



细细品味 C#

——.Net Remoting 专题

精
华
集
锦

csAxp

虾皮工作室

<http://www.cnblogs.com/xia520pi/>

2011 年 11 月 1 日

目录

1.1、版权声明.....	2
1.2、内容详情.....	2
1.2.1 .Net Remoting基础	2
1.2.2 Marshal、Disconnect与生命周期以及跟踪服务	17
1.2.3 Remoting事件处理全接触.....	25
1.2.4 关于Remoting.....	43
1.2.5 关于Remoting（续）	53
1.2.6 关于Remoting一些更改.....	57
1.2.7 Remoting的几个疑惑.....	59
1.2.8 Remoting疑惑续集.....	60
1.2.9 Remoting疑惑续集之再续	62
1.2.10 基于消息与.Net Remoting的分布式处理架构	66
1.2.11 .Net Remoting测试小技巧	78
1.2.12 .NET Remoting中的通道注册	80
1.2.13 在Remoting客户端激活用替换类以分离接口与实现.....	81
2.1、版权声明.....	84
2.2、内容详情.....	84
2.2.1 一步一步学Remoting之一：从简单开始.....	84
2.2.2 一步一步学Remoting之二：激活模式.....	87
2.2.3 一步一步学Remoting之三：复杂对象.....	92
2.2.4 一步一步学Remoting之四：承载方式（1）	97
2.2.5 一步一步学Remoting之四：承载方式（2）	101
2.2.6 一步一步学Remoting之五：异步操作.....	105
2.2.7 一步一步学Remoting之六：事件（1）	110
2.2.8 一步一步学Remoting之六：事件（2）	117
3.1、版权声明.....	123
3.2、内容详情.....	123
3.2.1 Remoting基本原理及其扩展机制（上）	123
3.2.2 Remoting基本原理及其扩展机制（中）	128
3.2.3 Remoting基本原理及其扩展机制（下）	134

1.1、版权声明

文章出处：<http://www.cnblogs.com/wayfarer/category/1235.html>

文章作者：TW 张逸

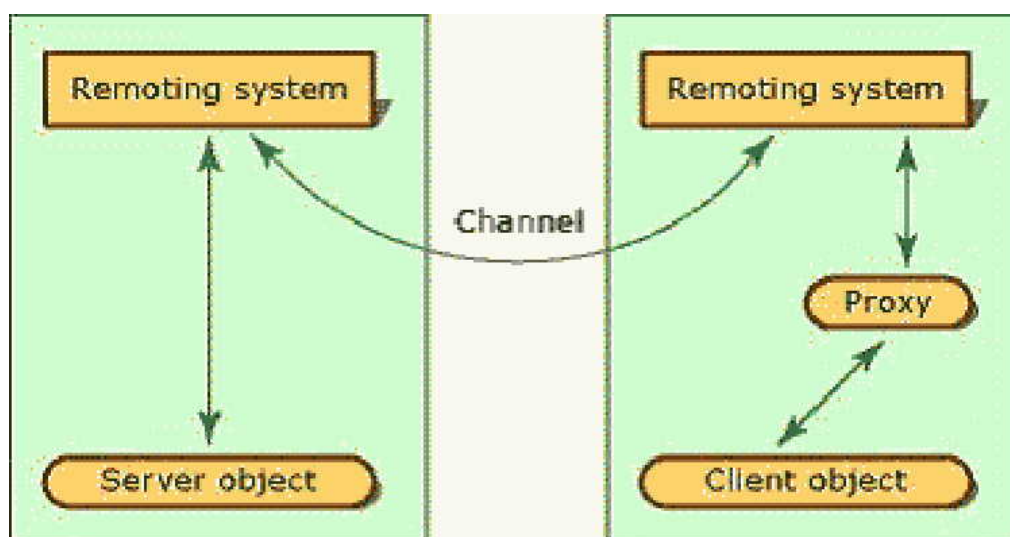
1.2、内容详情

1.2.1 .Net Remoting基础

【1】Remoting 基础

什么是 Remoting，简而言之，我们可以将其看作是一种分布式处理方式。从微软的产品角度来看，可以说 Remoting 就是 DCOM 的一种升级，它改善了很多功能，并极好的融合到 .Net 平台下。Microsoft? .NET Remoting 提供了一种允许对象通过应用程序域与另一对象进行交互的框架。这也正是我们使用 Remoting 的原因。为什么呢？在 Windows 操作系统中，是将应用程序分离为单独的进程。这个进程形成了应用程序代码和数据周围的一道边界。如果不采用进程间通信（RPC）机制，则在一个进程中执行的代码就不能访问另一进程。这是一种操作系统对应用程序的保护机制。然而在某些情况下，我们需要跨过应用程序域，与另外的应用程序域进行通信，即穿越边界。

在 Remoting 中是通过通道（channel）来实现两个应用程序域之间对象的通信的。如图所示：



首先，客户端通过 Remoting，访问通道以获得服务端对象，再通过代理解析为客户端对象。这就提供一种可能性，即以服务的方式来发布服务器对象。远程对象代码可以运行在服务器上（如服务器激活的对象和客户端激活的对象），然后客户端再通过 Remoting 连接服

务器，获得该服务对象并通过序列化在客户端运行。

在 Remoting 中，对于要传递的对象，设计者除了需要了解通道的类型和端口号之外，无需再了解数据包的格式。但必须注意的是，客户端在获取服务器端对象时，并不是获得实际的服务端对象，而是获得它的引用。这既保证了客户端和服务端有关对象的松散耦合，同时也优化了通信的性能。

1) Remoting 的两种通道

Remoting 的通道主要有两种：Tcp 和 Http。在 .Net 中，System.Runtime.Remoting.Channel 中定义了 IChannel 接口。IChannel 接口包括了 TcpChannel 通道类型和 Http 通道类型。它们分别对应 Remoting 通道的这两种类型。

TcpChannel 类型放在名字空间 System.Runtime.Remoting.Channel.Tcp 中。Tcp 通道提供了基于 Socket 的传输工具，使用 Tcp 协议来跨越 Remoting 边界传输序列化的消息流。TcpChannel 类型默认使用二进制格式序列化消息对象，因此它具有更高的传输性能。HttpChannel 类型放在名字空间 System.Runtime.Remoting.Channel.Http 中。它提供了一种使用 Http 协议，使其能在 Internet 上穿越防火墙传输序列化消息流。默认情况下，HttpChannel 类型使用 Soap 格式序列化消息对象，因此它具有更好的互操作性。通常在局域网内，我们更多地使用 TcpChannel；如果要穿越防火墙，则使用 HttpChannel。

2) 远程对象的激活方式

在访问远程类型的一个对象实例之前，必须通过一个名为 Activation 的进程创建它并进行初始化。这种客户端通过通道来创建远程对象，称为对象的激活。在 Remoting 中，远程对象的激活分为两大类：服务器端激活和客户端激活。

(1) 服务器端激活，又叫做 WellKnown 方式，很多又翻译为知名对象。为什么称为知名对象激活模式呢？是因为服务器应用程序在激活对象实例之前会在一个众所周知的统一资源标识符(URI)上来发布这个类型。然后该服务器进程会为此类型配置一个 WellKnown 对象，并根据指定的端口或地址来发布对象。 .Net Remoting 把服务器端激活又分为 SingleTon 模式和 SingleCall 模式两种。

SingleTon 模式：此为有状态模式。如果设置为 SingleTon 激活方式，则 Remoting 将为所有客户端建立同一个对象实例。当对象处于活动状态时，SingleTon 实例会处理所有后来的客户端访问请求，而不管它们是同一个客户端，还是其他客户端。SingleTon 实例将在方法调用中一直维持其状态。举例来说，如果一个远程对象有一个累加方法 (i=0; ++i)，被多个客户端（例如两个）调用。如果设置为 SingleTon 方式，则第一个客户获得值为 1，第二个客户获得值为 2，因为他们获得的对象实例是相同的。如果熟悉 Asp.Net 的状态管理，我们可以认为它是一种 Application 状态。

SingleCall 模式：SingleCall 是一种无状态模式。一旦设置为 SingleCall 模式，则当客户端调用远程对象的方法时，Remoting 会为每一个客户端建立一个远程对象实例，至于对象实例的销毁则是由 GC 自动管理的。同上一个例子而言，则访问远程对象的两个客户获得的

都是 1。我们仍然可以借鉴 Asp.Net 的状态管理，认为它是一种 Session 状态。

(2) 客户端激活。与 WellKnown 模式不同，Remoting 在激活每个对象实例的时候，会给每个客户端激活的类型指派一个 URI。客户端激活模式一旦获得客户端的请求，将为每一个客户端都建立一个实例引用。SingleCall 模式和客户端激活模式是有区别的：首先，对象实例创建的时间不一样。客户端激活方式是客户端一旦发出调用的请求，就实例化；而 SingleCall 则是要等到调用对象方法时再创建。其次，SingleCall 模式激活的对象是无状态的，对象生命期的管理是由 GC 管理的，而客户端激活的对象则有状态，其生命周期可自定义。其三，两种激活模式在服务器端和客户端实现的方法不一样。尤其是在客户端，SingleCall 模式是由 GetObject() 来激活，它调用对象默认的构造函数。而客户端激活模式，则通过 CreateInstance() 来激活，它可以传递参数，所以可以调用自定义的构造函数来创建实例。

【2】远程对象的定义

前面讲到，客户端在获取服务器端对象时，并不是获得实际的服务端对象，而是获得它的引用。因此在 Remoting 中，对于远程对象有一些必须的定义规范要遵循。

由于 Remoting 传递的对象是以引用的方式，因此所传递的远程对象类必须继承 MarshalByRefObject。MSDN 对 MarshalByRefObject 的说明是：MarshalByRefObject 是那些通过使用代理交换消息来跨越应用程序域边界进行通信的对象的基类。不是从 MarshalByRefObject 继承的对象会以隐式方式按值封送。当远程应用程序引用一个按值封送的对象时，将跨越远程处理边界传递该对象的副本。因为您希望使用代理方法而不是副本方法进行通信，因此需要继承 MarshalByRefObject。

以下是一个远程对象类的定义：

```
public class ServerObject : MarshalByRefObject
{
    public Person GetPersonInfo(string name, string sex, int age)
    {
        Person person = new Person();
        person.Name = name;
        person.Sex = sex;
        person.Age = age;
        return person;
    }
}
```

这个类只实现了最简单的方法，就是设置一个人的基本信息，并返回一个 Person 类对象。注意这里返回的 Person 类。由于这里所传递的 Person 则是以传值的方式来的，而 Remoting 要求必须是引用的对象，所以必须将 Person 类序列化。

因此，在 Remoting 中的远程对象中，如果还要调用或传递某个对象，例如类，或者结构，则该类或结构则必须实现串行化 Attribute[SerializableAttribute]：

```
[Serializable]
public class Person
{
    public Person()
    {

    }

    private string name;
    private string sex;
    private int age;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string Sex
    {
        get { return sex; }
        set { sex = value; }
    }

    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

将该远程对象以类库的方式编译成 Dll。这个 Dll 将分别放在服务器端和客户端，以添加引用。

在 Remoting 中能够传递的远程对象可以是各种类型，包括复杂的 DataSet 对象，只要它能够被序列化。远程对象也可以包含事件，但服务器端对于事件的处理比较特殊，我将在本系列之三中介绍。

【3】服务器端

根据第一部分所述，根据激活模式的不同，通道类型的不同服务器端的实现方式也有所不同。大体上说，服务器端应分为三步：

1) 注册通道

要跨越应用程序域进行通信，必须实现通道。如前所述，Remoting 提供了 IChannel 接口，分别包含 TcpChannel 和 HttpChannel 两种类型的通道。这两种类型除了性能和序列化数据的格式不同外，实现的方式完全一致，因此下面我们就以 TcpChannel 为例。

注册 TcpChannel，首先要在项目中添加引用“System.Runtime.Remoting”，然后 using 名字空间：System.Runtime.Remoting.Channel.Tcp。代码如下：

```
TcpChannel channel = new TcpChannel(8080);  
ChannelServices.RegisterChannel(channel);
```

在实例化通道对象时，将端口号作为参数传递。然后再调用静态方法 RegisterChannel()来注册该通道对象即可。

2) 注册远程对象

注册了通道后，要能激活远程对象，必须在通道中注册该对象。根据激活模式的不同，注册对象的方法也不同。

(1) Singleton 模式

对于 WellKnown 对象，可以通过静态方法
RemotingConfiguration.RegisterWellKnownServiceType()来实现：

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof(ServerRemoteObject.ServerObject),  
    "ServiceMessage", WellKnownObjectMode.Singleton);
```

(2) SingleCall 模式

注册对象的方法基本上和 Singleton 模式相同，只需要将枚举参数 WellKnownObjectMode 改为 SingleCall 就可以了。

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof(ServerRemoteObject.ServerObject),  
    "ServiceMessage", WellKnownObjectMode.SingleCall);
```

(3) 客户端激活模式

对于客户端激活模式，使用的方法又有不同，但区别不大，看了代码就一目了然。

```
RemotingConfiguration.ApplicationName = "ServiceMessage";  
RemotingConfiguration.RegisterActivatedServiceType(  
    typeof(ServerRemoteObject.ServerObject));
```

为什么要在注册对象方法前设置 `ApplicationName` 属性呢？其实这个属性就是该对象的 URI。对于 `WellKnown` 模式，URI 是放在 `RegisterWellKnownServiceType()` 方法的参数中，当然也可以拿出来专门对 `ApplicationName` 属性赋值。而 `RegisterActivatedServiceType()` 方法的重载中，没有 `ApplicationName` 的参数，所以必须分开。

3) 注销通道

如果要关闭 `Remoting` 的服务，则需要注销通道，也可以关闭对通道的监听。在 `Remoting` 中当我们注册通道的时候，就自动开启了通道的监听。而如果关闭了对通道的监听，则该通道就无法接受客户端的请求，但通道仍然存在，如果你想再一次注册该通道，会抛出异常。

```
//获得当前已注册的通道；
IChannel[] channels = ChannelServices.RegisteredChannels;

//关闭指定名为 MyTcp 的通道；
foreach (IChannel eachChannel in channels)
{
    if (eachChannel.ChannelName == "MyTcp")
    {
        TcpChannel tcpChannel = (TcpChannel)eachChannel;
        //关闭监听；
        tcpChannel.StopListening(null);
        //注销通道；
        ChannelServices.UnregisterChannel(tcpChannel);
    }
}
```

代码中，`RegisteredChannel` 属性获得的是当前已注册的通道。在 `Remoting` 中，是允许同时注册多个通道的，这一点会在后面说明。

【4】客户端

客户端主要做两件事，一是注册通道。这一点从图一就可以看出，`Remoting` 中服务器端和客户端都必须通过通道来传递消息，以获得远程对象。第二步则是获得该远程对象。

1) 注册通道：

```
TcpChannel channel = new TcpChannel();
ChannelServices.RegisterChannel(channel);
```

注意在客户端实例化通道时，是调用的默认构造函数，即没有传递端口号。事实上，这个端口号是缺一不可的，只不过它的指定被放在后面作为了 `Uri` 的一部分。

2) 获得远程对象

与服务器端相同，不同的激活模式决定了客户端的实现方式也将不同。不过这个区别仅仅是 WellKnown 激活模式和客户端激活模式之间的区别，而对于 SingleTon 和 SingleCall 模式，客户端的实现完全相同。

(1) WellKnown 激活模式

要获得服务器端的知名远程对象，可通过 Activator 进程的 GetObject()方法来获得：

```
ServerRemoteObject.ServerObject serverObj =  
(ServerRemoteObject.ServerObject)Activator.GetObject(  
    typeof(ServerRemoteObject.ServerObject),  
    "tcp://localhost:8080/ServiceMessage");
```

首先以 WellKnown 模式激活，客户端获得对象的方法是使用 GetObject ()。其中参数第一个是远程对象的类型。第二个参数就是服务器端的 uri。如果是 http 通道，自然是用 http://localhost:8080/ServiceMessage 了。因为我是用本地机，所以这里是 localhost，你可以用具体的服务器 IP 地址来代替它。端口必须和服务器端的端口一致。后面则是服务器定义的远程对象服务名，即 ApplicationName 属性的内容。

(2) 客户端激活模式

如前所述，WellKnown 模式在客户端创建对象时，只能调用默认的构造函数，上面的代码就说明了这一点，因为 GetObject()方法不能传递构造函数的参数。而客户端激活模式则可以通过自定义的构造函数来创建远程对象。

客户端激活模式有两种方法：

1) 调用 RemotingConfiguration 的静态方法 RegisterActivatedClientType()。这个方法返回值为 Void，它只是将远程对象注册在客户端而已。具体的实例化还需要调用对象类的构造函数。

