# 地址

在WCF中，每一个服务都有唯一的地址。地址包含两个重要的元素:服务位置及传输协议或用于服务通信的传输样式。服务位置包含目标机器名、站点（或者是网站）、通讯端口、管道（或者是队列），以及一个可选的特点路径或URI。URI可以是任务的唯一标识的字符串，如服务台名称或全局统一标示符（GUID）。

WCF支持下列传输样式：

1. HTTP/HTTPS
2. TCP
3. IPC
4. Perr network
5. MSMQ
6. Service bus

地址通信常采用以下格式：

[基地址]/[可选的URI]

基地址通常采用以下格式：

[传输协议]://[机器名或域名][:可选端口]

1. HTTP/HTTPS

<http://localhost:8080/myService/WebService.svc>

1. TCP

net.tcp://localhost:89:/myService/WebService.svc

1. IPC

net.pipe://localhost:888/mypipe

1. MSMQ

net.msmq://localhost:999/myMsmq

# 终结点EndpointAddress

WCF终结点表示服务信息，包含三个元素

1. Address：服务地址
2. Binding：服务绑定的类型
3. Contract：契约

对应的编程类为Systrm.ServiceModel.Description.ServiceEndpoint

终结点地址的描述信息又使用System.ServiceModel.EndpiontAddress,包含三个元素

1. Uri：服务的目标地址
2. Headers：包含地址报头的集合，类型为System.ServiceModel.Channels.AddressHeader
3. Identity：表示服务的身份，被客户端用于身份认证服务。

## 2.1服务端终结点地址

WCF通过ServiceHost完成对服务的寄宿而被寄宿的服务通过添加到ServiceHost的终结点暴露出来成为可悲调用寻址和调用的资源。

### 2.1.1寄宿方式

1.可用通过使用ServiceHost中AddServiceEndpoint方法手工添加终结点后寄宿。

ServiceHost继承自ServiceHostBase，ServiceHostBase中包含5个重载的AddServiceEndpoint方法，可更具不同需要调用。

2.config文件寄宿方式为在Config文件中添加如下节点。

http自寄宿

<system.serviceModel>

<services>

<service name="TaobaoServiceLibrary.TaobaoService" behaviorConfiguration="TaobaoServiceBehavior">

<endpoint address="TaobaoWCFService"

binding="basicHttpBinding"

contract="TaobaoServiceLibrary.ITaobaoService">

</endpoint>

<endpoint address="mex"

binding="mexHttpBinding"

contract="IMetadataExchange">

</endpoint>

<host>

<baseAddresses>

<add baseAddress="http://localhost:8877/"/>

</baseAddresses>

</host>

</service>

</services>

<behaviors>

<serviceBehaviors>

<behavior name="TaobaoServiceBehavior">

<serviceMetadata httpGetUrl="http://localhost:8877/wsdl" httpGetEnabled="true"/>

</behavior>

</serviceBehaviors>

</behaviors>

</system.serviceModel>

寄宿代码,添加System.ServiceModel引用

using (ServiceHost host = new ServiceHost(typeof(TaobaoServiceLibrary.TaobaoService)))

{

host.Open();

Console.ReadKey();

host.Close();

}

Msmq寄宿

<system.serviceModel>

<services>

<service name="TaobaoServiceLibrary.TaobaoMSMQService" behaviorConfiguration="TaobaoServiceBehavior">

<endpoint address="net.msmq://localhost/Private/TaobaoMSMQService"

binding="netMsmqBinding" bindingConfiguration="msmqconfig"

contract="TaobaoServiceLibrary.ITaobaoMSMQService">

</endpoint>

<endpoint address="mex"

binding="mexHttpBinding"

contract="IMetadataExchange">

</endpoint>

<host>

<baseAddresses>

<add baseAddress="http://localhost:8877/"/>

</baseAddresses>

</host>

</service>

</services>

<behaviors>

<serviceBehaviors>

<behavior name="TaobaoServiceBehavior">

<serviceMetadata httpGetUrl="http://localhost:8877/wsdl" httpGetEnabled="true"/>

</behavior>

</serviceBehaviors>

</behaviors>

<bindings>

<netMsmqBinding>

<binding name="msmqconfig">

<security mode="None"/>

</binding>

</netMsmqBinding>

</bindings>

</system.serviceModel>

寄宿代码，添加System.ServiceModel引用

if (!MessageQueue.Exists(@".\Private$\TaobaoMSMQService"))

