[老老实实学WCF] 第四篇 初探通信--ChannelFactory

老老实实学WCF

第四篇 初探通信--ChannelFactory

通过前几篇的学习，我们简单了解了WCF的服务端-客户端模型，可以建立一个简单的WCF通信程序，并且可以把我们的服务寄宿在IIS中了。我们不禁感叹WCF模型的简单，寥寥数行代码和配置，就可以把通信建立起来。然而，仔细品味一下，这里面仍有许多疑点：服务器是如何建起服务的？我们在客户端调用一个操作后发生了什么？元数据到底是什么东西？等等。我们现在对WCF的理解应该还处于初级阶段，我们就会觉得有许多这样的谜团了。

虽然我们生活在WCF为我们构建的美好的应用层空间中，但是对于任何一项技术，我们都应力求做到知其所以然，对于底层知识的了解有助于我们更好的理解上层应用，因此在刚开始学习入门的时候，慢一点、细一点，我觉得是很有好处的。

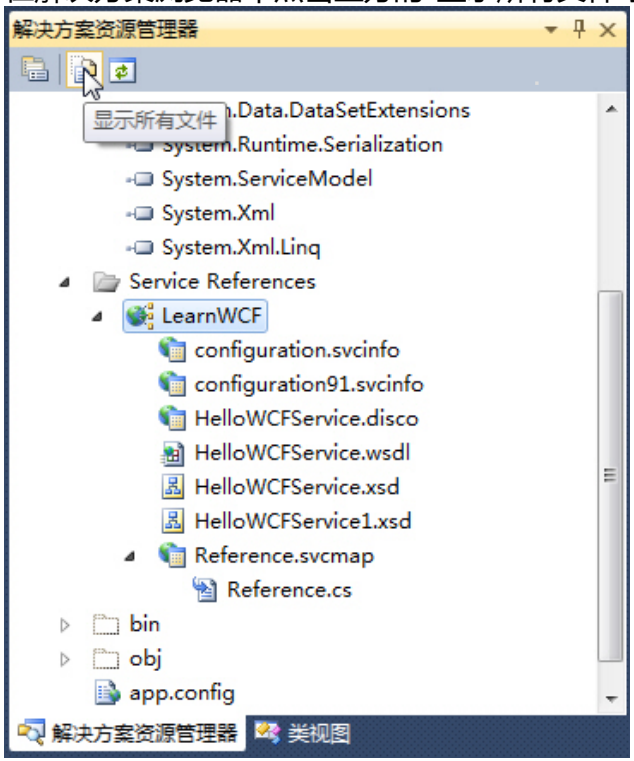
言归正传，我们现在已经知道了一件最基本的事情，客户端和服务器是要进行通信的。那么这个通信是如何发生的呢？根据我们前面的学习，从实际操作上看，我们在服务端定义好协定和实现，配置好公开的终结点，打开元数据交换，在客户端添加服务引用，然后就直接new出来一个叫做XXXClient 的对象，这个对象拥有服务协定里的所有方法，直接调用就可以了。仔细想想？天哪，这一切是怎么发生的？！

服务端定义协定和实现并公开终结点，这看上去没什么问题，虽然我们对底层的实现不了解，但总归是合乎逻辑的，而客户端怎么就通过一个添加服务引用就搞定一切了呢？似乎秘密在这个添加的服务引用中。