```
RemotingConfiguration.RegisterActivatedClientType(  
    typeof(ServerRemoteObject.ServerObject),  
    "tcp://localhost:8080/ServiceMessage");  
ServerRemoteObject.ServerObject serverObj = new ServerRemoteObject.ServerObject();
```

2) 调用进程 Activator 的 CreateInstance()方法。这个方法将创建方法参数指定类型的类对象。它与前面的 GetObject()不同的是，它要在客户端调用构造函数，而 GetObject()只是获得对象，而创建实例是在服务器端完成的。CreateInstance()方法有很多个重载，我着重说一下其中常用的两个。

```
a、 public static object CreateInstance(Type type, object[] args, object[] activationAttributes);
```

参数说明：

type：要创建的对象类型。

args：与要调用构造函数的参数数量、顺序和类型匹配的参数数组。如果 **args** 为空数组或空引用（Visual Basic 中为 **Nothing**），则调用不带任何参数的构造函数（默认构造函数）。

activationAttributes：包含一个或多个可以参与激活的属性的数组。

这里的参数 **args** 是一个 **object[]** 数组类型。它可以传递要创建对象的构造函数中的参数。从这里其实可以得到一个结论：**WellKnown** 激活模式所传递的远程对象类，只能使用默认的构造函数；而 **Activated** 模式则可以用用户自定义构造函数。**activationAttributes** 参数在这个方法中通常用来传递服务器的 **url**。

假设我们的远程对象类 **ServerObject** 有个构造函数：

```
ServerObject(string pName,string pSex,int pAge)
{
    name = pName;
    sex = pSex;
    age = pAge;
}
```

那么实现的代码是：

```
object[] attrs = { new UriAttribute("tcp://localhost:8080/ServiceMessage")};
object[] objs = new object[3];
objs[0] = "wayfarer";
objs[1] = "male";
objs[2] = 28;
ServerRemoteObject.ServerObject = Activator.CreateInstance(
    typeof(ServerRemoteObject.ServerObject),objs,attrs);
```

可以看到，**objs[]** 数组传递的就是构造函数的参数。

```
b、public static ObjectHandle CreateInstance(string assemblyName, string typeName, object[]
activationAttribute);
```

参数说明：

assemblyName：将在其中查找名为 **typeName** 的类型的程序集的名称。如果 **assemblyName** 为空引用（Visual Basic 中为 **Nothing**），则搜索正在执行的程序集。

typeName：首选类型的名称。

activationAttributes：包含一个或多个可以参与激活的属性的数组。

参数说明一目了然。注意这个方法返回值为 **ObjectHandle** 类型，因此代码与前不同：

```
object[] attrs = { new UriAttribute("tcp://localhost:8080/EchoMessage")};
```

```
ObjectHandle handle = Activator.CreateInstance("ServerRemoteObject",
                                                "ServerRemoteObject.ServerObject",attrs);
ServerRemoteObject.ServerObject obj =
    (ServerRemoteObject.ServerObject)handle.Unwrap();
```

这个方法实际上是调用的默认构造函数。ObjectHandle.Unwrap()方法是返回被包装的对象。

说明：要使用 UriAttribute，还需要在命名空间中添加：using System.Runtime.Remoting.Activation;

【5】Remoting 基础的补充

通过上面的描述，基本上已经完成了一个最简单的 Remoting 程序。这是一个标准的创建 Remoting 程序的方法，但在实际开发过程中，我们遇到的情况也许千奇百怪，如果只掌握一种所谓的“标准”，就妄想可以“一招鲜、吃遍天”，是不可能的。

1) 注册多个通道

在 Remoting 中，允许同时创建多个通道，即根据不同的端口创建不同的通道。但是，Remoting 要求通道的名字必须不同，因为它要用来作为通道的唯一标识符。虽然 IChannel 有 ChannelName 属性，但这个属性是只读的。因此前面所述的创建通道的方法无法实现同时注册多个通道的要求。

这个时候，我们必须用到 System.Collection 中的 IDictionary 接口：

注册 Tcp 通道：

```
IDictionary tcpProp = new Hashtable();
tcpProp["name"] = "tcp9090";
tcpProp["port"] = 9090;
IChannel channel = new TcpChannel(tcpProp,
    new BinaryClientFormatterSinkProvider(),
    new BinaryServerFormatterSinkProvider());
ChannelServices.RegisterChannel(channel);
```

注册 Http 通道：

```
IDictionary httpProp = new Hashtable();
httpProp["name"] = "http8080";
httpProp["port"] = 8080;
IChannel channel = new HttpChannel(httpProp,
    new SoapClientFormatterSinkProvider(),
    new SoapServerFormatterSinkProvider());
ChannelServices.RegisterChannel(channel);
```

在 `name` 属性中，定义不同的通道名称就可以了。

2) 远程对象元数据相关性

由于服务器端和客户端都要用到远程对象，通常的方式是生成两份完全相同的对象 `Dll`，分别添加引用。不过为了代码的安全性，且降低客户端对远程对象元数据的相关性，我们有必要对这种方式进行改动。即在服务器端实现远程对象，而在客户端则删除这些实现的元数据。

由于激活模式的不同，在客户端创建对象的方法也不同，所以要分离元数据的相关性，也应分为两种情况。

(1) WellKnown 激活模式：

通过接口来实现。在服务器端，提供接口和具体类的实现，而在客户端仅提供接口：

```
public interface IServerObject
{
    Person GetPersonInfo(string name,string sex,int age);
}

public class ServerObject:MarshalByRefObject,IServerObject
{ .....}
```

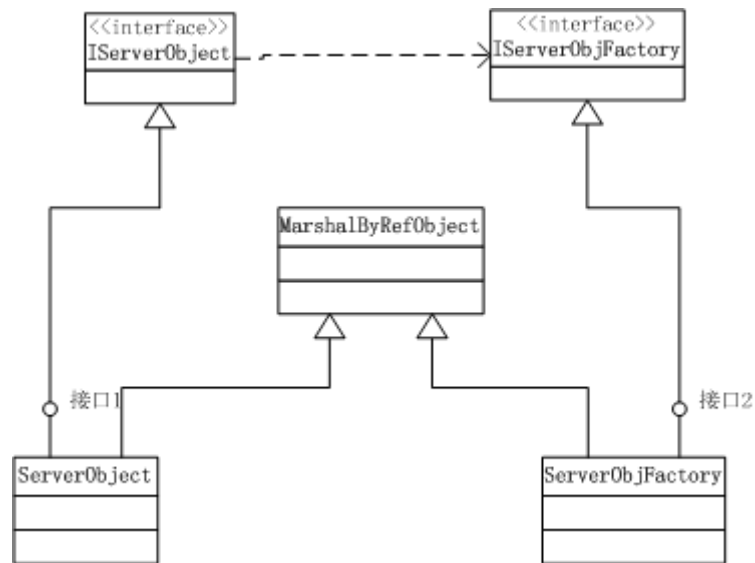
注意：两边生成该对象程序集的名字必须相同，严格地说，是命名空间的名字必须相同。

(2) 客户端激活模式：

如前所述，对于客户端激活模式，不管是使用静态方法，还是使用 `CreateInstance()` 方法，都必须在客户端调用构造函数实例化对象。所以，在客户端我们提供的远程对象，就不能只提供接口，而没有类的实现。实际上，要做到与远程对象元数据的分离，可以由两种方法供选择：

a、利用 WellKnown 激活模式模拟客户端激活模式：

方法是利用设计模式中的“抽象工厂”，下面的类图表描述了总体解决方案：



我们在服务器端的远程对象中加上抽象工厂的接口和实现类：

```

public interface IServerObject
{
    Person GetPersonInfo(string name,string sex,int age);
}

public interface IServerObjFactory
{
    IServerObject CreateInstance();
}

public class ServerObject:MarshalByRefObject,IServerObject
{
    public Person GetPersonInfo(string name,string sex,int age)
    {
        Person person = new Person();
        person.Name = name;
        person.Sex = sex;
        person.Age = age;
        return person;
    }
}

public class ServerObjFactory:MarshalByRefObject,IServerObjFactory
{
    public IServerObject CreateInstance()
    {
        return new ServerObject();
    }
}
  
```

```
}
}
```

然后再客户端的远程对象中只提供工厂接口和原来的对象接口：

```
public interface IServerObject
{
    Person GetPersonInfo(string name,string sex,int age);
}

public interface IServerObjFactory
{
    IServerObject CreateInstance();
}
```

我们用 WellKnown 激活模式注册远程对象，在服务器端：

```
//传递对象;
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(ServerRemoteObject.ServerObjFactory),
    "ServiceMessage",WellKnownObjectMode.SingleCall);
```

注意这里注册的不是 ServerObject 类对象，而是 ServerObjFactory 类对象。

```
客户端：
ServerRemoteObject.IServerObjFactory serverFactory =
    (ServerRemoteObject.IServerObjFactory) Activator.GetObject(
        typeof(ServerRemoteObject.IServerObjFactory),
        "tcp://localhost:8080/ServiceMessage");

ServerRemoteObject.IServerObject serverObj = serverFactory.CreateInstance();
```

为什么说这是一种客户端激活模式的模拟呢？从激活的方法来看，我们是使用了 SingleCall 模式来激活对象，但此时激活的并非我们要传递的远程对象，而是工厂对象。如果客户端要创建远程对象，还应该通过工厂对象的 CreateInstance()方法来获得。而这个方法正是在客户端调用的。因此它的实现方式就等同于客户端激活模式。

b、利用替代类来取代远程对象的元数据

实际上，我们可以用一个 trick，来欺骗 Remoting。这里所说的替代类就是这个 trick 了。既然是提供服务，Remoting 传递的远程对象其实现的细节当然是放在服务器端。而要在客户端放对象的副本，不过是因为客户端必须调用构造函数，而采取的无奈之举。既然具体的实现是在服务器端，又为了能在客户端实例化，那么在客户端就实现这些好了。至于实现的

细节，就不用管了。

如果远程对象有方法，服务器端则提供方法实现，而客户端就提供这个方法就 OK 了，至于里面的实现，你可以是抛出一个异常，或者 `return` 一个 `null` 值；如果方法返回 `void`，那么里面可以是空。关键是这个客户端类对象要有这个方法。这个方法的实现，其实和方法的声明差不多，所以我说是一个 `trick`。方法如是，构造函数也如此。

还是用代码来说明这种“阴谋”，更直观：

服务器端：

```
public class ServerObject:MarshalByRefObject
{
    public ServerObject()
    {

    }

    public Person GetPersonInfo(string name,string sex,int age)
    {
        Person person = new Person();
        person.Name = name;
        person.Sex = sex;
        person.Age = age;
        return person;
    }
}
```

客户端：

```
public class ServerObject:MarshalByRefObject
{
    public ServerObj()
    {
        throw new System.NotImplementedException();
    }

    public Person GetPersonInfo(string name,string sex,int age)
    {
        throw new System.NotImplementedException();
    }
}
```

比较客户端和服务端，客户端的方法 `GetPersonInfo()`，没有具体的实现细节，只是抛出了一个异常。或者直接写上语句 `return null`，照样 OK。我们称客户端的这个类为远程对象的替代类。

3) 利用配置文件实现

前面所述的方法，于服务器 uri、端口、以及激活模式的设置是用代码来完成的。其实我们也可以用配置文件来设置。这样做有个好处，因为这个配置文件是 Xml 文档。如果需要改变端口或其他，我们就不需要修改程序，并重新编译，而是只需要改变这个配置文件即可。

(1) 服务器端的配置文件：

```
<configuration>
  <system.runtime.remoting>
    <application name="ServerRemoting">
      <service>
        <wellknown mode="Singleton" type="ServerRemoteObject.ServerObject"
objectUri="ServiceMessage"/>
      </service>
      <channels>
        <channel ref="tcp" port="8080"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

如果是客户端激活模式，则把 wellknown 改为 activated，同时删除 mode 属性。

把该配置文件放到服务器程序的应用程序文件夹中，命名为 ServerRemoting.config。那么前面的服务器端程序直接用这条语句即可：

```
RemotingConfiguration.Configure("ServerRemoting.config");
```

(2) 客户端配置文件

如果是客户端激活模式，修改和上面一样。调用也是使用 RemotingConfiguration.Configure() 方法来调用存储在客户端的配置文件。

配置文件还可以放在 machine.config 中。如果客户端程序是 web 应用程序，则可以放在 web.config 中。

4) 启动/关闭指定远程对象

Remoting 中没有提供类似 UnregisterWellKnownServiceType() 的方法，也即是说，一旦通过注册了远程对象，如果没有关闭通道的话，该对象就一直存在于通道中。只要客户端激活该对象，就会创建对象实例。如果 Remoting 传送的只有一个远程对象，这不存在问题，

关闭通道就可以了。如果传送多个远程对象呢？要关闭指定的远程对象应该怎么做？关闭之后又需要启动又该如何？

我们注意到在 **Remoting** 中提供了 **Marshal()**和 **Disconnect()**方法,答案就在这里。**Marshal()**方法是将 **MarshalByRefObject** 类对象转化为 **ObjRef** 类对象，这个对象是存储生成代理以与远程对象通讯所需的所有相关信息。这样就可以将该实例序列化以便在应用程序域之间以及通过网络进行传输，客户端就可以调用了。而 **Disconnect()**方法则将具体的实例对象从通道中断开。

方法如下：

首先注册通道：

```
TcpChannel channel = new TcpChannel(8080);  
ChannelServices.RegisterChannel(channel);
```

接着启动服务：

先在服务器端实例化远程对象。

```
ServerObject obj = new ServerObject();
```

然后，注册该对象。注意这里不用 **RemotingConfiguration.RegisterWellKnownServiceType()**，而是使用 **RemotingServices.Marshal()**：

```
ObjRef objrefWellKnown = RemotingServices.Marshal(obj, "ServiceMessage");
```

如果要注销对象，则：

```
RemotingServices.Disconnect(obj);
```

要注意，这里 **Disconnect** 的类对象必须是前面实例化的对象。正因为此，我们可以根据需要创建指定的远程对象，而关闭时，则 **Disconnect** 之前实例化的对象。

至于客户端的调用，和前面 **WellKnown** 模式的方法相同，仍然是通过 **Activator.GetObject()**来获得。但从实现代码来看，我们会注意到一个问题，由于服务器端是显式的实例化了远程对象，因此不管客户端有多少，是否相同，它们调用的都是同一个远程对象。因此我们将这个方法称为模拟的 **SingleTon** 模式。

客户端激活模式

我们也可以通过 **Marshal()**和 **Disconnect()**来模拟客户端激活模式。首先我们来回顾“远程对象元数据相关性”一节，在这一节中，我说到采用设计模式的“抽象工厂”来创建对象实例，以此用 **SingleCall** 模式来模拟客户端激活模式。在仔细想想前面的模拟的 **SingleTon** 模式。是不是答案就将呼之欲出呢？

在“模拟的 Singleton”模式中，我们是将具体的远程对象实例进行 Marshal，以此让客户端获得该对象的引用信息。那么我们换一种思路，当我们用抽象工厂提供接口，工厂类实现创建远程对象的方法。然后我们在服务器端创建工厂类实例。再将这个工厂类实例进行 Marshal。而客户端获取对象时，不是获取具体的远程对象，而是获取具体的工厂类对象。然后再调用 CreateInstance()方法来创建具体的远程对象实例。此时，对于多个客户端而言，调用的是同一个工厂类对象；然而远程对象是在各个客户端自己创建的，因此对于远程对象而言，则是由客户端激活，创建的是不同对象了。

当我们要启动/关闭指定对象时，只需要用 Disconnect()方法来注销工厂类对象就可以了。

【6】结论

Microsoft.Net Remoting 真可以说是博大精深。整个 Remoting 的内容不是我这一篇小文所能尽述的，更不是我这个 Remoting 的初学者所能掌握的。王国维在《人间词话》一书中写到：古今之成大事业大学问者，必经过三种境界。“昨夜西风凋碧树，独上高楼，望尽天涯路。”此第一境界也。“衣带渐宽终不悔，为伊消得人憔悴。”此第二境界也。“众里寻他千百度，蓦然回首，那人却在灯火阑珊处。”此第三境界也。如以此来形容我对 Remoting 的学习，还处于“独上高楼，望尽天涯路”的时候，真可以说还未曾登堂入室。

