，不会影响消息协定的结构；如果返回值不是 void 类型并且输入参数是消息协定，那么返回类型也必须是消息协定，这样可保证消息在往返过程中遵守协定架构。

完整的示例代码请参考\第 3 章\Example_15。

### 3.4.2 包装元素

与数据协定类似，消息协定也可以设置独立的命名空间与包装元素的名称。前提条件是 MessageContractAttribute 的 IsWrapped 属性要设置为 true，如果该属性修改为 false，则消息协定在写入消息的 body 元素时不会有包装元素。

数据协定是一个整体，因此只需要为类型设置包装元素名称和命名空间即可，类型的成员不用设置；而消息协定类型中的成员自身是可以独立存储的，它们可能被放置到消息的 header 元素下，也可能位于 body 元素下，每个成员都可以单独包装，所以，消息协定类型的每个成员都允许设置元素名称和命名空间。

下面通过一个示例来比较一下，当 IsWrapped 属性分别为 true 和 false 的情况下，所生成的消息正文有什么不同。

首先定义两个消息协定，如下：

```
namespace V1
{
    [MessageContract(IsWrapped = true, WrapperNamespace = "demo-msgc-root", WrapperName
= "MyMessage")]
public class InfoItem
    {
        [MessageBodyMember(Namespace = "demo-data-item", Name = "stu_name")]
        public string Name { get; set; }
        [MessageBodyMember(Namespace = "demo-data-item", Name = "stu_age")]
        public int Age { get; set; }
```

```
        [MessageBodyMember(Namespace = "demo-data-item", Name = "stu_gender")]
        public byte Gender { get; set; }
    }
}

namespace V2
{
    [MessageContract(IsWrapped = false, WrapperNamespace = "demo-msgc-root", WrapperName
 = "MyMessage")]
public class InfoItem
    {
        …
    }
}
```

这两个消息协定的唯一不同是 IsWrapped 属性。V1 版本中,该属性设置为 true,而在 V2 版本中则设置为 false,这样安排是为了方便读者用于比较。

按照上面代码的定义,消息协定生成 SOAP 消息的正文后,用于包装类型成员的 XML 元素名为 MyMessage,命名空间为 demo-msgc-root。而包装类型成员的 XML 元素名称分别为 stu_name、stu_age 及 stu_gender,命名空间均为 demo-data-item。

接着定义服务协定,代码如下:

```
[ServiceContract(Namespace = "service-ct-root", Name = "DemoService")]
public interface IDemo
{
    [OperationContract(Name = "postV1", Action = "post-v1")]
    void Post_V1(V1.InfoItem pItem);
    [OperationContract(Name = "postV2", Action = "post-v2")]
    void Post_V2(V2.InfoItem pItem);
}
```

该服务协定包含两个操作方法,分别使用上面定义的两个版本的消息协定来作为输入参数。服务协定的实现代码如下:

```
internal class DemoService : IDemo
{
    public void Post_V1(V1.InfoItem pItem)
    {
        Message msg = OperationContext.Current.RequestContext.RequestMessage;
        // 获取当前调用的 Action 头内容
        string curAction = msg.Headers.Action;
        Console.Write("\n\n");
        Console.WriteLine("操作 {0} 被调用.", curAction);
```

```
        Console.WriteLine( $ "传递的信息:Name = {pItem.Name},Age = {pItem.Age},Gender =
{pItem.Gender}");
        // 输出客户端请求的 SOAP 消息
        Console.WriteLine( $ "生成 SOAP 消息:\n{msg}");
    }

    public void Post_V2(V2.InfoItem pItem)
    {
        …
    }
}
```

Post_V2 方法的处理过程与 Post_V1 方法是一样的,此处就省略了 Post_V2 方法的处理代码。以上两个操作方法主要做了以下三点:一是输出当前被调用的操作名称,二是输出参数 pItem 的信息,最后输出客户端发出的 SOAP 消息。

OperationContext.Current 是静态属性,可以获得当前正在访问的操作协定的上下文信息,随后访问 RequestContext.RequestMessage 属性就可以得到从客户端接收到的 SOAP 消息实例。

然后我们可以在客户端分别调用这两个操作,如下:

```
BasicHttpBinding binding = new BasicHttpBinding();
EndpointAddress epaddress = new EndpointAddress(svaddress);
IDemo channel = ChannelFactory<IDemo>.CreateChannel(binding, epaddress);
V1.InfoItem item1 = new V1.InfoItem
{
    Name = "Jack",
    Age = 22,
    Gender = 1
};
channel.Post_V1(item1);
V2.InfoItem item2 = new V2.InfoItem
{
    Name = "Lucy",
    Age = 20,
    Gender = 0
};
channel.Post_V2(item2);

((IClientChannel)channel).Close();
```