{

MessageQueue.Create(@".\Private$\TaobaoMSMQService",true);

}

using (ServiceHost host = new ServiceHost(typeof(TaobaoServiceLibrary.TaobaoMSMQService)))

{

host.Open();

Console.ReadKey();

host.Close();

}

寄宿服务后通过ServiceHost的Description属性的Enpoints获取所有终结点信息。

IIS寄宿

与自我寄宿不同的是，IIS寄宿的方式需要为服务创建一个.svc文件，并将该.svc文件部署到一个Web应用中。由于.svc文件的地址就是服务的地址，因此无需再通过配置指定终结点 的地址。

### 2.1.2基地址与相对地址

除了以绝对路径的方式指定某个服务的终结点地址之外，还可以通过“基地址+相对地址”的方式对其进行设置。

在ServiceHost实例化时调用构造函数

ServiceHost(Type serviceType,params Uri[] baseAddresses);

ServiceHost(object singletonInstance,params Uri[] baseAddresses);

或者在config中service节点下添加

<host>

<baseAddresses>

<add baseAddress="http://localhost:8877/"/>

</baseAddresses>

</host>

WCF在进行基地址和相对地址匹配的时间，会根据终结点的绑定类型从基地址列表中获取与传输协议前缀相匹配的基地址。但是一种传输协议类型只能具有一个唯一的基地址。

### 2.1.3协议映射

对于默认终结点，WCF会使用基地址的scheme来推测绑定类型。

这个推测过程称为协议映射。加入是TCP/IPC和MSMQ,则只有一个映射选择。但是，对于HTTP/HTTPS，WCF默认使用basic绑定，则要在配置文件里使用protocolMapping节点来设置：

<protocolMapping>

<add scheme="http" binding="wsHttpBinding"/>

</protocolMapping>

# 3.元数据交换

默认情况下五福不会发布其元素，然后，这并不算排除客户端通过其他机制调用服务来获得元数据。有两种方式可发布服务的元数据：一种是基于HTTP-GET协议提供元数据，它是一种绝大多数平台都能支持简单的tetx-based协议；另一种则是使用专门的终结点方式提供元数据。

## 3.1基于HTTP-GET的元数据

在config文件中添加行为

<behaviors>

<serviceBehaviors>

<behavior name="TaobaoServiceBehavior">

<serviceMetadata httpGetUrl="http://localhost:8877/wsdl" httpGetEnabled="true"/>

</behavior>

</serviceBehaviors>

</behaviors>

## 3.2元数据交换终结点

基于HTTP-GET发布元数据仅仅是WCF的一个特性，它并不保证其他交互平台也会支持。元数据的发布具有标准的方式，既通过一个称为元数据交换终结点的特殊终结点发布。服务能够根据交换终结点发布自己的元数据。

Config添加元数据交换终结点

<endpoint address="mex"

binding="mexHttpBinding"

contract="IMetadataExchange">

</endpoint>

# 生成代理

Visual Studio中右键客户端添加服务引用，输入服务的地址后，找到服务点击确定就可以自动生成客户端调用服务的代理。

# WCF会话

在Web中为什么要会话呢？毕竟每个用户在一个Web应用中可能不止进行一次操作，比如，某二手飞机交易网站，用户A登陆后，可能他会修改他的个人信息，他也有可能看好了一架二手飞机，打算入手，就把商品放到他的“购物车”中，这些过程中，都会产生许多与用户A相关的数据，这些数据只是对A有效，而当用户B登陆后，对于B，又会有他自己的数据，总的一句话就是，**每个客户端在服务器上都有其的独立数据存储区，互不相干，就好像A和服务器在单独谈话一样，所以叫会话**。

## 5.1服务端开启会话

1. [ServiceContract(SessionMode = SessionMode.Required)]
2. public interface IService
3. {
4. 。。。。。
5. }