或许需得“衣带渐宽”，学得 Remoting “终不悔”，方才可以“蓦然回首”吧。

1.2.2 Marshal、Disconnect与生命周期以及跟踪服务

【1】远程对象的激活

在 Remoting 中有三种激活方式，一般的实现是通过 RemotingServices 类的静态方法来完成。工作过程事实上是将该远程对象注册到通道中。由于 Remoting 没有提供与之对应的 Unregister 方法来注销远程对象，所以如果需要注册/注销指定对象，微软推荐使用 Marshal（一般译为编组）和 Disconnect 配对使用。在《Net Remoting 基础篇》中我已经谈到：Marshal()方法是将 MarshalByRefObject 类对象转化为 ObjRef 类对象，这个对象是存储生成代理以及远程对象通讯所需的所有相关信息。这样就可以将该实例序列化以便在应用程序域之间以及通过网络进行传输，客户端就可以调用了。而 Disconnect()方法则将具体的实例对象从通道中断开。

根据上述说明，Marshal()方法对远程对象以引用方式进行编组（Marshal-by-Reference，MBR），并将对象的代理信息放到通道中。客户端可以通过 Activator.GetObject()来获取。如果用户要注销该对象，则通过调用 Disconnect()方法。那么这种方式对于编组的远程对象是否存在生命周期的管理呢？这就是本文所要描述的问题。

【2】生命周期

在 CLR 中，框架提供了 GC（垃圾回收器）来管理内存中对象的生命周期。同样的，.Net

Remoting 使用了一种分布式垃圾回收，基于租用的形式来管理远程对象的生命周期。

早期的 DCOM 对于对象生命周期的管理是通过 ping 和引用计数来确定对象何时应当作为垃圾回收。然而 ping 引起的网络流量对分布式应用程序的性能是一种痛苦的负担，它大大地影响了分布式处理的整体性能。.Net Remoting 在每个应用程序域中都引入一个租用管理器，为每个服务器端的 SingleTon，或每个客户端激活的远程对象保存着对租用对象的引用。（说明：对于服务器端激活的 SingleCall 方式，由于它是无状态的，对于每个激活的远程对象，都由 CLR 的 GC 来自动回收，因此对于 SingleCall 模式激活的远程对象，不存在生命周期的管理。）

1) 租用

租用是个封装了 TimeSpan 值的对象，用以管理远程对象的生存期。在 .Net Remoting 中提供了定义租用功能的 ILease 接口。当 Remoting 通过 SingleTon 模式或客户端激活模式来激活远程对象时，租用对象调用从 System.MarshalByRefObject 继承的 InitializeLifetimeService 方法，向对象请求租用。

ILease 接口定义了有关生命周期的属性，均为 TimeSpan 值。如下：

InitialLeaseTime：初始化有效时间，默认值为 300 秒，如果为 0，表示永不过期；

RenewOnCallTime：调用远程对象一个方法时的租用更新时间，默认值为 120 秒；

SponsorshipTimeout：超时值，通知 Sponsor（发起人）租用过期后，Remoting 会等待的时间，默认值为 120 秒；

CurrentLeaseTime：当前租用时间，首次获得租用时，为 InitializeLeaseTime 的值。

Remoting 的远程对象因为继承了 MarshalByRefObject，因此默认继承了 InitializeLifetimeService 方法，那么租用的相关属性为默认值。如果要改变这些设置，可以在远程对象中重写该方法。例如：

```
public override object InitializeLifetimeService()
{
    ILease lease = (ILease)base.InitializeLifetimeService();
    if (lease.CurrentState == LeaseState.Initial)
    {
        lease.InitialLeaseTime = TimeSpan.FromMinutes(1);
        lease.RenewOnCallTime = TimeSpan.FromSeconds(20);
    }
    return lease;
}
```

也可以忽略该方法，将对象的租用周期改变为无限：

```
public override object InitializeLifetimeService()
{
    return null;
}
```

```
}
```

2) 租用管理器

如果是前面所说的租用主要是应用在每个具体的远程对象上,那么租用管理器是服务器端专门用来管理远程对象生命周期的管理器,它维持着一个 `System.Hashtable` 成员,将租用映射为 `System.DateTime` 实例表示每个租用何时应过期。`Remoting` 采用轮询的方式以一定的时间唤醒租用管理器,检查每个租用是否过期。默认为每 10 秒钟唤醒一次。轮询的间隔可以配置,如将轮询间隔设置为 5 分钟:

```
LifetimeService.LeaseManagerPollTime = System.TimeSpan.FromMinutes(5);
```

我们还可以在租用管理器中设置远程对象租用的属性,如改变远程对象的初始有效时间为永久有效:

```
LifetimeServices.LeaseTime = TimeSpan.Zero;
```

我们也可以通过配置文件来设置生命周期,如:

```
<configuration>
  <system.runtime.remoting>
    <application name = "SimpleServer">
      <lifetime leaseTime = "0" sponsorshipTimeOut = "1M" renewOnCallTime = "1M" pollTime = "30S"/>
    </application>
  </system.runtime.remoting>
</configuration>
```

注: 配置文件中的 `pollTime` 即为上面所说的租用管理器的轮询间隔时间 `LeaseManagerPollTime`。

租用管理器对于生命周期的设置是针对服务器上所有的远程对象。当我们通过配置文件或租用管理器设置租用的属性时,所有远程对象的生命周期都遵循该设置,除非我们对于指定的远程对象通过重写 `InitializeLifetimeService` 方法,改变了相关配置。也就是说,远程对象的租用配置优先级高于服务器端配置。

3) 发起人 (Sponsor)

发起人是针对客户端而言的。远程对象就是发起人要租用的对象,发起人可以与服务器端签订租约,约定租用时间。一旦到期后,发起人还可以续租,就像现实生活中租方的契约,房东、租房者之间的关系一样。

在 .Net Framework 中的 `System.Runtime.Remoting.Lifetime` 命名空间中定义了 `ClientSponsor` 类,该类继承了 `System.MarshalByRefObject`,并实现了 `ISponsor` 接口。

ClientSponsor 类的属性和方法，可以参考 MSDN。

客户端要使用发起人机制，必须创建 ClientSponsor 类的一个实例。然后调用相关方法如 Register()或 Renewal()方法来注册远程对象或延长生命周期。如：

```
RemotingObject obj = new RemotingObject();
ClientSponsor sponsor = new ClientSponsor();
sponsor.RenewalTime = TimeSpan.FromMinutes(2);
sponsor.Register(obj);
```

续租时间也可以在 ClientSponsor 的构造函数中直接设置，如：

```
ClientSponsor sponsor = new ClientSponsor(TimeSpan.FromMinutes(2));
sponsor.Register(obj);
```

我们也可以自己编写 Sponsor 来管理发起人机制，这个类必须继承 ClientSponsor 并实现 ISponsor 接口。

【3】跟踪服务

如前所述，我们要判断通过 Marshal 编组远程对象是否存在生命周期的管理。在 Remoting 中，可以通过跟踪服务程序来监视 MBR 对象的编组进程。

我们可以创建一个简单的跟踪处理程序，该程序实现接口 ITrackingHandler。接口 ITrackingHandler 定义了 3 个方法，MarshalObject、UnmarshalObject 和 DisconnectedObject。当远程对象被编组、解组和断开连接时，就会调用相应的方法。下面是该跟踪处理类的代码：

```
public class MyTracking:ITrackingHandler
{
    public MyTracking()
    {
        //
        // TODO: 在此处添加构造函数逻辑
        //
    }

    public void MarshaledObject(object obj,ObjRef or)
    {
        Console.WriteLine();
        Console.WriteLine(" 对 象 " + obj.ToString() + " is marshaled at " +
DateTime.Now.ToShortTimeString());
    }
}
```

```

public void UnmarshaledObject(object obj, ObjRef or)
{
    Console.WriteLine();
    Console.WriteLine(" 对 象 " + obj.ToString() + " is unmarshaled at " +
DateTime.Now.ToShortTimeString());
}

public void DisconnectedObject(object obj)
{
    Console.WriteLine(obj.ToString() + " is disconnected at "
+ DateTime.Now.ToShortTimeString());
}
}

```

然后再服务器端创建该跟踪处理类的实例，并注册跟踪服务：

```
TrackingServices.RegisterTrackingHandler(new MyTracking());
```

【4】测试

1) 建立两个远程对象，并重写 InitializeLifetimeService 方法：

对象一：AppService1

初始生命周期：1 分钟

```

public class AppService1:MarshalByRefObject
{
    public void PrintString(string contents)
    {
        Console.WriteLine(contents);
    }
    public override object InitializeLifetimeService()
    {
        ILease lease = (ILease)base.InitializeLifetimeService();
        if (lease.CurrentState == LeaseState.Initial)
        {
            lease.InitialLeaseTime = TimeSpan.FromMinutes(1);
            lease.RenewOnCallTime = TimeSpan.FromSeconds(20);
        }
        return lease;
    }
}

```

对象二：AppService2

初始生命周期：3 分钟

```
public class AppService2:MarshalByRefObject
{
    public void PrintString(string contents)
    {
        Console.WriteLine(contents);
    }

    public override object InitializeLifetimeService()
    {
        ILease lease = (ILease)base.InitializeLifetimeService();
        if (lease.CurrentState == LeaseState.Initial)
        {
            lease.InitialLeaseTime = TimeSpan.FromMinutes(3);
            lease.RenewOnCallTime = TimeSpan.FromSeconds(40);
        }
        return lease;
    }
}
```

为简便起见，两个对象的方法都一样。

2) 服务器端

- (1) 首先建立如上的监控处理类；
- (2) 注册通道：

```
TcpChannel channel = new TcpChannel(8080);
ChannelServices.RegisterChannel(channel);
```

- (3) 设置租用管理器的初始租用时间为无限：

```
LifetimeServices.LeaseTime = TimeSpan.Zero;
```

- (4) 创建该跟踪处理类的实例，并注册跟踪服务：

```
TrackingServices.RegisterTrackingHandler(new MyTracking());
```

- (5) 编组两个远程对象：

```
ServerAS.AppService1 service1 = new ServerAS1.AppService1();
ObjRef objRef1 = RemotingServices.Marshal((MarshalByRefObject)service1,"AppService1");
```

```
ServerAS.AppService2 service2 = new ServerAS1.AppService2();
ObjRef objRef2 = RemotingServices.Marshal((MarshalByRefObject)service2,"AppService2");
```

(6) 使服务器端保持运行：

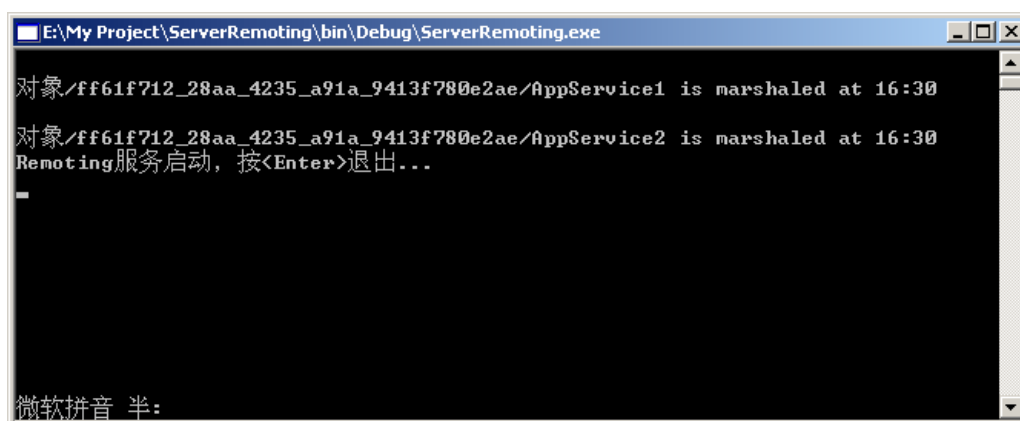
```
Console.WriteLine("Remoting 服务启动，按退出...");
Console.ReadLine();
```

3) 客户端

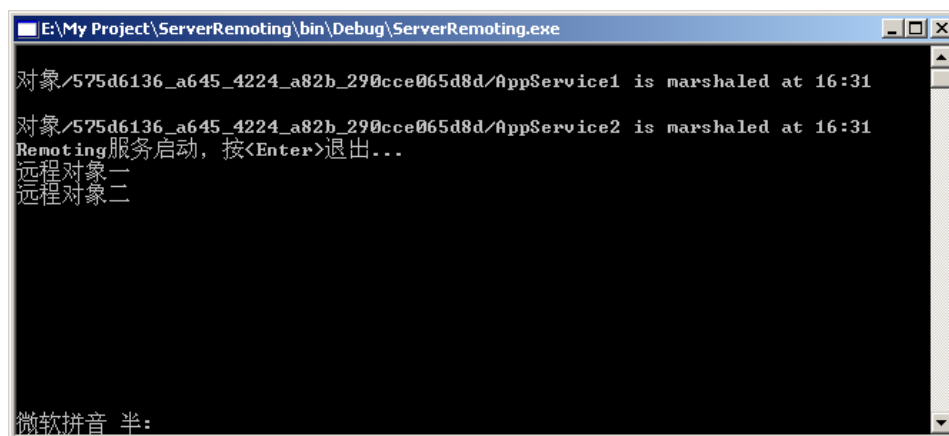
通过 Activator.GetObject()获得两个远程对象，并调用其方法 PrintString。代码略。

4) 运行测试：

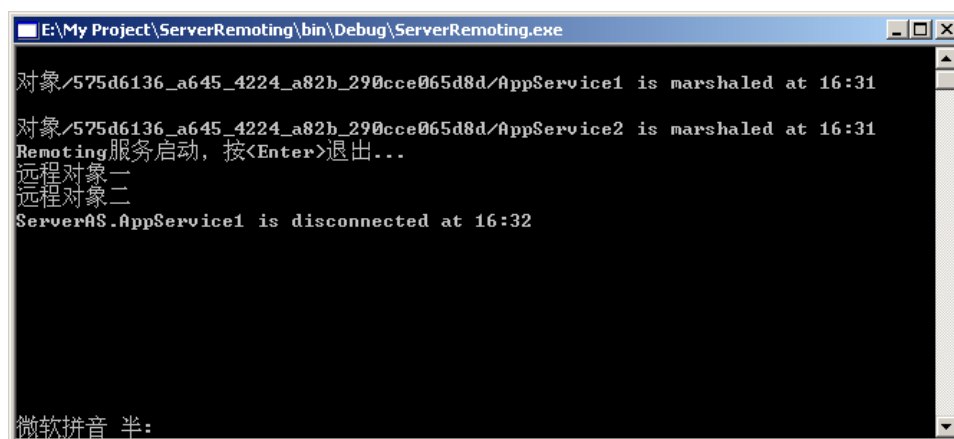
运行服务器端和客户端，由于监控程序将监视远程对象的编组进程，因此在运行开始，就会显示远程对象已经被 Marshal：



然后再客户端调用这两个远程对象的 PrintString 方法，服务器端接受字符串：



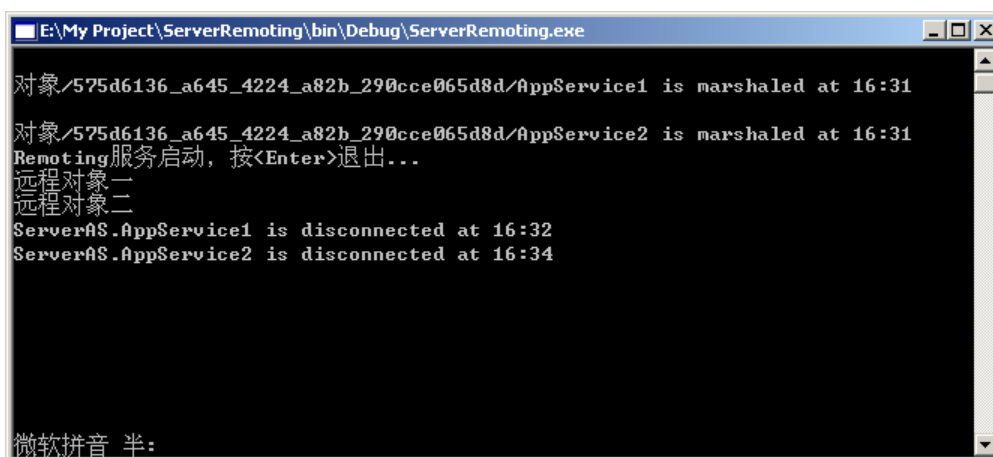
一分钟后，远程对象一自动被 Disconnect：



```
E:\My Project\ServerRemoting\bin\Debug\ServerRemoting.exe
对象/575d6136_a645_4224_a82b_290cce065d8d/AppService1 is marshaled at 16:31
对象/575d6136_a645_4224_a82b_290cce065d8d/AppService2 is marshaled at 16:31
Remoting服务启动, 按<Enter>退出...
远程对象一
远程对象二
ServerAS.AppService1 is disconnected at 16:32
微软拼音 半:
```