打开第二篇中我们建立的客户端(如果你为第三篇的IIS服务建立了客户端，打开这个也行，我用的就是这个)，看看服务引用里面有什么。

1. 服务引用初探

在解决方案浏览器中点击上方的"显示所有文件"按钮，然后展开服务引用。



这么一大堆，有一些xsd文件我们可能知道是框架描述的文档，那wsdl什么的是什么，还有disco(迪斯科？)是什么，一头雾水。
其中有一个cs文件，这个想必我们应该看得懂，打开来看看

```csharp
//------------------------------------------------------------------------------
// <auto-generated>
//     此代码由工具生成。
//     运行时版本:4.0.30319.261
//
//     对此文件的更改可能会导致不正确的行为，并且如果
//     重新生成代码，这些更改将会丢失。
// </auto-generated>
//------------------------------------------------------------------------------
```

```
10.
11. namespace ConsoleClient.LearnWCF {
12.
13.
14.     [System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "4.0.0.0")]
15.     [System.ServiceModel.ServiceContractAttribute(ConfigurationName="LearnWCF.IHelloWCF")]
16.     public interface IHelloWCF {
17.
18.         [System.ServiceModel.OperationContractAttribute(Action="http://tempuri.org/IHelloWCF/Hell
19.         string HelloWCF();
20.     }
21.
22.     [System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "4.0.0.0")]
23.     public interface IHelloWCFChannel : ConsoleClient.LearnWCF.IHelloWCF, System.ServiceModel.I
24.     }
25.
26.     [System.Diagnostics.DebuggerStepThroughAttribute()]
27.     [System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel", "4.0.0.0")]
28.     public partial class HelloWCFClient : System.ServiceModel.ClientBase<ConsoleClient.LearnWCF
29.
30.         public HelloWCFClient() {
31.         }
32.
33.         public HelloWCFClient(string endpointConfigurationName) :
34.                 base(endpointConfigurationName) {
35.         }
36.
37.         public HelloWCFClient(string endpointConfigurationName, string remoteAddress) :
38.                 base(endpointConfigurationName, remoteAddress) {
39.         }
40.
41.         public HelloWCFClient(string endpointConfigurationName, System.ServiceModel.EndpointAdd
42.                 base(endpointConfigurationName, remoteAddress) {
43.         }
44.
45.         public HelloWCFClient(System.ServiceModel.Channels.Binding binding, System.ServiceModel.
46.                 base(binding, remoteAddress) {
47.         }
48.
49.         public string HelloWCF() {
50.             return base.Channel.HelloWCF();
```

```
51.         }
52.     }
53. }
```

这么一堆代码，都不是我们写的，初步看似乎有几个类/接口，IHelloWCF，这个应该是服务协定，估计可能是从服务端下载来的，HelloWCFClient，这个就是我们的客户端代理嘛，我们在前面用过，原来是在这里定义的，可是后面继承的ClientBase<>是干嘛用的，还重载了这么多的构造函数。还有一个IHelloWCFChannel接口，我们找遍解决方案也找不到什么地方用到他了啊，干嘛在这里定义出来呢？
先不去细想这些代码的具体意义，看到这里，我们在对VS2010由衷赞叹的同时，不由得心中升起一丝忧虑，如果没有了VS2010，没有了IDE，没有了"添加服务引用"，我们该怎么办？

2. 我们自己写通信

虽然我们料想VS2010也不会消失，我们是可以一直享受它提供给我们的便利的。但是我们今天在这里研究，不妨把控制级别向下探一个层次。看看下面有什么。

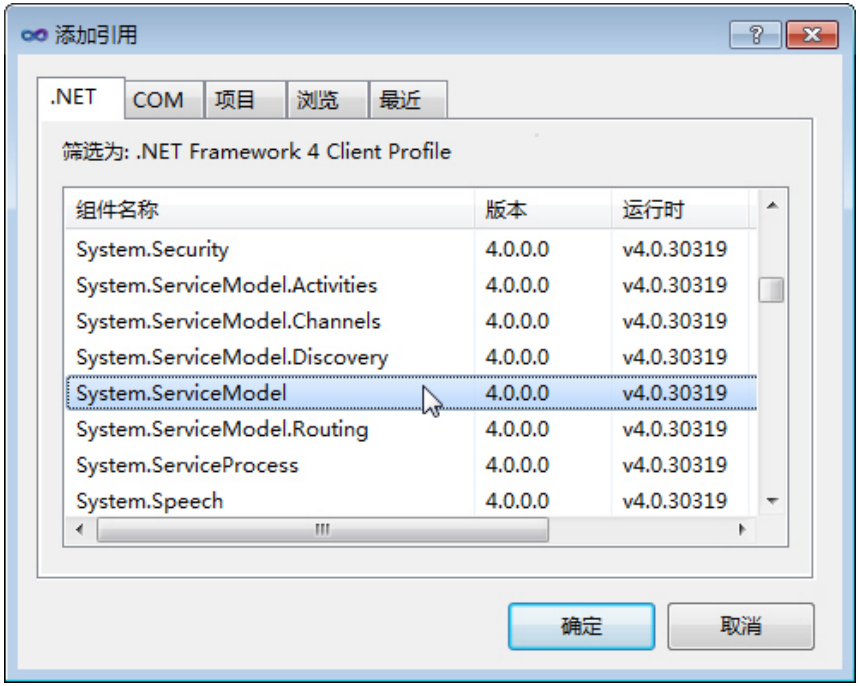
通信到底是怎么发生的？简单说就是两个终结点一个通道，实际上客户端也是有一个终结点的，客户端会在这两个终结点之间建立一个通道，然后把对服务端服务的调用封装成消息沿通道送出，服务器端获得消息后在服务器端建立服务对象，然后执行操作，将返回值再封装成消息发给客户端。
过程大概是这样的，有些地方可能不太严谨，但是这个逻辑我们是可以理解的。如此看来，通讯的工作主要部分都在客户端这边，他要建立通道、发送消息，服务端基本上在等待请求。