这种方式会影响IService实现的类的所有方法

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]

public class TaobaoService : ITaobaoService

{

…

}

这种方式只影响本实例类。

WCF默认的BascHttpBinding不支持会话，需改为wsHttpBinding

然后再配置文件中添加

<bindings>

<wsHttpBinding>

<binding name="wsconfig">

<security mode="None"/>

<reliableSession enabled="true"/>

</binding>

</wsHttpBinding>

</bindings>

## 5.1服务端方法声明关闭会话

1. [ServiceContract(SessionMode = SessionMode.Required)]
2. public interface IService
3. {
4. [OperationContract(IsOneWay = true,IsInitiating = true, IsTerminating = false)]
5. void SetValue(int n);
6. [OperationContract]
7. int GetValue();
8. [OperationContract(IsInitiating = false, IsTerminating = true)]
9. void EndSession();
10. }

a、IsInitiating：如果为真，则当调用该操作时就启用会话。

b、IsTerminating：如果为真，则说明当该用该操作时终结会话。

**传输可靠性**：指的是点对点可靠性，确保数据包按顺序到达。它不受网络等因素的影响

**消息可靠性**：指提供了消息端对端的保证，确保消息的顺序无误。如果传输失败，它还能提供重试功能自动处理网络阻塞、消息缓存、流控制，根据具体情况适时调整发送消息数。它还能通过对连接的验证管理连接自身，并在不需要的时候清除连接。

理论上，WCF服务的实现与服务契约的定义与它使用的绑定以及属性无关。实际上，服务处理消息以及契约定义都可能与消息的有序传递有关。在WCF中使用DeliveryRequirementsAttribute来对服务或者契约应用这种约束。

如果 DeliveryRequirementsAttribute 应用于契约上，则实现此种契约的服务都应用可靠的有序传递的约束。如：

[复制代码](javascript:void(0);)

    [ServiceContract]

    [DeliveryRequirements(RequireOrderedDelivery = true)]  
    public interface ICalculator1  
    {  
        [OperationContract]  
        int Add(int x, int y);  
    }

[复制代码](javascript:void(0);)

如果应用于实现某契约类型的服务，则只有此服务应用可靠的有序传递的约束。

[复制代码](javascript:void(0);)

    [DeliveryRequirements(RequireOrderedDelivery = true)]

    publicclass CalculationService : ICalculator1  
    {  
        /\*\*/  
    }

[复制代码](javascript:void(0);)

如果实现契约的服务实现多个服务契约，也可以通过DeliveryRequirements指定只有支持某种契约终结点的服务才应

用这种可靠的有序传递的约束。如下：

[DeliveryRequirements(RequireOrderedDelivery = true, TargetContract = typeof (ICalculator1))]  
public class CalculationService : ICalculator1, ICalculator2  
{      
        /\*\*/  
}

RequireOrderedDelivery默认为False，如果在接口或者服务类上使用Delivery却不设置,RequireOrderedDelivery

则也是不会开启有序传递的。

# 服务契约

## 6.1操作重载

WCF服务支持核心的Web服务协议，同样其元数据交换也是基于XML语言描述，客户端通过WSDL文件来了解服务方法相关的信息，包括参数的个数，类型，返回值，调用顺序等重要信息。由于WSDL不支持方法的重载，因此我们的WCF服务操作重载就无法通过WSDL暴露给客户端。如果我们在服务契约里定义了方法的重载，编译可以正常通过，但是启动服务宿主就会抛出System.InvaidOperationException异常。

WCF给我们提供了一个解决办法，让我们可以在WCF服务类里使用服务操作的重载。WCF定义了一个机制OperationContract，使用OperationContract特性的Name属性，为操作指定别名：

[AttributeUsage(AttributeTargets.Method)]  
public sealed class OperationContractAttribute : Attribute  
{  
   public string Name  
   {get;set;}  
  
   //更多成员  
  
}

[ServiceContract]

public interface IMyService

{

[OperationContract(Name = "AddInt")]

double Add(int a, int b);

[OperationContract(Name = "AddDouble")]

double Add(double a, double b);