此时客户端如要调用远程对象一，会抛出 `RemotingException` 异常；

又一分种后，远程对象二被 Disconnect 了：



```
E:\My Project\ServerRemoting\bin\Debug\ServerRemoting.exe
对象/575d6136_a645_4224_a82b_290cce065d8d/AppService1 is marshaled at 16:31
对象/575d6136_a645_4224_a82b_290cce065d8d/AppService2 is marshaled at 16:31
Remoting服务启动, 按<Enter>退出...
远程对象一
远程对象二
ServerAS.AppService1 is disconnected at 16:32
ServerAS.AppService2 is disconnected at 16:34
微软拼音 半:
```

用户还可以根据这个代码测试 `RenewOnCallTime` 的时间是否正确。也即是说，在对象还未被 Disconnect 时，调用对象，则从调用对象的这一刻起，其生命周期不再是原来设定的初始有效时间值（`InitialLeaseTime`），而是租用更新时间值（`RenewOnCallTime`）。另外，如果这两个远程对象没有重写 `InitializeLifetimeService` 方法，则生命周期应为租用管理器所设定的值，为永久有效（设置为 0）。那么这两个对象不会被自动 Disconnect，除非我们显式指定关闭它的连接。当然，如果我们显式关闭连接，跟踪程序仍然会监视到它的变化，然后显示出来。

【5】结论

通过我们的测试，其实结论已经很明显了。通过 `Marshal` 编组的对象要受到租用的生命周期所控制。注意对象被 Disconnect，并不是指这个对象被 GC 回收，而是指这个对象保存在通道的相关代理信息被断开了，而对象本身仍然在服务器端存在。

所以我们通过 `Remoting` 提供服务，应根据实际情况指定远程对象的生命周期，如果不

指定，则为 Remoting 默认的设定。要让所有的远程对象永久有效，可以通过配置文件或租用管理器将初始有效时间设为 0。

1.2.3 Remoting 事件处理全接触

前言：在 Remoting 中处理事件其实并不复杂，但其中有些技巧需要你去挖掘出来。正是这些技巧，仿佛森严的壁垒，让许多人望而生畏，或者是不知所谓，最后放弃了事件在 Remoting 的使用。关于这个主题，在网上也有很多讨论，相关的技术文章也不少，遗憾的是，很多文章概述的都不太全面。我在研究 Remoting 的时候，也对事件处理发生了兴趣。经过参考相关的书籍、文档，并经过反复的试验，深信自己能够把这个问题阐述清楚了。本文对于 Remoting 和事件的基础知识不再介绍，有兴趣的可以看我的系列文章，或查阅相关的技术文档。

本文示例代码下载：

Remoting 事件(客户端发传真)

Remoting 事件(服务端广播)

Remoting 事件(服务端广播改进)

应用 Remoting 技术的分布式处理程序，通常包括三部分：远程对象、服务端、客户端。因此从事件的方向上看，就应该有三种形式：

- 1、服务端订阅客户端事件
- 2、客户端订阅服务端事件
- 3、客户端订阅客户端事件

服务端订阅客户端事件，即由客户端发送消息，服务端捕捉该消息，然后响应该事件，相当于下级向上级发传真。反过来，客户端订阅服务端事件，则是由服务端发送消息，此时，所有客户端均捕获该消息，激发事件，相当于是一个系统广播。而客户端订阅客户端事件呢？就类似于聊天了。由某个客户端发出消息，其他客户端捕获该消息，激发事件。可惜的是，我并没有找到私聊的解决办法。当客户端发出消息后，只要订阅了该事件的，都会获得该信息。

然而不管是哪一种方式，究其实质，真正包含事件的还是远程对象。原理很简单，我们想一想，在 Remoting 中，客户端和服务端传递的内容是什么呢？毋庸置疑，是远程对象。因此，我们传递的事件消息，自然是被远程对象所包裹。这就像 EMS 快递，远程对象是运送信件的汽车，而事件消息就是汽车所装载的信件。至于事件传递的方向，只是发送者和订阅者的角色发生了改变而已。

【1】服务端订阅客户端事件

服务端订阅客户端事件，相对比较简单。我们就以发传真为例。首先，我们必须具备传

真机和要传真的文件，这就好比我们的远程对象。而且这个传真机上必须具备“发送”的操作按钮。这就好比是远程对象中的一个委托。当客户发送传真时，就需要在客户端上激活一个发送消息的方法，这就好比我们按了“发送”按钮。消息发送到服务端后，触发事件，这个事件正是服务端订阅的。服务端获得该事件消息后，再处理相关业务。这就好比接收传真的人员，当传真收到后，会听到接通的声音，此时选择“接收”后，该消息就被捕获了。

现在，我们就来模拟这个流程。首先定义远程对象，这个对象处理的应该是一个发送传真的业务：

首先是远程对象的公共接口（Common.dll）：

```
public delegate void FaxEventHandler(string fax);
public interface IFaxBusiness
{
    void SendFax(string fax);
}
```

注意，在公共接口程序集中，定义了一个公共委托。

然后我们定义具体处理传真业务的远程对象类（FaxBusiness.dll），在这个类中，先要添加对公共接口程序集的引用：

```
public class FaxBusiness : MarshalByRefObject, IFaxBusiness
{
    public static event FaxEventHandler FaxSendedEvent;

    #region

    public void SendFax(string fax)
    {
        if (FaxSendedEvent != null)
        {
            FaxSendedEvent(fax);
        }
    }

    #endregion

    public override object InitializeLifetimeService()
    {
        return null;
    }
}
```

这个远程对象中，事件的类型就是我们在公共程序集 Common.dll 中定义的委托类型。

SendFax 实现了接口 IFaxBusiness 中的方法。这个方法的签名和定义的委托一致，它调用了事件 FaxSendedEvent。

特殊的地方是我们定义的远程对象最好是重写 MarshalByRefObject 类的 InitializeLifetimeService()方法。返回 null 值表明这个远程对象的生命周期为无限大。为什么要重写该方法呢？道理不言自明，如果生命周期不进行限制的话，一旦远程对象的生命周期结束，事件就无法激活了。

接下来就是分别实现客户端和服务端了。服务端是一个 Windows 应用程序，界面如下：



我们在加载窗体的时候，注册通道和远程对象：

```
private void ServerForm_Load(object sender, System.EventArgs e)
{
    HttpChannel channel = new HttpChannel(8080);
    ChannelServices.RegisterChannel(channel);

    RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(FaxBusiness), "FaxBusiness.soap", WellKnownObjectMode.Singleton);
    FaxBusiness.FaxSendedEvent += new FaxEventHandler(OnFaxSended);
}
```

我们采用的是 SingleTon 模式，注册了一个远程对象。注意看，这段代码和一般的 Remoting 服务端有什么区别？对了，它多了一行注册事件的代码：

```
FaxBusiness.FaxSendedEvent += new FaxEventHandler(OnFaxSended);
```

这行代码，就好比我们服务端的传真机，一直切换为“自动”模式。它会一直监听着来自客户端的传真信息，一旦传真信息从客户端发过来了，则响应事件方法，即 OnFaxSended 方法：

```
public void OnFaxSended(string fax)
{
    txtFax.Text += fax;
    txtFax.Text += System.Environment.NewLine;
}
```

这个方法很简单，就是把客户端发过来的 Fax 显示到 txtFax 文本框控件上。

而客户端呢？仍然是一个 Windows 应用程序。代码非常简单，首先为了简便起见，我们仍然让它在装载窗体的时候，激活远程对象：

```
private void ClientForm_Load(object sender, System.EventArgs e)
{
    HttpChannel channel = new HttpChannel(0);
    ChannelServices.RegisterChannel(channel);

    faxBus = (IFaxBusiness)Activator.GetObject(typeof(IFaxBusiness),
        "http://localhost:8080/FaxBusiness.soap");
}
```

呵呵，可以说客户端激活对象的方法和普通的 Remoting 客户端应用程序没有什么不同。该写传真了！我们在窗体上放一个文本框对象，改其 Multiline 属性为 true。再放一个按钮，负责发送传真：

```
private void btnSend_Click(object sender, System.EventArgs e)
{
    if (txtFax.Text != String.Empty)
    {
        string fax = "来自" + GetIpAddress() + "客户端的传真:"
        + System.Environment.NewLine;
        fax += txtFax.Text;
        faxBus.SendFax(fax);
    }
    else
    {
        MessageBox.Show("请输入传真内容!");
    }
}

private string GetIpAddress()
{
    IPHostEntry ipHE = Dns.GetHostByName(Dns.GetHostName());
```

```
return ipHE.AddressList[0].ToString();  
}
```

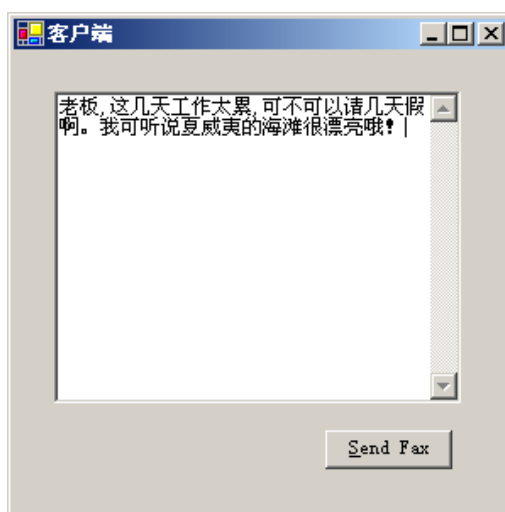
在这个按钮单击事件中，只需要调用远程对象 faxBus 的 SendFax()方法就 OK 了，非常简单。可是慢着，为什么你的代码有这么多行啊？其实，没有什么奇怪的，我只是想到发传真的客户可能会很多。为了避免服务端人员犯糊涂，搞不清楚是谁发的，所以要求在传真上加上各自的签名，也就是客户端的 IP 地址了。既然要获得计算机的 IP 地址，请一定要记得加上对 DNS 的命名空间引用：

```
using System.Net;
```

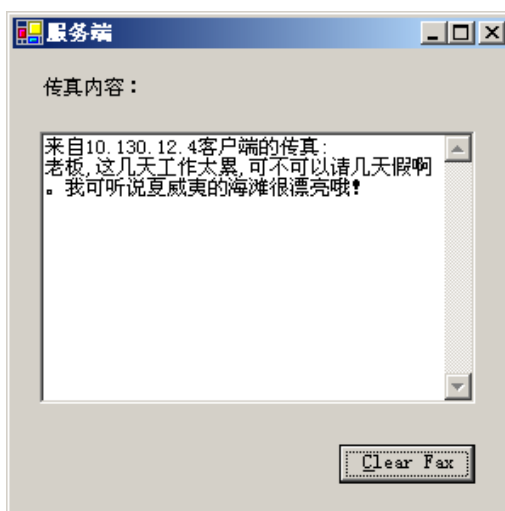
因为我们严格按照分布式处理程序的部署方式，所以在客户端只需要添加公共程序集 (Common.dll) 的引用就可以了。而在服务端呢，则必须添加公共程序集和远程对象程序集两者的引用。

OK，程序完成，我们来看看这个简陋的传真机：

客户端：



嘿嘿，做梦都想放假啊。好的，传真写好了，发送吧！再看看服务端，great，老板已经收到我的请假条传真了！



【2】客户端订阅服务端事件

嘿嘿，吃甘蔗要先吃甜的一段，做事情我也喜欢先做容易的。现在，好日子过去了，该吃点苦头了。我们先回忆一下刚才的实现方法，再来思考怎么实现客户端订阅服务端事件？

在前一节，事件被放到远程对象中，客户端激活对象后，就可以发送消息了。而在服务端，只需要订阅该事件就可以。现在思路应该反过来，由客户端订阅事件，服务端发送消息。就这么简单吗？先不要高兴得太早。我们想一想，发送消息的任务是谁来完成的？是远程对象。而远程对象是什么时候创建的呢？我们仔细思考 **Remoting** 的几种激活方式，不管是服务端激活，还是客户端激活，他们的工作原理都是：客户端决定了服务器创建远程对象实例的时机，例如调用了远程对象的方法。而服务端所作的工作则是注册该远程对象。

回忆这三种激活方式在服务端的代码：

SingleCall 激活方式：

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof(BroadCastObj), "BroadCastMessage.soap",  
    WellKnownObjectMode.Singlecall);
```

SingleTon 激活方式：

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof(BroadCastObj), "BroadCastMessage.soap",  
    WellKnownObjectMode.Singleton);
```

客户端激活方式：

```
RemotingConfiguration.ApplicationName = "BroadCastMessage.soap"  
RemotingConfiguration.RegisterActivatedServiceType(typeof(BroadCastObj));
```

请注意 **Register** 这个词语，它表达的含义就是注册。也就是说，在服务端并没有显示的创建远程对象实例。没有该实例，又如何广播消息呢？

或许有人会想，在注册远程对象之后，显式实例该对象不就可以了吗？也就是说，在注册后加上这一段代码：

```
BroadCastObj obj = new BroadCastObj();
```

然而，我们要明白一个事实：就是服务端和客户端是处于两个不同的应用程序域中。因此在 **Remoting** 中，客户端获得的远程对象实际是服务端注册对象的代理。如果我们在注册后，人工去创建一个实例，而非 **Remoting** 在激活后自动创建的对象，那么客户端获得的对象与服务端人工创建的实例是两个迥然不同的对象。客户端获得的代理对象并没有指向你刚

才创建的 obj 实例。所以 obj 发送的消息，客户端根本无法捕捉。

那么，我们只有望洋兴叹，束手无策了吗？别着急，别忘了在服务器注册对象方法中，还有一种方法，即 Marshal 方法啊。还记得 Marshal 的实现方式吗？

```
BroadCastObj Obj = new BroadCastObj();
ObjRef objRef = RemotingServices.Marshal(Obj,"BroadCastMessage.soap");
```

这个方法与前不一样。前面的三种方式，远程对象是根据客户端调用的方式，来自动创建的。而 Marshal 方法呢？则显式地创建了远程对象实例，然后将其 Marshal 到通道中，形成 ObjRef 指向对象的代理。只要生命周期没有结束，这个对象就一直存在。而此时客户端获得的对象，正是创建的 Obj 实例的代理。

OK，这个问题解决了，我们来看看具体实现。

公共程序集和远程对象与前相似，就不再赘述，只附上代码：

公共程序集：

```
public delegate void BroadCastEventHandler(string info);

public interface IBroadCast
{
    event BroadCastEventHandler BroadCastEvent;
    void BroadCastingInfo(string info);
}
```

远程对象类：

```
public event BroadCastEventHandler BroadCastEvent;

#region IBroadCast 成员
//[OneWay]
public void BroadCastingInfo(string info)
{
    if (BroadCastEvent != null)
    {
        BroadCastEvent(info);
    }
}
#endregion

public override object InitializeLifetimeService()
{
    return null;
}
```


下面，该实现服务端了。在实现之前，我还想罗嗦几句。在第一节中，我们实现了服务端订阅客户端事件。由于订阅事件是在服务端发生的，因此事件本身并未被传送。被序列化的仅仅是传递的消息，即 Fax 而已。现在，方向发生了改变，传送消息的是服务端，客户端订阅了事件。但这个事件是放在远程对象中的，因此事件必须被序列化。而在 .Net Framework 1.1 中，微软对序列化的安全级别进行了限制。有关委托和事件的序列化、反序列化默认是禁止的，所以我们应该将 `TypeFilterLevel` 的属性值设置为 `Full` 枚举值。因此在服务端注册通道的方式就发生了改变：

```
private void StartServer()
{
    BinaryServerFormatterSinkProvider serverProvider = new
        BinaryServerFormatterSinkProvider();
    BinaryClientFormatterSinkProvider clientProvider = new
        BinaryClientFormatterSinkProvider();
    serverProvider.TypeFilterLevel = TypeFilterLevel.Full;

    IDictionary props = new Hashtable();
    props["port"] = 8080;
    HttpChannel channel = new HttpChannel(props, clientProvider, serverProvider);
    ChannelServices.RegisterChannel(channel);

    Obj = new BroadCastObj();
    ObjRef objRef = RemotingServices.Marshal(Obj, "BroadCastMessage.soap");
}
```

注意语句 `serverProvider.TypeFilterLevel = TypeFilterLevel.Full`；此语句即设置序列化安全级别的。要使用 `TypeFilterLevel` 属性，必须申明命名空间：

```
using System.Runtime.Serialization.Formatters;
```