客户端需要哪些东西才能完成这一系列的操作呢？元数据和一些服务的类。服务类由System.ServiceModel类库提供了，只差元数据。提到元数据我们不禁倒吸一口凉气，难道是那一堆XSD OOXX的东西么？我觉得，是也不是。就我们这个例子来说，元数据包括：服务协定、服务端终结点地址和绑定。对，就这么多。我们是不是一定要通过元数据交换下载去服务端获取元数据呢，当然不是，就这个例子来说，服务端是我们设计的，这三方面的元数据我们当然是了然于胸的。

所以，让服务引用见鬼去吧，我们自己来。

(1) 建立客户端。
这个过程我们很熟悉，建立一个控制台应用程序，不做任何其他事，仅有清清爽爽的program.cs

(2) 添加必要的引用
前面说过，客户端完成通信的发起需要一些服务类的支持，这些类都定义在System.ServiceModel中，因此我们要添加这个程序集的引用。(注意是添加引用，不是服务引用)。



然后在Program.cs中using这个命名空间

```
[csharp]
1. using System.ServiceModel;
```

(2) 编写服务协定
服务协定是元数据中最重要的部分(还可能有数据协定等)，协定接口是服务器和客户端共同持有的，客户端依靠协定来创建通道，然后在通道上调用协定的方法，方法的实现，客户端是不知道的。客户端只知道方法签名和返回值(即接口)。
我们把在服务端定义的服务协定原封不动的照搬过来，注意，只把接口搬过来，不要把实现也搬过来，那是服务端才能拥有的。
服务协定我们都很熟悉了，背着打出来吧。就写在Program类的后面

```
[csharp]
```

```
[ServiceContract]
public interface IHelloWCF
{
    [OperationContract]
    string HelloWCF();
}
```

OK，元数据的第一部分完成了，另外两部分我们在代码里面提供。

(3) 通道工厂登场
System.ServiceModel提供了一个名为ChannelFactory<>的类，他接受服务协定接口作为泛型参数，这样new出来的实例叫做服务协定XXX的通道工厂。顾名思义了，这个工厂专门生产通道，这个通道就是架设在服务器终结点和客户端终结点之间的通信通道了。由于这个通道是用服务协定来建立的，所以就可以在这个通道上调用这个服务协定的操作了。
这个通道工厂类的构造函数接受一些重载参数，使用这些参数向通道工厂提供服务端终结点的信息，包括地址和绑定，这正是元数据的其他两部分。我们先把这两样做好预备着。

地址，也可以称终结点地址，实际上就是个URI了，我们也有一个专用的服务类来表示他，叫做EndpointAddress，我们new一个它的实例：

```
[csharp]

EndpointAddress address = new EndpointAddress("http://localhost/IISService/HelloWCFService.svc");
```

只接受一个String参数，就是URI地址，这里我用了第三篇中建立的IIS服务的地址。

绑定，我们的服务端终结点要求的是wsHttpBinding，我们也可以用服务类来表示，叫做WSHttpBinding，我们new一个它的实例：

```
[csharp]

WSHttpBinding binding = new WSHttpBinding();
```

使用参数为空的构造函数就可以。

好的，元数据的其他两样也准备齐全，现在请通道工厂闪亮登场，我们new一个它的实例，用刚才建立的服务协定接口作为泛型参数，使用上面建立的地址和绑定对象作为构造函数的参数：

```
[csharp]

1.  ChannelFactory<IHelloWCF> factory = new ChannelFactory<IHelloWCF>(binding, address);
```

有了工厂，接下来就要开始生产通道，通过执行通道工厂的CreateChannel方法来生产一个通道，由于工厂是用我们的协定接口创建的，所生产的通道也是实现这个接口的，我们可以直接用协定接口来声明其返回值。

```
[csharp]

1.  IHelloWCF channel = factory.CreateChannel();
```