最后比较一下,IsWrapped 属性在被设置为不同的值的条件下,所生成的 body 元素的差异。

当 IsWrapped 属性为 true 时,消息协定的成员会有一个 MyMessage 元素来包裹,具体

的 XML 内容如下：

```
<s:Body>
  <MyMessage xmlns = "demo - msgc - root">
    <stu_age xmlns = "demo - data - item"> 22 </stu_age>
    <stu_gender xmlns = "demo - data - item"> 1 </stu_gender>
    <stu_name xmlns = "demo - data - item"> Jack </stu_name>
  </MyMessage>
</s:Body>
```

而如果 IsWrapped 属性为 false，消息协定的各个成员会直接作为 body 元素的子级来添加，而不再有 MyMessage 元素。具体的 XML 如下：

```
<s:Body>
  <stu_age xmlns = "demo - data - item"> 20 </stu_age>
  <stu_gender xmlns = "demo - data - item"> 0 </stu_gender>
  <stu_name xmlns = "demo - data - item"> Lucy </stu_name>
</s:Body>
```

通过这个示例，读者可以很直观地看到消息协定的包装元素的作用，以及 IsWrapped 属性所起到的作用。

完整的示例代码请参考\第 3 章\Example_16。

### 3.4.3 MessageParameterAttribute 与消息协定不应该同时使用

通过前面的学习，读者也了解到，MessageParameterAttribute 是用于指定服务操作协定中参数（包括返回值）的名字，在 SOAP 消息的 body 元素下，会用参数的名字来包装参数的值。而完成上一小节的示例后，读者会发现，当在 WCF 中使用了消息协定后，SOAP 消息的 body 元素下就不再使用操作方法的参数名称来命名包装元素，而是使用消息协定的名字。正因为如此，如果操作协定中使用了消息协定，就不应该再使用 MessageParameterAttribute 来修饰参数了。

下面就通过示例来看一下，不使用消息协定和已使用消息协定的操作方法所生成的 SOAP 消息的区别。本示例所定义的服务协定有两个方法，它们的功能相同——计算两个整数的和。定义代码如下：

```
[ServiceContract(Name = "demo", Namespace = "demo - root")]
public interface IDemo
{
    [OperationContract(Name = "addv1", Action = "add - v1")]
    [return: MessageParameter(Name = "resultOutput")]
    int Add_v1([MessageParameter(Name = "num1")] int m,
               [MessageParameter(Name = "num2")] int n);
```

```
    [OperationContract(Name = "addv2", Action = "add-v2")]
    OutputDataMessage Add_v2(InputDataMessage p);
}
```

Add_v1 方法使用直接参数，所以使用 MessageParameterAttribute 为各个参数指定约定名称，而 Add_v2 方法使用的是消息协定，输入参数类型为 InputDataMessage，输出参数（返回值）类型为 OutputDataMessage。这两个消息协定的定义如下：

```
[MessageContract(WrapperNamespace = "demo-data", IsWrapped = true, WrapperName = "Input_
Nums")]
public class InputDataMessage
{
    [MessageBodyMember(Name = "x", Namespace = "demo-data-input")]
    public int Number1 { get; set; }
    [MessageBodyMember(Name = "y", Namespace = "demo-data-input")]
    public int Number2 { get; set; }
}

[MessageContract(IsWrapped = true, WrapperName = "Output_Res", WrapperNamespace = "demo-
data")]
public class OutputDataMessage
{
    [MessageBodyMember(Name = "r", Namespace = "demo-data-output")]
    public int Result { get; set; }
}
```

然后实现服务协定，如下：

```
class MyService : IDemo
{
    public int Add_v1(int m, int n)
    {
        return m + n;
    }

    public OutputDataMessage Add_v2(InputDataMessage p)
    {
        OutputDataMessage output = new OutputDataMessage();
        output.Result = p.Number1 + p.Number2;
        return output;
    }
}
```

当调用 Add_v1 操作方法时，生成的消息中会用操作名称来包装输入参数，参数的元素

名称由 MessageParameterAttribute 的 Name 属性提供。调用生成的 SOAP 消息如下：

```
<s:Envelope xmlns:s = "http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header>
    <Action s:mustUnderstand = "1" xmlns = "http://schemas.microsoft.com/ws/2005/05/
addressing/none">add-v1</Action>
  </s:Header>
  <s:Body>
    <addv1 xmlns = "demo-root">
      <num1>9</num1>
      <num2>12</num2>
    </addv1>
  </s:Body>
</s:Envelope>
```

服务器返回给客户端的消息正文中，返回值的包装元素也是由 MessageParameter-Attribute 提供。返回的消息如下：

```
<s:Envelope xmlns:s = "http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header />
  <s:Body>
    <addv1Response xmlns = "demo-root">
      <resultOutput>21</resultOutput>
    </addv1Response>
  </s:Body>
</s:Envelope>
```