而后面两条语句就是注册远程对象。由于在我的广播程序中，发送广播消息是放在另一个窗口中，因此我将该远程对象声明为公共静态对象：

```
public static BroadCastObj Obj = null;
```

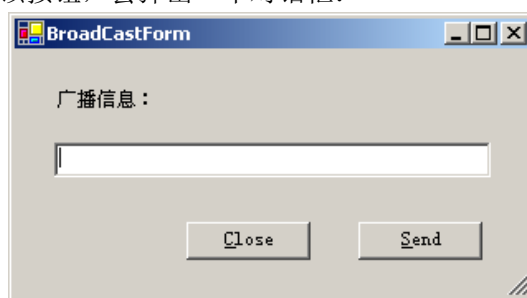
然后在调用窗口事件中加入：

```
private void ServerForm_Load(object sender, System.EventArgs e)
{
    StartServer();
    lbMonitor.Items.Add("Server started!");
}
```

来看看界面，首先启动服务端主窗口：



我放了一个 ListBox 控件来显示一些信息，例如显示服务器启动了。而 BroadCast 按钮就是广播消息的，单击该按钮，会弹出一个对话框：



BraodCast 按钮的代码：

```
private void btnBC_Click(object sender, System.EventArgs e)
{
    BroadCastForm bcForm = new BroadCastForm();
    bcForm.StartPosition = FormStartPosition.CenterParent;
    bcForm.ShowDialog();
}
```

在对话框中，最主要的就是 Send 按钮：

```
if (txtInfo.Text != string.Empty)
{
    ServerForm.Obj.BroadCastingInfo(txtInfo.Text);
}
else
{
    MessageBox.Show("请输入信息！");
}
```

但是很简单，就是调用远程对象的发送消息方法而已。

现在该实现客户端了。我们可以参照前面的例子，只是把服务端改为客户端而已。另外考虑到序列化安全级别的问题，所以代码会是这样：

```
private void ClientForm_Load(object sender, System.EventArgs e)
{
    BinaryServerFormatterSinkProvider serverProvider = new
        BinaryServerFormatterSinkProvider();
    BinaryClientFormatterSinkProvider clientProvider = new
        BinaryClientFormatterSinkProvider();
    serverProvider.TypeFilterLevel = TypeFilterLevel.Full;
    IDictionary props = new Hashtable();
    props["port"] = 0;
    HttpChannel channel = new HttpChannel(props, clientProvider, serverProvider);
    ChannelServices.RegisterChannel(channel);
    watch = (IBroadcast)Activator.GetObject(
        typeof(IBroadcast), "http://localhost:8080/BroadcastMessage.soap");
    watch.BroadcastEvent += new BroadcastEventHandler(BroadcastingMessage);
}
```

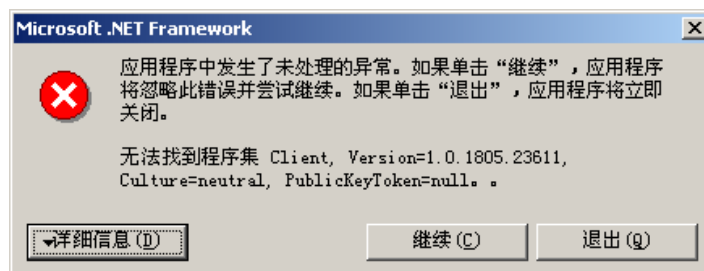
注意客户端通道的端口号应设置为 0，这表示客户端自动选择可用的端口号。如果要设置为指定的端口号，则必须保证与服务端通道的端口号不相同。
然后是，BroadcastEventHandler 委托的方法：

```
public void BroadcastingMessage(string message)
{
    txtMessage.Text += "I got it:" + message;
    txtMessage.Text += System.Environment.NewLine;
}
```

客户端界面如图：



好，下面让我们满怀期盼，来运行这段程序。首先启动服务端应用程序，然后启动客户端。哎呀，糟糕，居然出现了错误信息！



“人之不如意事，十常居八九。”不用沮丧，让我们分析原因。首先看看错误信息，它报告我们没有找到 Client 程序集。然而事实上，Client 程序集当然是有的。那么再来调试一下，是哪一步出现的问题呢？设置好断点，进行逐语句跟踪。前面注册通道一切正常，当运行到 `watch.BroadcastEvent += new BroadcastEventHandler(BroadcastingMessage)` 语句时，错误出现了！

也就是说，远程对象的创建是成功的，但在订阅事件的时候失败了。原因是什么呢？原来，客户端的委托是通过序列化后获得的，在订阅事件的时候，委托试图装载包含与签名相同的方法的程序集，也就是 `BroadcastingMessage` 方法所在的程序集 Client。然而这个装载的过程发生在服务端，而在服务端，并没有 Client 程序集存在，自然就发生了上面的异常。

原因清楚了，怎么解决？首先 `BroadcastingMessage` 方法肯定是在客户端中，所以不可避免，委托装载 Client 程序集的过程也必须在客户端完成。而服务端事件又是由远程对象来捕获的，因此，在客户端注册的也就必须是远程对象事件了。一个要求必须在客户端，一个又要求必须在服务端，事情出现了自相矛盾的地方。

那么，让我们先想想这样一个例子。假设我们要交换 x 和 y 的值，该怎样完成？很简单，引入一个中间变量就可以了。

```
int x=1,y=2,z;
z = x;
x = y;
y = z;
```

这个游戏相信大家都会玩吧，那么好的，我们也需要引入这样一个“中间”对象。这个中间对象和原来的远程对象在事件处理方面，代码完全一致：

```
public class EventWrapper : MarshalByRefObject
{
    public event BroadcastEventHandler LocalBroadcastEvent;

    //[OneWay]
    public void Broadcasting(string message)
    {
```

```

        LocalBroadCastEvent(message);
    }

    public override object InitializeLifetimeService()
    {
        return null;
    }
}

```

不过不同之处在于：这个 **Wrapper** 类必须在客户端和服务端上都要部署，所以，这个类应该放在公共程序集 **Common.dll** 中。

现在再来修改原来的客户端代码：

```

watch = (IBroadCast)Activator.GetObject(
    typeof(IBroadCast),"http://localhost:8080/BroadCastMessage.soap");
watch.BroadCastEvent += new BroadCastEventHandler(BroadCastingMessage);

```

修改为：

```

watch = (IBroadCast)Activator.GetObject(
    typeof(IBroadCast),"http://localhost:8080/BroadCastMessage.soap");
EventWrapper wrapper = new EventWrapper();
wrapper.LocalBroadCastEvent += new BroadCastEventHandler(BroadCastingMessage);
watch.BroadCastEvent += new BroadCastEventHandler(wrapper.BroadCasting);

```

为什么这样做就可以了呢？也许画一幅图就很容易说明，可惜我的艺术天分实在很糟糕，我希望以后可以改进这一点。还是用文字来说明吧。

前面说，委托要装载 **client** 程序集。现在我们把远程对象委托装载的权利移交给 **EventWrapper**。因为这个类对象是放在客户端的，所以它要装载 **client** 程序集丝毫没有问题。语句：

```

EventWrapper wrapper = new EventWrapper();
wrapper.LocalBroadCastEvent += new BroadCastEventHandler(BroadCastingMessage);

```

实现了这个功能。

不过此时虽然订阅了事件，但事件还是客户端的，没有与服务端联系起来。而服务端的事件是放到远程对象中的，所以，还要订阅事件，这个任务由远程对象 **watch** 来完成。但此时它订阅的不再是 **BroadCastingMessage** 了，而是 **EventWrapper** 的触发事件方法 **BroadCasting**。那么此时委托同样要装载程序集，但此时装载的就是 **BroadCasting** 所在的程序集了。由于装载发生的地点是在服务端。呵呵，高兴的是，**BroadCasting** 所在的程序集正

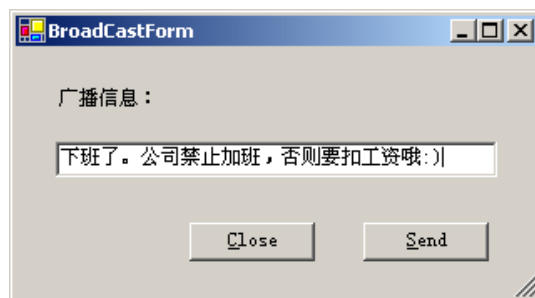
是公共程序集（前面已说过，EventWrapper 应放到公共程序集 Common.dll 中），而公共程序集在服务端和客户端都已经部署了。自然就不会出现找不到程序集的问题了。

注意：EventWrapper 因为要重写 InitializeLifetimeService() 方法，所以仍然要继承 MarshalByRefObject 类。

现在再来运行程序。首先运行服务端；然后运行客户端，OK，客户端窗体出现了：



然后我们在服务端单击“BroadCast”按钮，发送广播消息：



单击“Send”发送，再来看看客户端，会是怎样？Fine，I got it!



怎么样，很酷吧！你也可以同时打开多个客户端，它们都将收到这个广播信息。如果你觉得这个广播声音太吵，那就请你在客户端取消广播吧。在 **Cancle** 按钮中：

```
private void btnCancle_Click(object sender, System.EventArgs e)
{
    watch.BroadcastEvent -= new BroadcastEventHandler(wrapper.Broadcasting);
    MessageBox.Show("取消订阅广播成功!");
}
```

当然这个时候 wrapper 对象应该被申明为 private 对象了：

```
private EventWrapper wrapper = null;
```



取消后，你试着再广播一下，恭喜你，你不会听到噪音了！

【3】客户端订阅客户端事件

有了前面的基础，再来看客户端订阅客户端事件，就简单多了。而本文写到这里，我也很累了，你也被我啰嗦得不耐烦了。你心里在喊，“饶了我吧！”其实，我又何尝不是如此。所以我只提供一个思路，有兴趣的朋友，可以自己写一个程序。

其实方法很简单，和第二种情况类似。发送信息的客户端，只需要获得远程对象后，发送消息就可以了。而接收信息的客户端，负责订阅该事件。由于事件都是放到远程对象中，因此订阅的方法和第二种情况没有什么区别！

特殊的情况是，我们可以用第三种情况来代替第二种。只要你把发送信息的客户端放到服务端就可以了。当然需要做一些额外的工作，有兴趣的朋友可以去实现一下。在我的示例程序中，已经用这种方法模拟实现了服务端的广播，大家可以去看看。

【4】一点补充

我在前面的事件处理中，使用的都是默认的 EventArgs。如果要定义自己的 EventArgs，就不相同了。因为该信息是传值序列化，因此必须加上 [Serializable]，且必须放到公共程序集中，部署到服务端和客户端。例如：

```
[Serializable]
public class BroadcastEventArgs : EventArgs
```

```

{
    private string msg = null;

    public BroadcastEventArgs(string message)
    {
        msg = message;
    }

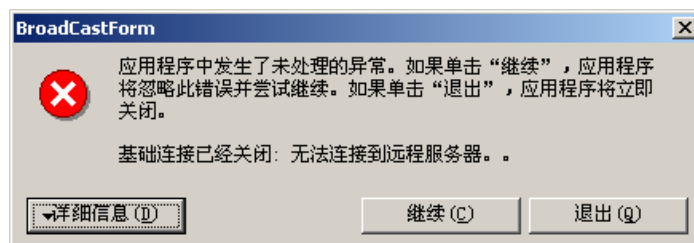
    public string Message
    {
        get { return msg; }
    }
}

```

【5】持续改进

也许，细心的读者注意到了，在我的远程对象类和 EventWrapper 类中，触发事件方法的 Attribute[OneWay]被我注释掉了。我看到很多资料上写到，在 Remoting 中处理事件，触发事件的方法必须具有这个 Attribute。这个 attribute 究竟有什么用？

在发送事件消息的时候，事件的订阅者会触发事件，然后响应该事件。然而当事件的订阅者发生错误的时候呢？例如，发送事件消息的时候，才发现根本没有事件订阅者；或者事件的订阅者出现故障，如断电、或异常关机。此时，发送事件一方会因为找不到正确的事件订阅者，而发生异常。以我的程序为例。当我们分别打开服务端和客户端程序的时候，此时广播信息正常。然而，当我们关闭客户端后，由于该客户端没有取消订阅，此时异常发生，提示信息如图：



（不知道为什么，这个异常与客户端连接服务端出现的异常一样。这个异常容易让人产生误会。）

如果这个时候我们同时打开了多个客户端，那么其他客户端就会因为这一个客户端关闭造成的错误，而无法收到广播信息。那么让我们先做第一步改进：

1) 先考虑正常情况。在我的客户端，虽然提供了取消订阅的操作，但并没有考虑用户关闭客户端的情况。即，关闭客户端时，并未取消事件的订阅，所以我们应该在关闭客户端窗体中写入：

```

private void ClientForm_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    watch.BroadcastEvent -= new BroadcastEventHandler(wrapper.Broadcasting);
}

```


2) 仅仅是这样还不够。如果客户端并没有正常关闭，而是因为突然断电而导致客户端关闭呢？此时，客户端还没有来得及取消事件订阅呢。在这种情况下，我们需要用到 `OneWayAttribute`。

前面说到，发送事件一方如果找不到正确的事件订阅者，会发生异常。也就是说，这个事件是 `unreachable` 的。幸运的是，`OneWayAttribute` 恰好解决了这个问题。其实从该特性的命名 `OneWay`，大约也能猜到其中的含义。当事件不可到达，无法发送时，正常情况下，会返回一个异常信息。如果加上 `OneWayAttribute`，这个事件的发送就变成单向的了。假如此时发生异常，那么系统会自动抛掉该异常信息。由于没有异常信息的返回，发送信息方会认为发送信息成功了。程序会正常运行，错误的客户端被忽略，而正确的客户端仍然能够收到广播信息。

因此，远程对象的代码就应该是这样：

```
public event BroadcastEventHandler BroadcastEvent;
#region IBroadcast 成员
//[OneWay]
public void BroadcastingInfo(string info)
{
    if (BroadcastEvent != null)
    {
        BroadcastEvent(info);
    }
}
#endregion
public override object InitializeLifetimeService()
{
    return null;
}
```

3) 最后的改进

使用 `OneWay` 固然可以解决上述的问题，但不够友好。因为对于广播消息的一方来说，象被蒙上了眼睛一样，对于客户端发生的事情懵然不知。这并不是一个好的 `idea`。在 Ingo Rammer 的 `Advanced .NET Remoting` 一书中，Ingo Rammer 先生提出了一个更好的办法，就是在发送信息一方时，检查了委托链。并在委托链的遍历中来捕获异常。当其中一个委托发生异常时，显示提示信息。然后继续遍历后面的委托，这样既保证了异常信息的提示，又保证了其他订阅者正常接收消息。因此，我对本例的远程对象进行了修改，注释掉 `[OneWay]`，修改了 `BroadcastInfo()` 方法：