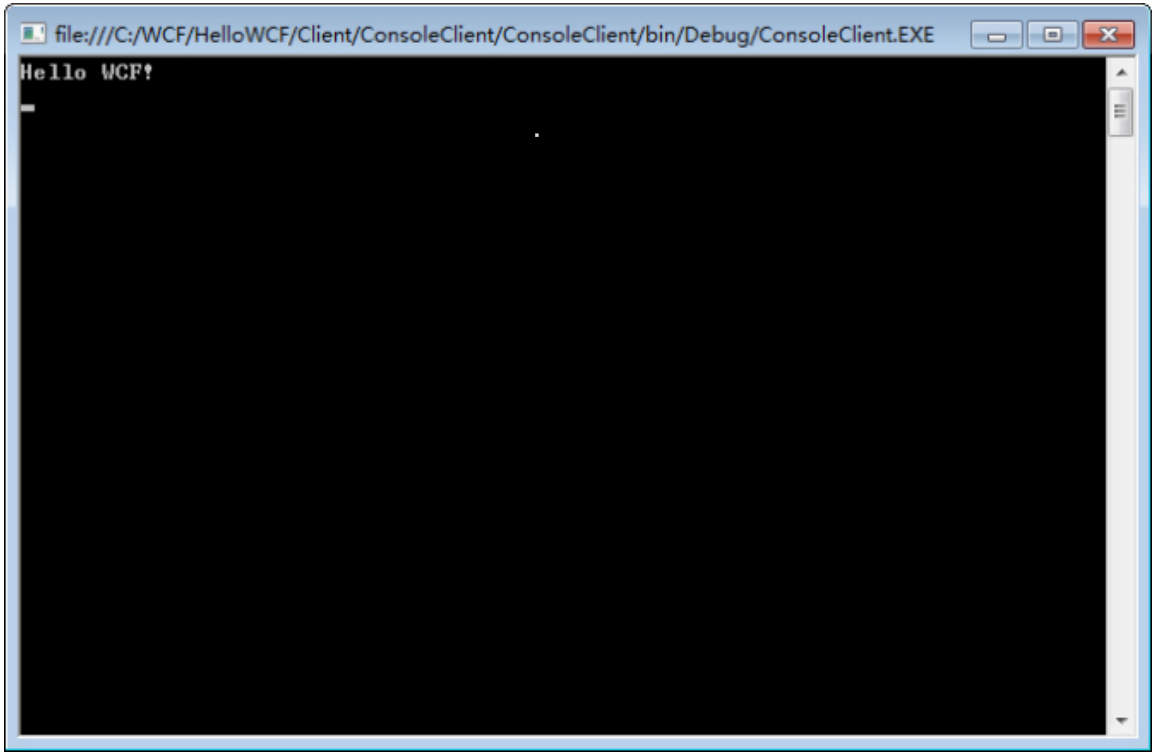
现在，通道已经打开，我们可以调用这个通道上的协定方法了。

```
[csharp]

1.  string result = channel.HelloWCF();
```

然后我们把结果输出，就像之前做的客户端程序一样。

```
[csharp]

1.  Console.WriteLine(result);
2.  Console.ReadLine();
```

F5运行一下，看结果



Yahoo!，我们没有使用元数交换的功能，凭着手绘的元数据和代码就完成了客户端到服务器端的通信。没有服务引用、没有配置文件，我们依然做得到。虽然对整个过程还不能完全明了，但是对通信过程已经有些理解了。
Program.cs的全部代码

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;

namespace ConsoleClient
{
    class Program
```

```
10.     {
11.         static void Main(string[] args)
12.         {
13.             EndpointAddress address = new EndpointAddress("http://localhost/IISService/HelloWCFSe
14.             WSHttpBinding binding = new WSHttpBinding();
15.
16.             ChannelFactory<IHelloWCF> factory = new ChannelFactory<IHelloWCF>
    (binding, address);
17.
18.             IHelloWCF channel = factory.CreateChannel();
19.
20.             string result = channel.HelloWCF();
21.
22.             Console.WriteLine(result);
23.             Console.ReadLine();
24.         }
25.     }
26.
27.     [ServiceContract]
28.     public interface IHelloWCF
29.     {
30.         [OperationContract]
31.         string HelloWCF();
32.     }
33. }
```

4. 再展开一点点

到这里已经很成功了，我们再稍微展开一些，还记得我们稍早前在服务引用生成的文件reference.cs看到的一个接口IHelloWCFChannel么？我们找遍解决方案也没发现哪个地方用到它？他是不是没用的东西呢，当然不会，我们现在来研究一下它。

我们上面手绘的程序可以打开通道并调用服务，然而我们回忆我们之前通过服务引用建立的客户端都是提供一个Close()方法来关闭服务连接的。使用我们这种方法按说也应该关闭通道才对，虽然客户端关闭通道就会被关闭了，但是在使用完通道后关闭之总是好的习惯。

可是，如何实现呢？我们发现通道无法提供关闭的方法，这是因为我们用IHelloWCF接口声明的通道对象，那这个对象自然只能提供接口所规定的方法了。而实际上通道对象本身是提供关闭方法，只是被我们显示的接口声明给屏蔽了，通道其实已经实现了另一个接口叫做IClientChannel，这个接口提供了打开和关闭通道的方法。如果我们要调用，只需要把通道对象强制转换成IClientChannel接口类型就可以了：