// TODO: 在此添加您的服务操作

}

## 6.2契约继承

WCF支持服务契约之间的继承，但除ServiceContract除外。

[ServiceContract]

public interface IAddService

{

[OperationContract]

int Add(int a, int b);

}

public class CaculatorService:ICaculatorService

{

public double TriangleArea(double heigh, double width)

{

throw new NotImplementedException();

}

public int Add(int a, int b)

{

throw new NotImplementedException();

}

}

客户端导入一个服务终结点的元数据时，如果该终结点的契约属于接口层级的一部分，则生成的客户端契约将不再维持原来的层级关系。相反，它会取消层级，组成一个单独的契约，与终结点的契约名同名。这个单独的契约包含了层级中从上至下所有接口定义的操作。然而，如果使用OperationContract特性中的Action与ReplyAction属性，那么导入的接口定义仍然可以保留原来定义的每个操作的契约名。

[ServiceContract]

public interface ICaculatorService:IAddService

{

[OperationContract(Action = ".../ICaculatorService/TriangleArea",

ReplyAction = ".../ICaculatorService/TriangleAreaResponse")]

double TriangleArea(double heigh, double width);

}

# 数据契约

在wcF中，在服务契约定义了可供调用的服务操作方法，而数据契约则是定义l 服务端和客户端之间传输的自定义类型，在WCF项目中，不必可少地是传递数据，吧客户端需要传递的数据传送到服务中，服务接收到数据再对其进行处理。然而在WCF中，传递的类型必须标记为DataContractAttribute属性，且只有标记了DataMemberAttribute属性的属性才会被传送。

## 7.1序列化

WCF的实现原理沿用了[.NET Remoting](http://www.cnblogs.com/zhili/p/NETRemoting.html" \t "_blank)的实现机制，客户端在调用服务公开的服务方法，这个过程必然涉及到数据的传输过程，包括客户端传输相关需要处理的数据给服务或服务传输相关处理后的结果数据给客户端。在数据传输的过程中，自然就需要进行序列化的操作，通过序列化把.NET Object序列化成可保存或传输的形式，然后通过网络协议在网络上进行传递。对于序列化的实现是由序列化器（Serializer）来负责完成的，序列化的实现原理可以理解为通过反射机制分析程序集中对应的类型，然后把对应的类型映射为一个XML的结构。

　　序列化在.NET Framework相关专题就有所介绍，所以它并不是一个新的概念，相关内容可以参考MSDN：[序列化](http://msdn.microsoft.com/zh-cn/library/7ay27kt9(v=vs.100).aspx" \t "_blank)。然而.NET本身的序列化机制在WCF程序中并不适应，所以WCF又提出了新的序列化器。下面分别介绍下.NET 序列化机制和WCF中序列化机制。

## 7.2.NET序列化

在.NET Framework 3.0之前，提供了3中序列化器，序列化器理解为把可序列化的类型序列化成XML的类。这三种序列化器分别是[BinaryFormatter](http://msdn.microsoft.com/zh-cn/library/system.runtime.serialization.formatters.binary.binaryformatter(v=vs.110).aspx" \t "_blank)、[SoapFormatter](http://msdn.microsoft.com/zh-cn/library/system.runtime.serialization.formatters.soap.soapformatter(v=vs.110).aspx" \t "_blank)和[XmlSerializer](http://msdn.microsoft.com/zh-cn/library/system.xml.serialization.xmlserializer(v=vs.110).aspx" \t "_blank)类。下面分别介绍下这3种序列化器。

* BinaryFormatter类：把.NET Object序列化成二进制格式。在这个过程，对象的**公共字段和私有字段**以及类名称（包括类的程序集名），将转换成成字节流。
* SoapFormatter类：把.NET Object序列化成SOAP格式，SOAP是一种轻量、简单的，基于XML的协议。只序列化字段，包括**公共字段和私有字段**。
* XmlSerializer类：该类仅仅序列化**公共字段和属性，**且不保存类型的保真度。

　　对于这三种序列化机制，BinaryFormatter二进制序列化的优点是：性能高，但是不能跨平台。而SoapFormatter，XmlSerializer的优点是：跨平台、互操作性好，并且可读性强，但是传输性能不及BinaryFormatter。