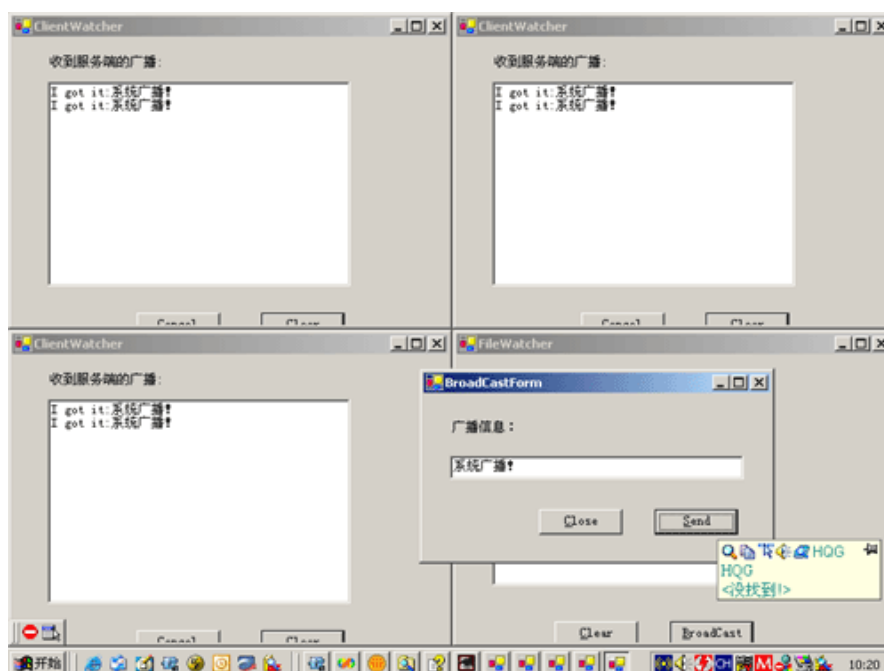
```
//[OneWay]
public void BroadcastingInfo(string info)
{
```

```

if (BroadCastEvent != null)
{
    BroadCastEventHandler tempEvent = null;
    int index = 1; //记录事件订阅者委托的索引，为方便标识，从1开始。
    foreach (Delegate del in BroadCastEvent.GetInvocationList())
    {
        try
        {
            tempEvent = (BroadCastEventHandler)del;
            tempEvent(info);
        }
        catch
        {
            MessageBox.Show("事件订阅者" + index.ToString() + "发生错误,系统将取消事件订阅!");
            BroadCastEvent -= tempEvent;
        }
        index++;
    }
}
else
{
    MessageBox.Show("事件未被订阅或订阅发生错误!");
}
}

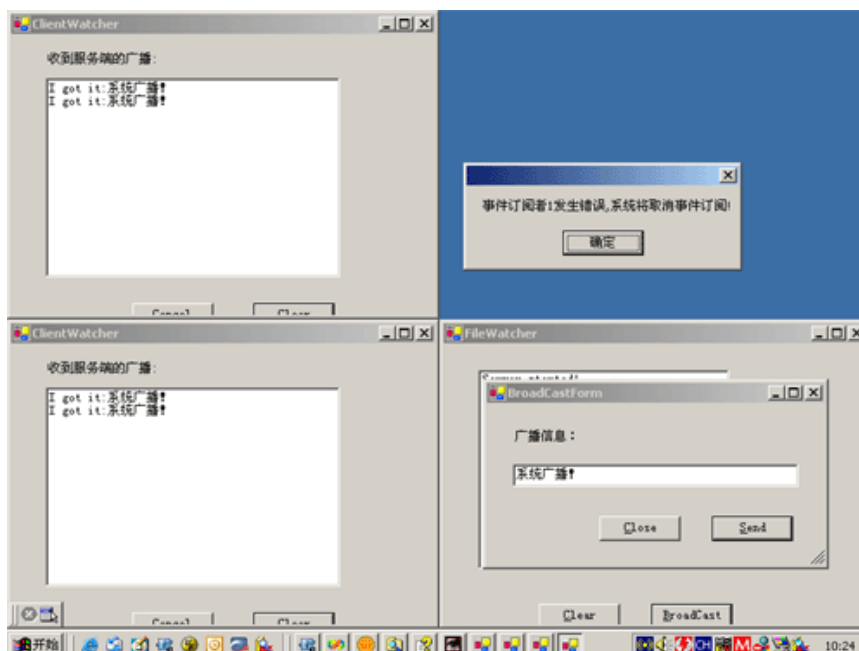
```

我们来试验一下。首先打开服务端，然后同时打开三个客户端。广播消息：



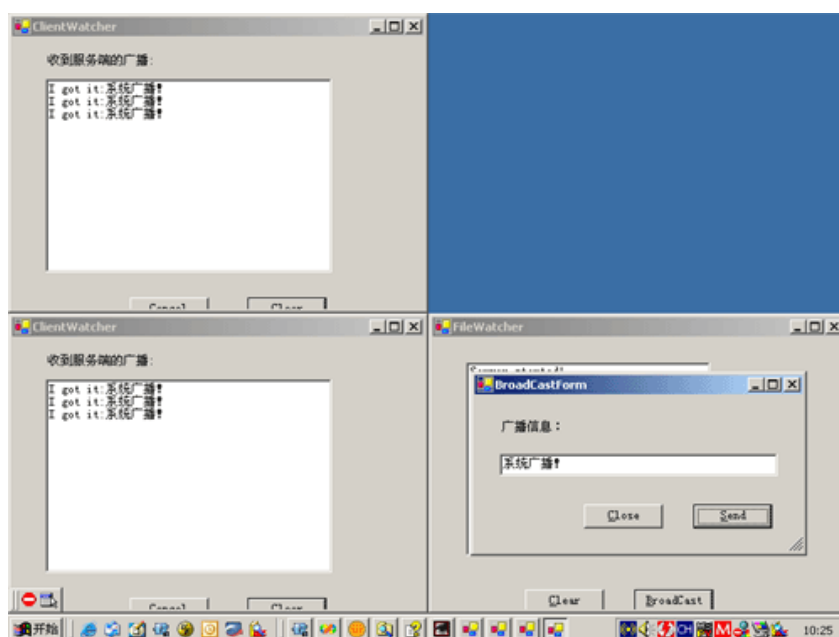
消息发送正常。

接着关闭其中一个客户端窗口，再广播消息（注意为模拟客户端异常情况，应在 ClientForm_Closing 方法中把第一步改进的取消订阅代码注释。否则不会发生异常。难道你真的愿意用断电来导致异常发生吗^_^），结果如图：



此时服务端报告了“事件订阅者 1 发生错误，系统将取消事件订阅”。注意此时另外两个客户端，还是和前面一样，只有两条广播信息。

当我们点击提示框的“确定”按钮后，广播仍然发送：



通过这样的改进后，程序更加的完善，也更加的健壮和友好！

1.2.4 关于Remoting

这几天看了不少 Remoting 文章。明白了不少技术细节，但困惑也不少。简单说来，Remoting 是一个分布式处理服务。服务器端首先创建通道（Channel），并自动开启监听通道。根据客户端发出的请求，传递远程对象。

因此，编写 Remoting 程序，主要分为三部分：

- 1、被传递的远程对象；
- 2、服务器端监听程序；
- 3、客户端请求和处理对象程序；

【1】被传递的远程对象

在 Remoting 中，被传递的远程对象类是有诸多限制的。首先，我们必须清楚，这里所谓的传递是以引用的方式，因此所传递的远程对象类必须继承 MarshalByRefObject。MarshalByRefObject 是那些通过使用代理交换消息来跨越应用程序域边界进行通信的对象的基类。不是从 MarshalByRefObject 继承的对象会以隐式方式按值封送。当远程应用程序引用一个按值封送的对象时，将跨越远程处理边界传递该对象的副本。因为您希望使用代理方法而不是副本方法进行通信，因此需要继承 MarshalByRefObject。（MSDN）

```
public class ServerObject : MarshalByRefObject
{
    public Person GetPersonInfo(string name, string sex, int age)
    {
        Person person = new Person();
        person.Name = name;
        person.Sex = sex;
        person.Age = age;
        return person;
    }
}
```

这个类只实现了最简单的方法，就是设置一个人的基本信息，并返回一个 Person 类对象。值得注意的是，这里返回的 Person 类。由于是以引用和远程调用的方式。这里所传递的 Person 则是以传值的方式来完成。因此必须涉及到一个序列化的问题。

所以，Remoting 要求对象类还要调用或传递某个对象，例如类，或者结构，则该类或结构则必须实现串行化 Attribute。[Serializable]。

[Serializable]

```
public class Person
{
    public Person()
    {

    }

    private string name;
    private string sex;
    private int age;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string Sex
    {
        get { return sex; }
        set { sex = value; }
    }

    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

这个服务器对象，以类库的方式编译成 Dll，这个工作就算完成了。

那么这个对象是怎么实现被客户端和服务端调用的呢？这就是下面我们必须要做的工作：将编译后的 DLL 分别添加到服务器端和客户端程序的引用中。也就是说，这个服务器对象 Dll 要拷贝两份，一份放在服务器端，一份放在客户端。为什么要这样？看了后面的代码就知道原因了。

如此一来，会有个问题存在。那就是代码的安全性。如果客户端必须要保持这个对象的 Dll，则该对象的实现方式对于客户而言就近乎透明了。另外，这样对于部署也不好。两份同样的 dll，如果传递的对象大，也会影响性能的。对于这个问题，我们可以使用接口来解决。显然，服务器端提供接口和具体类的实现，而客户端则只需要接口就可以了（**不过，对于客户端激活模式则必须有实现接口的类**）。

```
public interface IServerObject
```

```
{
    Person GetPersonInfo(string name, string sex, int age);
}

public class ServerObject: MarshalByRefObject, IServerObject
```

要注意的是：

- 1、两边生成该对象程序集的名字必须相同，严格地说，是命名空间的名字必须相同。
- 2、这种方式根据激活方式的不同，实现也不同。如果是服务器端激活（SingleTon 和 SingCall），那很简单。如上所述的方法即可；

如果是客户端激活，最好是利用抽象工厂，提供创建实例的方法。这样就必须在代码中还要加上抽象工厂的接口及实现：

```
public interface IServerObjFactory
{
    IServerObject CreateInstance();
}

public class ServerObject : MarshalByRefObject, IServerObject, IServerObjFactory
{
    public IServerObject CreateInstance()
    {
        return new ServerObject();
    }
}
```

但是由于客户端激活的方式，它必须调用类的构造函数来创建实例，因此，在客户端只实现接口似乎是不可能的。

说明：关于对象类继承 MarshalByRefObject，我作过测试，使可以间接地继承的。也就是我们可以先通过实现基类来继承它。然后实际所传递的对象在从基类中派生。

最后的代码应该是这样（服务器端，如果是客户端，只需要接口即可。这里我加了抽象工厂，因此该对象应该是以客户端激活模式。如果是服务器端激活模式，应该把抽象工厂接口和实现方法去掉）

```
using System;

namespace ServerRemoteObject
{
    [Serializable]
    public class Person
```

```
{

    public Person()
    {

    }

    private string name;
    private string sex;
    private int age;

    public string Name
    {
        get {return name;}
        set {name = value;}
    }

    public string Sex
    {
        get {return sex;}
        set {sex = value;}
    }

    public int Age
    {
        get {return age;}
        set {age = value;}
    }
}

public interface IServerObject
{
    Person GetPersonInfo(string name, string sex, int age);
}

public interface IServerObjFactory
{
    IServerObject CreateInstance();
}

public class ServerObject:MarshalByRefObject, IServerObject
{
    public Person GetPersonInfo(string name, string sex, int age)
    {
        Person person = new Person();
    }
}
```

```
        person.Name = name;
        person.Sex = sex;
        person.Age = age;
        return person;
    }
}

public class ServerObjFactory:MarshalByRefObject, IServerObjFactory
{
    public IServerObject CreateInstance()
    {
        return new ServerObject();
    }
}
```

要补充说明的是，这里传递的对象显然比 Socket 更多，它甚至可以传递 DataSet 对象。其实，我们可以将它理解为 Webservice。

【2】服务器端监听程序

写到这里，我想先介绍一下远程对象的三种激活模式。激活模式分为两大类：服务器端激活和客户端激活。其中服务器端激活又分为：Singleton 和 SingleCall 两种。

1) 服务器端激活，又叫做 WellKnow 方式。我不想从理论上来讲述，而只是按照自己的理解来解释。简单地说，Singleton 激活方式，是对所有用户都建立一个对象实例，不管这些用户是在同一客户端还是不同客户端。而 SingleCall 则是对客户端的每个用户都建立一个远程对象实例。至于对象实例的销毁则是由 GC 自动管理的。举例来说，如果远程对象的一个累加方法 (i=0; ++i) 被多个客户端（例如两个）调用。如果是 Singleton 方式，则第一个客户获得值为 1，第二个客户获得值为 2，因为他们获得的对象实例是同样的。而 SingleCall 方式，则两个客户获得的都是 1。原因不言自明。

2) 客户端激活。则是对每个客户端都建立一个实例。粗看起来和 SingleCall 相同。实际上是有区别的。首先，对象实例创建的时间不一样。客户端激活方式是客户端一旦发出调用的请求，就实例化；而 SingleCall 则是要等到调用对象方法时在创建。其次，WellKnow 方式对对象的管理是由 GC 管理的，而客户端则可以自定义生命周期，管理他的销毁。其三，WellKnow 对象是无状态的，客户端激活对象则是有状态的。当然具体到使用上来说，实现的方式也就不一样了。

好了，现在我们就开始创建服务器端监听程序。

Remoting 传递远程对象实质上来说还是通过 Socket 来传递，因此必须有一个传递对象的通道 (Channel)。一个通道必须提供一个端口。在 Remoting 中，通道是由 IChannel 接口

提供。它分别有两种类型：**TcpChannel** 和 **HttpChannel**。**Tcp** 是以二进制的方式来传递，**Http** 则是以 **Soap** 的方式来传递。两种通道各有优势，从性能上看，**Tcp** 更好。但 **Http** 可以更好地通过防火墙。因此用户可以根据自己情况选择使用通道的方式。（本文使用 **TcpChannel**，事实上两种可以混合使用，现创建 **TcpChannel**，如果连接失败，在创建 **HttpChannel**。）通道实现的类为同名类，命名空间则是在 **System.Runtime.Remoting.Channel** 下。通过通道可以传递对象，而且可以一次传递多个对象。对象的传递和选择激活方式，是通过 **RemotingConfiguratin** 的静态方法 **RegisterWellKnownServiceType()**（针对服务器激活模式）和 **RegisterActivatedServiceType()**（针对客户端激活模式）来实现的。代码如下：

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace ServerRemoting
{
    /**////<summary>
    /// Class1 的摘要说明。
    /// </summary>
    class Server
    {
        /**////<summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //创建通道，使用 8080 端口；
            TcpChannel channel = new TcpChannel(8080);

            //注册通道；
            ChannelServices.RegisterChannel(channel);

            //传递对象；
            RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(ServerRemoteObject.ServerObject),
                "ServiceMessage", WellKnownObjectMode.SingleCall);

            Console.WriteLine("Open the server listener");
            Console.ReadLine();
        }
    }
}
```

代码很简单。我是用控制台来提供服务的。这里重点说一样 `RegisterWellKnownServiceType()` 方法的参数。第一个参数就是要传递对象的类型。第二个参数是自己定义的远程对象服务名，其实它是作为客户端的 `Uri` 的一部分。第三个参数自然是定义激活的方式。`WellKnownObjectMode` 是一个枚举，有 `SingleCall` 和 `SingleTon` 两个。

如果是客户端激活模式，稍有不同：

```
RemotingConfiguration.ApplicationName = "ServiceMessage";
RemotingConfiguration.RegisterActivatedServiceType(
    typeof(ServerRemoteObject.ServerObject));
```

几点说明：

1) Dll 的引用。要添加第一步所创建的远程对象 Dll；添加 `System.Runtime.Remoting` 的引用。

2) 关于一个通道传递几个对象。其实没什么复杂的，再接着用 `RegisterWellKnownServiceType()` 方法就是了。只要这个对象符合我前面所讲的要求，同时添加了引用。当然客户端也要增加相应的代码。

3) 关于多个通道的使用。在 `Remoting` 中，可以同时使用多个通道。但要求通道名必须不同。默认建立的 `TcpChannel` 名字为 `Tcp`，`HttpChannel` 名字为 `Http`。如果再要建立一个 `TcpChannel`，则必须自己定义一个名字。`Channel` 本身提供 `ChannelName` 字段，但该字段是只读的。所以方法有点变化，要使用 `HashTable` 和 `IDictionary`(要添加 `System.Collection` 命名空间)：

```
IDictionary channelProps = new Hashtable();
channelProps["name"] = "MyTCP";
channelProps["priority"] = "1";
channelProps["Port"] = "8081";

TcpChannel channel2 = new TcpChannel(channelProps,
    new SoapClientFormatterSinkProvider(),
    new SoapServerFormatterSinkProvider());
```

4) 服务的停止。对于通道的使用，主要是用于传递远程对象，并开启对通道的监听。因此我们可以关闭对其的监听。也可以直接注销掉该通道。关闭监听使用通道实例对象的 `StopListening()` 方法实现。注销通道则是使用 `ChannelServices.UnregisterChannel()` 方法。关闭监听并没有注销掉通道，只是关闭了对客户端请求的监听。还可以通过 `StartListening()` 方法来重新打开监听。而通道一旦被注销，则需要重新使用如前所述的注册通道的方法重新注册。

```
//获得当前已注册的通道;
IChannel[] channels = ChannelServices.RegisteredChannels;

//关闭指定名为 MyTcp 的通道;
```

```

foreach (IChannel eachChannel in channels)
{
    if (eachChannel.ChannelName == "MyTcp")
    {
        TcpChannel tcpChannel = (TcpChannel)eachChannel;

        //关闭监听;
        tcpChannel.StopListening(null);

        //注销通道;
        ChannelServices.UnregisterChannel(tcpChannel);
    }
}