```
[csharp]

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

但是这么做不够自然优雅，强制转换总是让人莫名烦恼的东西。能不能保持IHelloWCF的对象类型并让他可以提供关闭方法呢？当然是可以的。我们再建立一个接口，让这个接口同时实现IHelloWCF服务协定接口和IClientChannel接口，并用这个新接口去new 通道工厂，这样生产出来的通道对象不就可以同时使用IHelloWCF中的服务操作和IClientChannel中的关闭通道的方法了么？

首先，做这个新接口，这个接口只是把服务协定接口和IClientChannel拼接，本身没有其他成员，是一个空接口。

```
[csharp]

1. public interface IHelloWCFChannel : IHelloWCF, IClientChannel
2. {
3.
4. }
```

然后，修改一下前面的建立通道工厂的语句，用这个新接口名作为泛型参数来new通道工厂

```
[csharp]
1. ChannelFactory<IHelloWCFChannel> factory = new ChannelFactory<IHelloWCFChannel>
(binding, address);
```

修改一下生产通道方法的对象声明类型，用新接口类型声明：

```
[csharp]
1. IHelloWCFChannel channel = factory.CreateChannel();
```

最后，我们在调用服务操作之后，就可以直接调用关闭通道的方法了：

```
[csharp]
1. channel.Close();
```

修改后的program.cs源代码：

```
[csharp]
1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Text;
5. using System.ServiceModel;
6.
7. namespace ConsoleClient
8. {
9.     class Program
10.     {
11.         static void Main(string[] args)
12.         {
13.             EndpointAddress address = new EndpointAddress("http://localhost/IISService/HelloWCFSe
14.             WSHttpBinding binding = new WSHttpBinding();
15.
16.             ChannelFactory<IHelloWCFChannel> factory = new ChannelFactory<IHelloWCFChannel>
(binding, address);
```

```
17.
18.             IHelloWCFChannel channel = factory.CreateChannel();
19.
20.             string result = channel.HelloWCF();
21.
22.             channel.Close();
23.
24.             Console.WriteLine(result);
25.             Console.ReadLine();
26.         }
27.     }
28.
29.     [ServiceContract]
30.     public interface IHelloWCF
31.     {
32.         [OperationContract]
33.         string HelloWCF();
34.     }
35.
36.     public interface IHelloWCFChannel : IHelloWCF, IClientChannel
37.     {
38.
39.     }
40.
41. }
```

现在，我们明白了服务引用中那个看上去没用的接口是什么作用了吧。然而服务引用中的实现跟我们这种还是有区别的，它使用了一个叫做ClientBase<>的类来进行通道通信，我们后面会展开。

5. 总结

今天的研究稍微深入了一点点，没有完全理解也没有关系，心里有个概念就可以了。
我们通过手绘代码的方法实现了客户端和服务端的通信，没有借助元数据交换工具。这让我们对服务端和客户端通信有了更接近真相的一层认识。其实所谓的元数据交换就是让我们获得这样甚至更简单的编程模型，它背后做的事情跟我们今天做的是很类似的，想像一下，产品级的服务可能有许多的协定接口，许多的终结点，我们不可能也不应该耗费力气把他们在客户端中再手动提供一次，因此元数据交换是很有意义的，我们应该学会使用它，我们还要在后面的学习中不断掌握驾驭元数据交换工具的办法。

相关资源

MSDN关于客户端架构的文档，仔细研读，必有收获。
客户端体系结构 (http://msdn.microsoft.com/zh-cn/library/ms729718.aspx)