　　在.NET原有的序列化机制中，BinaryFormatter和SoapFormatter除了要序列化对象的状态信息外，还会将程序集和版本信息持久化到流中，因为只有这样才能保证对象呗反序列为正确的对象类型副本，这就要求客户端必须拥有原有的.NET 程序集，不能满足跨平台的需求。所以WCF不得不定义自己的序列化机制来满足面向服务的需求。

## 7.3WCF序列化

在WCF中，提供了专门用来序列化和反序列操作的类，该类就是[DataContractSerializer](http://msdn.microsoft.com/zh-cn/library/system.runtime.serialization.datacontractserializer(v=vs.110).aspx" \t "_blank)类。一般而言，WCF会自动选择使用DataContractSerializer来对可序列话数据契约进行序列化，不需要开发者直接调用。WCF除了支持DataContractSerializer类来进行序列化外，还支持另外两种序列化器，这两种序列化器分别为：XMLSerializer（定义在System.XML.Serialization namespace）和[NetDataContractSerializer](http://msdn.microsoft.com/zh-cn/library/system.runtime.serialization.netdatacontractserializer(v=vs.110).aspx" \t "_blank) (定义在System.XML.Serialization namespace)。XmlSerializer类不是WCF专用的类，Asp.net Web服务统一使用该类作为序列化器，但XmlSerializer类支持的类少于DataContractSerializer列支持的类型，但它允许对生成的XML进行更多的控制，并且支持更多的XML架构定义语言（XSD）标准。它不需要在可序列化类上有任何声明性的属性。

　　默认情况下，WCF 使用 [DataContractSerializer](http://msdn.microsoft.com/zh-cn/library/system.runtime.serialization.datacontractserializer(v=vs.110).aspx) 类来序列化数据类型。 此序列化程序支持下列类型：

* 基元类型（如：整数、字符串和字节数组）以及某些特殊类型（如 [XmlElement](http://msdn.microsoft.com/zh-cn/library/system.xml.xmlelement(v=vs.110).aspx) 和 [DateTime](http://msdn.microsoft.com/zh-cn/library/system.datetime(v=vs.110).aspx)）。
* 数据协定类型（用 [DataContractAttribute](http://msdn.microsoft.com/zh-cn/library/system.runtime.serialization.datacontractattribute(v=vs.110).aspx) 属性标记的类型）。
* 用 [SerializableAttribute](http://msdn.microsoft.com/zh-cn/library/system.serializableattribute(v=vs.110).aspx) 属性标记的类型，包括实现 [ISerializable](http://msdn.microsoft.com/zh-cn/library/system.runtime.serialization.iserializable(v=vs.110).aspx) 接口的类型。
* 实现 [IXmlSerializable](http://msdn.microsoft.com/zh-cn/library/system.xml.serialization.ixmlserializable(v=vs.110).aspx) 接口的类型。
* 许多常见集合类型，包括许多泛型集合类型。

　　DataContractSerializer类与NetDataContractSerializer类类似，它们之间主要的区别在于：在使用NetDataContractSerializer进行序列化时，不需要指定序列化的类型，如：

public class UserInfo

{

public int Id { get; set; }

public string Name { get; set; }

public string PhoneNumber { get; set; }

}

UserInfo user = new UserInfo();

user.Id = 1;

user.Name = "show";

user.PhoneNumber = "110";

DataContractSerializer serializer = new DataContractSerializer(typeof(UserInfo));

using (Stream stream = new MemoryStream())

{

//序列化

serializer.WriteObject(stream, user);

//反序列化

stream.Position = 0;

serializer.ReadObject(stream);