```

【3】客户端请求和处理对象程序

先看代码：

```

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace ClientRemoting
{
    /**/// <summary>
    /// Class1 的摘要说明。
    /// </summary>
    class Client
    {
        /**/// <summary>
        /// 应用程序的主入口点。
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            TcpChannel channel = new TcpChannel();
            ChannelServices.RegisterChannel(channel);

            ServerRemoteObject.IServerObject serverObj = (ServerRemoteObject.IServerObject)
                Activator.GetObject(
                    typeof(ServerRemoteObject.IServerObject),
                    "tcp://localhost:8080/ServiceMessage");
        }
    }
}

```

```

        Console.WriteLine("Invoke remoting object:");
        ServerRemoteObject.Person person = serverObj.GetPersonInfo("wayfarer", "male", 28);

        Console.WriteLine("name: {0}, sex: {1}, age: {2}", person.Name, person.Sex, person.Age);
        Console.ReadLine();
    }
}
}

```

代码非常简单。注意在客户端实例化通道时，是调用的默认构造函数，即没有传递端口号。事实上，这个端口号是缺一不可的，只不过它的指定被放在后面作为 Uri 的一部分。这个程序重点是这一行代码：

```

ServerRemoteObject.IServerObject serverObj = (ServerRemoteObject.IServerObject)
    Activator.GetObject(
        typeof(ServerRemoteObject.IServerObject),
        "tcp://localhost:8080/ServiceMessage");

```

首先以 WellKnown 模式激活，客户端获得对象的方法是使用 GetObject()。其中参数第一个是远程对象的类型。如前所述，我们在客户端只用了接口，因此返回的对象应该是接口类型。第二个参数就是服务器端的 uri。如果是 http 通道，自然是用 http://localhost:8080/ServiceMessage 了。因为我是用本地机，所以这里是 localhost，你可以用具体的服务器 IP 地址来代替它。端口必须和服务器端的端口一致。后面则是服务器定义的远程对象服务名。

如果服务器端采用客户端激活模式，调用的方法就不一样了。顺便说一句，上面的实现同样添加了远程对象的 Dll 引用。我前面已经讲过，这个 Dll 的命名空间名必须和服务器端的一致。因为我们的实现是在同一机器上完成的，所以大家在建立这个客户端远程对象时，要换个路径来建立这个对象类库。

还是回到刚才的问题，采用工厂，那么代码应稍作修改，因为远程对象不是直接实例化，而是通过工厂来创建的。显然，我们只需要改变上面那一行代码：

```

RemotingConfiguration.RegisterActivatedClientType(
    typeof(ServerRemoteObject.IServerObjFactory),
    "tcp://localhost:8080/ServiceMessage");
ServerRemoteObject.IServerObjFactory factory = new ServerRemoteObject.ServerObjFactory();

ServerRemoteObject.IServerObject serverObj = factory.CreateInstance();

```

即使是利用抽象工厂，如果是客户端激活模式，那么在客户端引用的远程对象还是要有类的具体实现，而不是只由接口的实现。从上面的代码就知道了，它获得对象实例的方法不是用 Activator.GetObject() 方法来获得对象实例，而是利用

RemotingConfiguration.RegisterActivatedClientType()静态方法去注册一个对象，然后再调用类的构造函数。

怎样像 WellKnown 模式那种，只在客户端实现接口，我还没有想到实现的方法。

通过上面的三个步骤，Remoting 的程序也就完成了。注意运行的时候，需要先运行服务器程序，再运行客户端程序。

【4】程序的完善

上面写的程序，关于服务器 uri、端口、以及激活模式的设置是用代码来完成的。其实我们也可以用配置文件来设置。这样做有个好处，因为这个配置文件是 Xml 文档。如果需要改变端口或其他，我们就不需要修改程序，并重新编译，而是只需要改变这个配置文件即可。

1) 服务器端的配置文件:

```
<configuration>
  <system.runtime.remoting>
    <application name="ServerRemoting">
      <service>
        <wellknown mode="Singleton" type="ServerRemoteObject.ServerObject"
          objectUri="ServiceMessage"/>
      </service>
      <channels>
        <channel ref="tcp" port="8080"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

如果是客户端激活模式，则把 wellknown 改为 activated，同时删除 mode 属性。

把该配置文件放到服务器程序的应用程序文件夹中，命名为 ServerRemoting.config。那么前面的服务器端程序直接用这条语句即可：

```
RemotingConfiguration.Configure("ServerRemoting.config");
```

2) 客户端配置文件

```
<configuration>
  <system.runtime.remoting>
    <application name="ClientRemoting">
      <client url="tcp://localhost:8080/ServerRemoting">
        <wellknown type="ServerRemoteObject.ServerObject">
```

```
objectUri="tcp://localhost:8080/ServerRemoting/ServiceMessage"/>
</client>
<channels>
  <channel ref="tcp" port="8080"/>
</channels>
</application>
</system.runtime.remoting>
</configuration>
```

如果是客户端激活模式，修改和上面一样。调用也是使用 `RemotingConfiguration.Configure()` 方法来调用存储在客户端的配置文件。

配置文件还可以放在 `machine.config` 中。如果客户端程序是 web 应用程序，则可以放在 `web.config` 中。

1.2.5 关于 Remoting（续）

昨天写了文章《关于 Remoting》，感觉有些问题没有说清楚。后来又看了一些文档和书，整理了一下，就算是续吧。

其实我发现主要的问题还是集中在客户端激活模式。我想再谈谈客户端激活模式和服务器端激活模式两者在代码实现上的区别。这两种模式在服务器监听程序上的区别不大，前面那篇文章已经说得很清楚了，主要还是客户端程序。为了让概念不至于模糊混淆，我下面提到客户端激活模式，用 `Activated`；服务器激活模式，用 `WellKnown`。

先从 VS 提供的方法来看：

WellKnown 模式：`Activator.GetObject()` 方法。它的返回值是方法参数里指定类型的对象实例。

```
ServerRemoteObject.IServerObject serverObj = (ServerRemoteObject.IServerObject)
    Activator.GetObject(
        typeof(ServerRemoteObject.IServerObject),
        "tcp://localhost:8080/ServiceMessage");
```

Activated 模式：

有两种方法：

1) 静态方法：`RemotingConfiguration.RegisterActivatedClientType()`。这个方法返回值为 `Void`，它只是将远程对象注册在客户端而已。具体的实例化还需要调用对象类的构造函数。

```
RemotingConfiguration.RegisterActivatedClientType(
    typeof(ServerRemoteObject.ServerObject),
    "tcp://localhost:8080/ServiceMessage");
```

```
ServerRemoteObject.ServerObject serverObj = new ServerRemoteObject.ServerObject();
```

2) `Activator.CreateInstance()`方法。这个方法将创建方法参数指定类型的类对象。它与前面的 `GetObject()`不同的是,它要在客户端调用构造函数,而 `GetObject()`只是获得对象,而创建实例是在服务器端完成的。`CreateInstance()`方法有很多个重载,我着重说一下其中常用的两个。

(1) `public static object CreateInstance(Type type, object[] args, object[] activationAttributes);`
参数说明:

type: 要创建的对象类型。

args : 与要调用构造函数的参数数量、顺序和类型匹配的参数数组。如果 **args** 为空数组或空引用(Visual Basic 中为 `Nothing`),则调用不带任何参数的构造函数(默认构造函数)。

activationAttributes : 包含一个或多个可以参与激活的属性的数组。

这里的参数 **args** 是一个 `object[]`数组类型。它可以传递要创建对象的构造函数中的参数。从这里其实可以得到一个结论: **WellKnown** 激活模式所传递的远程对象类,只能使用默认的构造函数;而 **Activated** 模式则可以用户自定义构造函数。

activationAttributes 参数在这个方法中通常用来传递服务器的 url。
假设我们的远程对象类 `ServerObject` 有个构造函数:

```
ServerObject(string pName, string pSex, int pAge)
{
    name = pName;
    sex = pSex;
    age = pAge;
}
```

那么实现的代码是:

```
object[] attrs = {new UriAttribute("tcp://localhost:8080/ServiceMessage")};
object[] objs = new object[3];
objs[0] = "wayfarer";
objs[1] = "male";
objs[2] = 28;
ServerRemoteObject.ServerObject = Activator.CreateInstance(
    typeof(ServerRemoteObject.ServerObject),
    objs, attrs);
```

可以看到, `objs[]`数组传递的就是构造函数的参数。

2) `public static ObjectHandle CreateInstance(string assemblyName, string typeName, object[] activationAttribute);`

参数说明：

assemblyName

将在其中查找名为 **typeName** 的类型的程序集的名称。如果 **assemblyName** 为空引用（Visual Basic 中为 **Nothing**），则搜索正在执行的程序集。

typeName

首选类型的名称。

activationAttributes

包含一个或多个可以参与激活的属性的数组。

参数说明一目了然。注意这个方法返回值为 **ObjectHandle** 类型，因此代码与前不同：

```
object[] attrs = { new UriAttribute("tcp://localhost:8080/EchoMessage") };
ObjectHandle handle = Activator.CreateInstance("ServerRemoteObject",
                                                "ServerRemoteObject.ServerObject", attrs);
ServerRemoteObject.ServerObject obj = (ServerRemoteObject.ServerObject)handle.Unwrap();
```

那么，这个方法实际上是调用的默认构造函数。**ObjectHandle.Unwrap()**方法是返回被包装的对象。

说明：要使用 **UriAttribute**，还需要在命名空间中添加：**using System.Runtime.Remoting.Activation;**

通过这些代码的比较，我们还可以得到一个不幸的结论：

对于 **Activated** 激活模式，不管是使用静态方法，还是使用 **CreateInstance()**方法，都必须在客户端调用构造函数实例化对象。这样一来，在客户端我们提供的远程对象，就不可能只提供接口，而没有类的实现。而在 **WellKnown** 模式，因为在客户端只是用 **GetObject()**获得对象，实例化是在服务器端完成的。所以客户端我们只提供接口就够了。

所以对于 **Activated** 模式，我们必须在服务器和客户端提供两份完全相同的远程对象 **Dll**，这个结果确实让人很沮丧。有没有其他方法实现呢？鉴于它的实现原理，答案显然是否定的。我看了 **MSDN** 上的文章，唯一的可行方案就是利用前文提到的用工厂的方法。要注意的是：这种方法是一种利用 **WellKnown** 模式来模拟 **Activated** 模式，是一种方法的折中。

前文说过，服务器端远程对象，提供两个接口。一个接口是具体要传递的远程对象的接口，一个就是工厂接口。还必须有个工厂类实现工厂接口，提供创建远程对象实例的方法。

```
public interface IServerObject
{
    Person GetPersonInfo(string name, string sex, int age);
}

public interface IServerObjFactory
```



```
{
    IServerObject CreateInstance();
}

public class ServerObject : MarshalByRefObject, IServerObject
{
    public Person GetPersonInfo(string name, string sex, int age)
    {
        Person person = new Person();
        person.Name = name;
        person.Sex = sex;
        person.Age = age;
        return person;
    }
}

public class ServerObjFactory : MarshalByRefObject, IServerObjFactory
{
    public IServerObject CreateInstance()
    {
        return new ServerObject();
    }
}
```

而客户端呢，只提供接口就可以了。

```
public interface IServerObject
{
    Person GetPersonInfo(string name, string sex, int age);
}

public interface IServerObjFactory
{
    IServerObject CreateInstance();
}
```

然后用 WellKnown 激活模式，在服务器端：

```
//传递对象:
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(ServerRemoteObject.ServerObjFactory),
    "ServiceMessage", WellKnownObjectMode.SingleCall);
```

注意这里注册的不是 `ServerObject` 类对象，而是 `ServerObjFactory` 类对象。

客户端：

```
ServerRemoteObject.IServerObjFactory serverFactory = (ServerRemoteObject.IServerObjFactory)
    Activator.GetObject(
        typeof(ServerRemoteObject.IServerObjFactory),
        "tcp://localhost:8080/ServiceMessage");

ServerRemoteObject.IServerObject serverObj = serverFactory.CreateInstance();
```

首先用 `GetObject()` 返回工厂接口，再利用该接口对象调用工厂的方法 `CreateInstance()` 来创建 `IServerObject` 对象。

也许有人会纳闷了，这里使用的是 `WellKnown` 模式啊，为什么说是 `Activated` 模式呢？是这样的，本质来说，它是 `WellKnown` 模式。但由于我们利用了工厂来创建实例，因此感觉上是在客户端来创建具体的远程对象实例。因此，我说这是一种方法的折中。

好了，写到这里，自认为交待清楚了。但 `Remoting` 的实现远不止于此，还有很多高级复杂的东西我们还没有用到。例如自定义通道，自定义 `MarshalByReferenceObject` 派生类，生命周期管理，自定义代理。这些东西我也弄不清楚。如果以后弄明白了，希望能写点东西。

1.2.6 关于 `Remoting` 一些更改

随着对 `Remoting` 的逐步了解，很多技术在实现上会有一些变化，起初肤浅的认识会逐渐扎实起来。而自己以前在文中的很多结论会被自己不断的推翻。没有改变是不会有进步的，我喜欢这种改变！

我在《关于 `Remoting`(续)》中这样写到：

对于 `Activated` 激活模式，不管是使用静态方法，还是使用 `CreateInstance()` 方法，都必须在客户端调用构造函数实例化对象。这样一来，在客户端我们提供的远程对象，就不可能只提供接口，而没有类的实现。

目前看来这个结论还是正确的。但是紧接着的结论就未免有些武断了。

所以对于 `Activated` 模式，我们必须在服务器和客户端提供两份完全相同的远程对象 `Dll`，这个结果确实让人很沮丧。

真的是这样吗？其实这里我们可以用一个 `trick`，来欺骗 `Remoting`。既然是提供服务，`Remoting` 传递的远程对象其实现的细节当然是放在服务器端。而要在客户端放对象的副本，不过是因为客户端必须调用构造函数，而采取的无奈之举。既然具体的实现是在服务器端，

又为了能在客户端实例化，那么在客户端就实现这些好了。至于实现的细节，就不用管了。

那么远程对象有方法，服务器端提供方法实现，客户端就提供这个方法就 OK 了，至于里面的实现，你可以是抛出一个异常，或者 `return` 一个 `null` 值；如果方法返回 `void`，那么里面可以是空。关键是这个客户端类对象要有这个方法。这个方法的实现，其实和方法的声明差不多，所以我说是一个 `trick`。方法如是，构造函数也如此。

还是用代码来说明这种“阴谋”，更直观：

服务器端：

```
public class ServerObject : MarshalByRefObject
{
    public ServerObject()
    {
    }

    public Person GetPersonInfo(string name, string sex, int age)
    {
        Person person = new Person();
        person.Name = name;
        person.Sex = sex;
        person.Age = age;
        return person;
    }
}
```

客户端：