再观察使用了消息协定的操作方法调用，生成的请求消息如下：

```
<s:Envelope xmlns:s = "http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header>
    <Action s:mustUnderstand = "1" xmlns = "http://schemas.microsoft.com/ws/2005/05/
addressing/none">add-v2</Action>
  </s:Header>
  <s:Body>
    <Input_Nums xmlns = "demo-data">
      <x xmlns = "demo-data-input">50</x>
      <y xmlns = "demo-data-input">40</y>
    </Input_Nums>
  </s:Body>
</s:Envelope>
```

服务器返回的消息如下：

```
<s:Envelope xmlns:s = "http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header />
  <s:Body>
    <Output_Res xmlns = "demo - data">
      <r xmlns = "demo - data - output">90</r>
    </Output_Res>
  </s:Body>
</s:Envelope>
```

从以上两种情形对比可以发现，一旦使用了消息协定，操作方法的参数名字就不再起作用，就算应用了 MessageParameterAttribute 也没有实际意义，因此，MessageParameterAttribute 不应该与消息协定同时使用。

完整的示例代码请参考\第 3 章\Example_17。

### 3.4.4 数组类型与消息头

在消息协定中，如果作为消息头的成员是数组类型，那么有两种方案可供选择：一种是前面提到过的，在成员上应用 MessageHeaderAttribute；另一种方案是应用 MessageHeaderArrayAttribute。对于数组类型的成员，以上两个特性类都可以用，那么它们之间有什么区别呢？下面通过示例来进行对比。

假设下面代码所定义的消息协定表示一封电子邮件的基本信息，如下：

```
[MessageContract(WrapperName = "mail", WrapperNamespace = "mail - info")]
public class MailMessage
{
    [MessageBodyMember(Name = "subject", Namespace = "mail - info")]
    public string Subject { get; set; }
    [MessageBodyMember(Name = "body", Namespace = "mail - info")]
    public string Body { get; set; }
    [MessageHeader(Name = "attachments", Namespace = "mail - info")]
    public Attachment[] Attachments { get; set; }
}
```

其中，Attachments 属性是一个数组，元素类型为 Attachment，其定义如下：

```
[DataContract(Namespace = "attachment - data")]
public class Attachment
{
    [DataMember]
    public string FileName;
    [DataMember]
    public long FileSize;
    [DataMember]
```

```
    public string FileType;
}
```

用于测试的服务协定与服务类如下：

```
[ServiceContract(Namespace = "mail-sender-root")]
public interface IMailSender
{
    [OperationContract(Action = "post-mail")]
    void PostMail(MailMessage msg);
}

class MailSenderService : IMailSender
{
    public void PostMail(MailMessage msg)
    {
        var reqmsg = OperationContext.Current.RequestContext.RequestMessage;
        Console.WriteLine("客户端提交的消息如下:\n{0}", reqmsg);
    }
}
```

PostMail 方法对输入参数不做任何处理，仅仅输出客户端发送的 SOAP 消息。

当示例运行后，通信所产生的 SOAP 消息头如下：

```
<s:Header>
  <h:attachments xmlns:h="mail-info" xmlns:i="http://www.w3.org/2001/XMLSchema-
instance">
    <Attachment xmlns="attachment-data">
      <FileName>track.mp3</FileName>
      <FileSize>3725812</FileSize>
      <FileType>MP3</FileType>
    </Attachment>
    <Attachment xmlns="attachment-data">
      <FileName>world.jpg</FileName>
      <FileSize>137820</FileSize>
      <FileType>JPEG</FileType>
    </Attachment>
  </h:attachments>
  ...
</s:Header>
```

如果把 MailMessage 类的 Attachments 属性上的 MessageHeaderAttribute 改为 MessageHeaderArrayAttribute，则生成的 SOAP 消息头如下：

```
<s:Header>
  <h:attachments xmlns:h="mail-info" xmlns:i="http://www.w3.org/2001/XMLSchema-
instance">
    <FileName xmlns="attachment-data">track.mp3</FileName>
      <FileSize xmlns="attachment-data">3725812</FileSize>
      <FileType xmlns="attachment-data">MP3</FileType>
  </h:attachments>
  <h:attachments xmlns:h="mail-info" xmlns:i="http://www.w3.org/2001/XMLSchema-
instance">
      <FileName xmlns="attachment-data">world.jpg</FileName>
      <FileSize xmlns="attachment-data">137820</FileSize>
      <FileType xmlns="attachment-data">JPEG</FileType>
  </h:attachments>
  ...
</s:Header>
```

从上面的对比读者会发现，使用 MessageHeaderAttribute 的时候，生成的消息头中每个 Attachment 实例都会用 Attachment 元素来封装，而使用 MessageHeaderArrayAttribute 后，就会把 Attachment 元素省略，即省去了数组中每个对象的包装元素。

因此，如果要填充到消息头的成员类型是数组，不妨优先考虑使用 MessageHeaderArrayAttribute 来修饰成员，这样可以精简所生成的 SOAP 消息的长度。

完整的示例代码请参考\第 3 章\Example_18。