}

## 7.4数据契约特性

如果只是在类或者结构类型上应用DataContract特性，WCF就不会序列化任务类型成员

添加DataMember特性的字段会被序列化。

[DataContract]

public class UserInfo

{

[DataMember(Name = "UserId")]//设置别名

public int Id { get; set; }

[DataMember]

public string Name { get; set; }

[DataMember]

public string PhoneNumber { get; set; }

}

对于SerializableAttribute属性的作用与DataContract的作用是一样的，都是标记为该类支持序列化。因为在默认情况下，用户定义的类型并不支持序列化，只有应用了SerializableAttribute或DataContractAttribute属性的，.NET序列化器才能对该类型进行序列化。然而这两者又存在不同， Serializable要求它的所有程序都要支持序列化，如果发现不支持序列化的成员就会抛出异常，即Serializable会把类型的所有成员都进行序列化，如果想某个成员不序列化化，则必须显式标记NoSerialized属性；而DataContract却不同，标记了DataContract属性的类只有标记了DataMember的成员才会被序列化，如果想类型的成员能够序列化，则应该应用DataMember属性。如果某个类型同时应用了DataContract和Serialized属性，此时该类型将会只应用DataContract，即Serialized属性会忽略。

## 7.5数据契约序列化事件

.NET为可序列化类型引入了对序列化事件的支持，WCF沿袭了这一技术，对数据契约提供了同样的支持。执行序列化和反序列化时，WF会调用数据契约的指定方法。WCF一共定义了4个序列化与反序列化事件。

1. OnSerializing：事件发生在序列化之前
2. OnSerialized：事件发生在序列化之后
3. OnDeserializing：事件发生在反序列化之前
4. OnDeserialized：事件发生在反序列化之后

[DataContract]

public class UserInfo

{

[OnSerializing]

public void OnSerializing(StreamingContext context)

{

}

[OnSerialized]

public void OnSerialized(StreamingContext context)

{

}

[OnDeserializing]

public void OnDeSerializing(StreamingContext context)

{

}

[OnDeserialized]

public void OnDeSerialized(StreamingContext context)

{

}

[DataMember(Name = "UserId")]//设置别名

public int Id { get; set; }

[DataMember]

public string Name { get; set; }

[DataMember]

public string PhoneNumber { get; set; }

}

注意：DataContract的类不会调用默认构造函数

## 7.6共享数据契约

当客户端引用WCF服务时，必须为服务定义一个唯一的命名空间。在默认情况下，这个服务中所用到的类会被导入到这个命名空间内。这样，在不同的服务内引用同一个数据契约时就会遇到一个问题：在客户端，同一个数据契约被导入到不同的命名空间中，无法实现转换。

在以下例子中，PersonService与ContractService同时包括对Person数据契约的引用，但通过PersonService获取到的Person对象，无法作为ContractService中的参数，因为在客户端，它们被引用到不同的命名空间之中，被视为不同的两个类。

[ServiceContract]

public interface IPersonService

{

[OperationContract]

List<Person> GetPerson();

}

public class PersonService : IPersonService

{

public List<Person> GetPerson()

{

throw new NotImplementedException();

}

}

[ServiceContract]

public interface IContractService

{

[OperationContract]

int GetContractIdByPerson(Person person);

}

public class ContractService : IContractService

{

public int GetContractIdByPerson(Person person)

{

throw new NotImplementedException();

}

}

[DataContract]

public class Person

{

[DataMember]

public int Id { get; set; }

[DataMember]

public string Address { get; set; }

[DataMember]

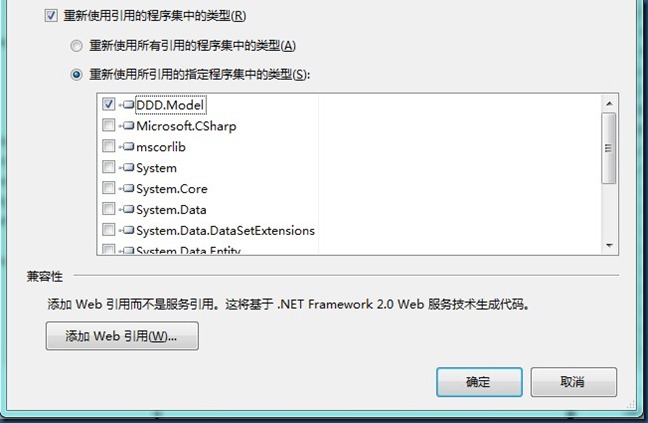
public int Age { get; set; }

[DataMember]

public string Name { get; set; }

}

从Visual Studio 2008开始，系统为客户提供了“共享数据契约”这一功能。首先把需要共享的数据契约包含在一个程序集中，生成一个Model.dll文件。在客户端引用这一程序集，在添加服务引用时，选择"高级"—>"重新使用引用的程序集中的类型（R）"—>"重新使用所引用的指定程序集中的类型（S）"，然后选择Model。这样，在客户端就会同时使用Model.dll中的Person对象。

[](http://images.cnblogs.com/cnblogs_com/leslies2/201107/201107232144409563.jpg)

## 7.7已知类型

C#语言允许开发者用子类替换基类，WCF却不支持这样的操作。默认情况下，不能用护士节契约的子类去替换基类。

[DataContract]

public class Person

{

[DataMember]

public int Id { get; set; }

[DataMember]

public string Address { get; set; }

[DataMember]

public int Age { get; set; }

[DataMember]

public string Name { get; set; }

}

[DataContract]

public class Student:Person

{

public int StudentId { get; set; }

}

[ServiceContract]

public interface IContractService

{

[OperationContract]

int GetContractIdByPerson(Person person);

}

public class ContractService : IContractService

{

public int GetContractIdByPerson(Person person)

{

return person.Id;

}

}

在VS2013 SP3版本中客户端引用服务后无法找到Student类，代理没有生成该类。

1.指定子类

[KnownType(typeof(Student))]

[DataContract]

public class Person

{

[DataMember]

public int Id { get; set; }

[DataMember]

public string Address { get; set; }

[DataMember]

public int Age { get; set; }

[DataMember]

public string Name { get; set; }

}

2.指定服务契约

public class ContractService : IContractService

{

[ServiceKnownType(typeof(Student))]

public int GetContractIdByPerson(Person person)

{

return person.Id;

}

}

注意：通过同事应用KnowType和ServiceKnowType特性，可以重复通知WCF需要多个已知类型。

## 7.8配置已知类型

已知类型的相关特性存在的主要缺陷是它们要求服务或者客户端必须事先知道那些子类可能会被其他调用方调用。添加一个新的子类是必须修改代码，重新编译和重新部署。为解决这一问题，WCF运行开发者在服务或客户端的配置文件中配置已知类型。我们不仅需要提供类型名，而且还要提供所在程序集名。

<system.runtime.serialization>

<dataContractSerializer>

<declaredTypes>

<add type="类名,程序集名,Version=1.0.0.0,Culture=neutral,PublicKeyToken=null">

<knownType type="类名,程序集名,Version=1.0.0.0,Culture=neutral,PublicKeyToken=null"/>

</add>

</declaredTypes>

</dataContractSerializer>

</system.runtime.serialization>

如果不依赖于程序集名或版本则可以使用程序集的友好名称

<add type="类名,程序集名">

<knownType type="类名,程序集名"/>

</add>

## 7.9枚举

枚举类型默认支持序列化，所以不必应用DataContract特性就可以在数据契约中使用。

如果要将确定的枚举值排除与数据契约之外，首先就需要为枚举类型标记DatContract特性。然后，再将那些希望包含在枚举数据契约中的枚举值，明确地标记为EnumMemberAttribute特性。

[DataContract]

public enum ContractType

{

[EnumMember]

Customer,

[EnumMember]

Vendor,

Parnter//不会成为数据契约的一部分

}

## 7.10集合

在.NET中，各种类型的集合均实现了IEnumerable或IEnumerable<T>接口。所有内建的.NET集合，如数组，列表和栈都实现这些接口。一个数据契约的数据成员可以是一个集合类型，服务契约也可以定义直接与集合交互的操作，这是因为.NET集合是.NET特有的，WCF不能再服务元数据中公开它们。因为集合类型非常有用，所以WCF专门为集合提供了编组原则。定义服务操作时，不管使用下列哪一种集合接口IEnumerable<T>,IList<T>,ICollection<T>,它们的传输型表示形式都使用了数组。

对于没有实现IEnumerable<T>,IList<T>,ICollection<T>接口的数据，使用CollectionContract特性来声明，序列化时则会使用集合类型。

注意：要改变服务端生成集合的类型，在引用的WCF服务中配置服务设置中修改。