```
public class ServerObject:MarshalByRefObject
{
    public ServerObject()
    {
        throw new System.NotImplementedException();
    }

    public Person GetPersonInfo(string name, string sex, int age)
    {
        throw new System.NotImplementedException();
    }
}
```

比较客户端和服务端，客户端的方法 `GetPersonInfo()`，没有具体的实现细节，只是抛

出了一个异常。或者直接写上语句 `return null`，照样 OK。
我们称客户端的这个类为远程对象的替代类。

在《[Remoting 的几个疑惑](#)》文章因为 `cls` 的帮助，解决了这个困惑。但今天又有新的困惑了。问题真是层出不穷啊。相信不断地提出问题，又不断地解决问题，学习 `Remoting` 最终可以满师吧？

这些困惑在《[Remoting 的几个疑惑](#)》文后的评论已经写了出来，这里就不再啰嗦了。

1.2.7 Remoting 的几个疑惑

写完《关于 `Remoting`》之后，算是把这几天学习 `Remoting` 的思路理了一下。`Remoting` 的基本知识是很简单的，但一旦深入，就会发现博大精深。昨天看到微软社区的一贴广告，说台湾的某个 MVP 写了一本书，是专门讲 IIS 的安全知识的。于是想到，如果要把微软产品的每一项功能去穷尽，可能每个看似很小的模块都能写成一本大部头书吧。`Remoting` 也是如此，要把每个细节都弄清楚，谈何容易。

我之学习 `Remoting` 是带着目的而来的，起因还是公司要做的项目。最初在 `WebService` 和 `Remoting` 之间权衡，最后因为项目主要应用在局域网中，而 `Remoting` 在局域网内的性能优势是 `WebService` 不可比肩的。所以选定了它。`Remoting` 是一个分布式处理服务。我们的想法是要在服务器上部署多个服务，而客户端则接收服务处理具体的业务。在 `Remoting` 中，我们可以简单地将它要传送的远程对象看成是一个服务。接收服务，就是实例化这个远程对象，再调用其方法就 OK 了。

看似简单，但问题也接踵而至。首先，我们在服务器端提供的服务是可以定制的。这里所谓定制，即管理员可以在服务器端关闭或启动指定的服务，同时可以查看其当前状态。也就是说，要提供类似 windows 操作系统下的“服务”系统。然而这就是问题的关键。Windows 服务中，每一项服务就是一个进程，关闭该服务，其实就是关闭其进程。而在 `Remoting` 中，我们也可以将其通道看作一个进程，一个通道又占用一个端口。

设想一下，如果我们要提供多个服务，且将一个服务当作一个进程的话，在 `Remoting` 中，就将占用多个通道和端口了。产生的问题是：

- 1、端口占用过多，是否会影响系统的性能？
- 2、不能只使用 80 端口。那么如果使用了其他端口，怎样通过防火墙？

如果大家还不太清楚这个问题，可以设想 `WebService`。在 `WebService` 中，每提供一个服务，都将在 IIS 下创建一个虚拟目录来指向它。我仔细看了 IIS 的 SDK，不管是用编程的方式，还是直接在操作系统下操作 IIS，都没有关闭一个虚拟目录的功能，除非将其删掉。也就是说，要关闭这个指定的 Web 服务是不行的。唯一的方法就是关闭其 Web 站点。默认情况下，你建立了多个 Web 服务，都会放在默认的 web 站点下。这样，你关闭了 Web 站点，事实上就关闭了所有的服务。这与 `Remoting` 何其相似！`Remoting` 中，你也能够关闭通道，

那么这个通道所承载的所有远程对象（即服务）也就被关闭了。

好，那么我们就退而求其次吧，就用关闭通道的方式来停止服务。我们有两个方案来选择。一是并不关闭通道，而是关闭其对客户端的监听，方法是 `StopListening()`。这样有个好处，就是通道仍然存在，一旦你需要使用，还可以开启通道，方法是 `StartListening`（`Remoting` 中，一旦注册了通道，默认就开启了监听）。第二个方案是注销通道，方法是 `UnRegisterChannel()`。一旦注销了该通道，通道就不能使用了，而通道所占用的端口也将被释放。

这两个方案都很有用。如果我们要暂停服务，可以使用第一种，它便于我们重启。如果要彻底停止服务，可以使用第二种。看来问题以一种妥协的方式解决了！？遗憾的是，我们在做测试的时候，发现这两种关闭通道的方法是由延迟的。当我们在客户端关闭通道后，客户端调用远程对象方法，发现仍然可以正常使用。起初我们怀疑是激活方式的问题。因为我们最初采用的是 `SingleCall` 方式，它是无状态的。但改为客户端激活方式，仍然如此。为对象加上生命周期，还是照旧。

那么什么时候它才真正关闭呢？老实说我不知道。在测试时，我们大约过了一分钟，再在客户端调用远程对象方法，这时才出现异常，提示目标不能正常连接。

即时这个方案能实现，仍然不符合我们要求的。我们理想的答案，还是希望能操作每个指定的对象。了解了一些 `Remoting` 的知识，发现 `Remoting` 的一个通道注册多个远程对象，它的管理方式是把这些远程对象放在一个哈希表中，然后为每个远程对象指定一个唯一标识，作为哈希表的键值。当客户端发出请求后，可以根据这个键值来找到请求的对象，然后实例化它。我们看通道注册对象的方法，`RegisterWellknownServiceType()`，这个 `Register` 大概就是将其添加到这个哈希表中吧。遗憾的是，没有找到对应的 `UnRegister()` 方法。因为我们的想法是，如果能够把对象从这个哈希表中移出，客户端发出请求时，自然找不到该对象，此时会抛出异常，不就等同于关闭了该服务吗？没有 `UnRegister()` 方法，我也没有能找到得到这个哈希表的方法，那么这个 `Idea` 也只能夭折了。真是不甘心啊！

总结我的疑惑就是：

- 1、怎样将已注册的对象从通道中注销（或移出）？
- 2、退而求其次的方案，那么为什么关闭了监听或关闭了通道，会有一个时间上的延迟？延迟又是多少？

1.2.8 Remoting 疑惑续集

上次写了一篇《关于 `Remoting` 的疑惑》，结果令人满意，解决了一个难题。但今天的问题有些奇怪，归结原因，还是对 `Remoting` 的内在机制不甚了解。

问题如下：

Remoting 传递远程对象是通过通道来传递的，而每个通道将占用一个端口。要在服务端提供远程对象的实现，首先要 **Register** 通道，然后将对象 **Marshal**。如果要停止该远程对象，再通过 **Disconnect**。但该通道仍然存在。如果要停止通道，必须通过 **Unregister** 来完成。此时注销了通道，应该释放了对该端口的占用。

如果没有注销通道，又再次注册通道，则会抛出异常“通常每个套接字地址（协议/网络地址/端口）只允许使用一次。”因为在 **Remoting** 里，要求通道的名字必须是唯一的，且每个通道只能对应一个端口。

在我的服务器端正是按照如上的规则来实现的。运行服务器端程序，注册通道->注销通道->注册通道，一切 OK！为了测试 **Remoting** 的性能，再运行客户端程序。在服务器端注册通道并 **Marshal** 远程对象的情况下，在客户端程序调用该远程对象，运行正常，该对象被激活并能调用对象的方法。此时注销通道（在服务器端，我的代码在注销通道前，已经将该通道下的所有远程对象 **Disconnect** 了），然后再注册通道，问题出现，系统抛出了异常“通常每个套接字地址（协议/网络地址/端口）只允许使用一次。”

请注意看，在服务器端上的操作是完全一致的：注册通道->注销通道->注册通道。唯一不同的是在注册通道和注销通道之间，客户端调用了远程对象。

疑问：按道理 **RemotingServices.UnregisterChannel()**方法在注销了通道后，应该释放了对该端口的占用。但从目前的情况来看，当客户端调用了远程对象后，该对象的代理信息保存在通道里，此时即使注销了通道，但由于通道保留了未被销毁的信息，因此端口仍然在占用中。是这样吗？

分析：上述分析显然错误。因为我在注销通道前，已经通过 **Disconnect()**方法断开了该对象和通道的连接，此时远程对象的代理信息在通道中应该不存在了。如果此时通过客户端去调用该对象方法，会抛出异常“未找到指定的服务”，事实正是如此！如果该代理信息仍然存在的话，是不会抛出异常的。那么是对象的生命周期在作怪吗？我将租用管理器的 **InitialLeaseTime** 和 **RenewOnCallTime** 分别设置为 1 分钟和 20 秒，再调用远程对象。然后注销服务器端通道，过两分钟后，再注册，仍然如此。仔细分析，确实不应该是生命周期的原因，因为所谓租用时间到期，即是要 **Disconnect** 对象。而我在 **Unregister** 时，已经指定停止该对象了。

那么是否因为客户端调用远程对象的缘故，即使注销了通道，由于该服务器对象仍然存在，则客户端与服务器端的连接仍然存在，所以该端口还被占用呢？然而当我注销了通道后，再在客户端调用远程对象时，得到的却是异常，报告我“基础连接已经关闭：无法连接到远程服务器。”这说明这个连接已经不存在了啊。

那么究竟原因何在呢？

补充：首先说明，上面在服务器提供远程对象的方式为 **Marshal** 和 **Disconnect** 方式。如果我采用 **RemotingConfiguration** 的静态方法 **RegisterWellKnownServiceType()** 方法以 **SingleTon** 模式激活对象。那么又是怎样呢？

先注册通道，然后注册远程对象。这时启动客户端程序，调用远程对象方法，此时服务器会激活该对象，调用成功。然后在服务器端注销该通道后，通过客户端调用远程对象方法。由于注销通道有一定延迟，因此此时调用仍然是成功的。也就是说该对象通过通道与服务器的连接仍然是存在的！现在再一次注销服务器端通道，可是此时操作是成功的。

这就完全推翻了我前面的猜测。也就是说注销通道后的再注册通道应该与远程对象无关。那么使用 Marshal 和 Disconnet 方式，为何会出现这个问题？

唉，头都大了。

1.2.9 Remoting疑惑续集之再续

如果各位不知道我的疑惑是什么，请看《Remoting 疑惑续集》。经过我和 CSL 的讨论，明白了一些问题，大家可以看文章的评论，也可参考《A Big Problem about the Microsoft .Net Remoting》。最近这段时间，由于要做其他事情，暂时抛开了 Remoting。前几天我在改我的 Remoting 服务管理器时，又根据以前讨论后的结果再做了测试，结果发现一些问题。

根据 MSDN 的官方文档，ChannelServices.UnregisterChannel(channel)，在注销通道的同时，已经释放了通道占用的端口，分析 System.Runtime.Remoting 的代码(《A Big Problem about the Microsoft .Net Remoting》有简单地分析)，也可以看到这点。但不知道是否是 Remoting 的 bug，一旦客户端调用了该通道提供的远程对象后，UnregisterChannel()方法事实上并没有释放掉端口的占用。

虽然 CSL 说 UnregisterChannel()方法，会默认调用 StopListening()关闭监听。但我还是想尝试一下显式调用的方法。最初我考虑到 Remoting 底层的通讯采用的是 Socket，所以我想调用 TcpListener 类的 Stop()方法来关闭对通道的调用。

```
TcpListener listener = new TcpListener(ipaddress, port);  
listener.Stop();
```

这里的 IP 地址和端口号，都是 Remoting 所使用的设置。我把这段代码放到注销通道的方法前。现在我按照原来出现的问题进行测试。首先建立 Remoting 服务，Marshal 远程对象。然后在客户端调用该对象。此时注销通道，紧接着立刻注册通道。OK，没有出现“**通常每个套接字地址 (协议/网络地址/端口)只允许使用一次。**”异常。似乎这个方案是可行的了。

由于用这个方法，必须在构造函数中指定 Remoting 要用的 ip 地址和端口号，很难和我原来的程序整合起来，所以我还想试验其他方法。既然 StopListening()方法最终还是调用 TcpListener 的 Stop()方法，那么直接显式调用它，是否也可行呢？

我在注销通道的方法中作了如下修改：

```
IChannel[] channels = ChannelServices.RegisteredChannels;
```



```
foreach (IChannel channel in channels)
{
    switch (channelType)
    {
        case ChannelType.Http:
            if (channel.ChannelName.ToLower().IndexOf("http") >= 0)
            {
                //关闭和释放资源:
                ((HttpChannel)channel).StopListening(null);

                ChannelServices.UnregisterChannel(channel);
            }
            break;
        case ChannelType.Tcp:
            if (channel.ChannelName.ToLower().IndexOf("tcp") >= 0)
            {
                ((TcpChannel)channel).StopListening(null);

                ChannelServices.UnregisterChannel(channel);
            }
            break;
    }
}
```

说明一下：**ChannelType** 是我定义的通道类型枚举。然后我在注册通道的时候，根据通道类型，分别为通道取名为“通道类型+端口号”，例如 **Tcp** 通道调用 9000 端口，则该通道名为：**Tcp9000**。这段代码是通过 **RegisteredChannels** 属性获得已经注册的通道，然后判断通道名，以获得通道的类型，并进行强制转换，进而调用 **StopListening()** 方法：
((TcpChannel)channel).StopListening(null);

再测试后，果然是正确的。

写到这里似乎文章该结束了，而且根据讨论的情况，其实应该在上一篇文章就该结束我这啰嗦的疑惑了。不过为了稳妥起见，我还是不断地测试，以保证程序的健壮性。终于在我测试了 **n** 次后，无意发现了大的 **Bug**。

我在程序中，是将 **Remoting** 的远程对象分别分为 **Http** 和 **Tcp** 两种通道类型来发布的。客户端在调用远程对象时，根据通道类型，选择 **Http** 或者 **Tcp** 组成正确的 **uri** 来调用。测试时，我在客户端调用 **Tcp** 通道的远程对象，然后注销 **Tcp** 通道，接着再次注册，**No problem!** 此时我再注销 **Http** 通道，接着再次注册，异常令人讨厌的出现了。反之，我在客户端调用 **Http** 通道的远程对象，注销和注册 **Http** 通道就没有问题，而对 **Tcp** 通道进行操作，仍然会抛出异常。

奇怪吧！确实够奇怪的。不过主要还是我的需求奇怪。首先，我的需求是：能够启动和停止指定远程对象，能够提供 Tcp 通道和 Http 通道两种类型，能够提供注册/注销通道的功能。

先分析我的实现方式：

1、首先注册通道：

```
//如果没有注册该通道,则注册;否则,跳过;
if (!IsRegistered(channelType, port))
{
    switch (channelType)
    {
        case ChannelType.Tcp:
            //根据端口号和通道类型注册通道;
            IDictionary tcpProp = new Hashtable();
            tcpProp["name"] = "tcp" + port.ToString();
            tcpProp["port"] = port;
            channel = new TcpChannel(tcpProp,
                new BinaryClientFormatterSinkProvider(),
                new BinaryServerFormatterSinkProvider());

            ChannelServices.RegisterChannel(channel);
            break;
        case ChannelType.Http:
            IDictionary httpProp = new Hashtable();
            httpProp["name"] = "http" + port.ToString();
            httpProp["port"] = port;
            channel = new HttpChannel(httpProp,
                new SoapClientFormatterSinkProvider(),
                new SoapServerFormatterSinkProvider());

            ChannelServices.RegisterChannel(channel);

            break;
    }
}
```

2、发布远程对象：

1) 利用反射创建对象：

```
//动态加载服务对象的 dll;
```

```
string dllPath = info.MainAssembly.AssemblyPath;
Assembly assembly = Assembly.LoadFrom(dllPath);
string typeFullName = null;

typeFullName = nameSpace+"."+mainClass;

Type type = assembly.GetType(typeFullName);

//动态创建服务对象;
Object obj = Activator.CreateInstance(type);
```

2) Marshal 该对象:

```
ObjRef objRef = RemotingServices.Marshal((MarshalByRefObject)obj, "Service");
```

3、停止该服务对象:

```
public void StopService(Object obj)
{
    try
    {
        RemotingServices.Disconnect((MarshalByRefObject)obj);
    }
    catch
    {
        throw new Exception("停止指定服务错误");
    }
}
```

4、注销通道（如前）

另外，我在注销的时候，关闭通道之前，先停止了该远程对象的连接。（为简便起见，代码作了相应修改）

我们注意到，发布对象时，Remoting 并没有方法决定该对象具体发布到哪个通道中。经过我的分析，事实上对对象进行 Marshal 后，Remoting 当前的所有通道都存在该远程对象。这也许就是问题症结。

即使如此，仍然让我困惑的是，当我调用 Tcp 通道的远程对象后，为什么注销/注册 Http 通道会出现异常，而 Tcp 通道则不存在问题呢？因为我在注销 Http 通道时，既停止了 Http 通道内的服务对象，又停止了对通道的监听，然后再注销了该通道。即使调用 Tcp 通道的远程对象后，这个对象会驻留在 Http 通道中，经过我的一系列操作，Http 通道的端口应该已经被释放了啊，为什么会抛出异常呢？从工作机制上看，两种类型通道的原理是一样的啊！？

疑惑!!!

1.2.10 基于消息与.Net Remoting的分布式处理架构

分布式处理在大型企业应用系统中，最大的优势是将负载分布。通过多台服务器处理多个任务，以优化整个系统的处理能力和运行效率。分布式处理的技术核心是完成服务与服务之间、服务端与客户端之间的通信。在.Net 1.1 中，可以利用 Web Service 或者.Net Remoting 来实现服务进程之间的通信。本文将介绍一种基于消息的分布式处理架构，利用了.Net Remoting 技术，并参考了 CORBA Naming Service 的处理方式，且定义了一套消息体制，来实现分布式处理。

一、消息的定义

要实现进程间的通信，则通信内容的载体——消息，就必须在服务两端具有统一的消息标准定义。从通信的角度来看，消息可以分为两类：Request Messge 和 Reply Message。为简便起见，这两类消息可以采用同样的结构。

消息的主体包括 ID, Name 和 Body，我们可以定义如下的接口方法，来获得消息主体的相关属性：

```
public interface IMessage : ICloneable
{

    IMessageItemSequence GetMessageBody();
    string GetMessageID();
    string GetMessageName();

    void SetMessageBody(IMessageItemSequence aMessageBody);
    void SetMessageID(string aID);
    void SetMessageName(string aName);
}
```

消息主体类 Message 实现了 IMessage 接口。在该类中，消息体 Body 为 IMessageItemSequence 类型。这个类型用于 Get 和 Set 消息的内容：Value 和 Item：

```
public interface IMessageItemSequence : ICloneable
{

    IMessageItem GetItem(string aName);
    void SetItem(string aName, IMessageItem aMessageItem);

    string GetValue(string aName);
}
```

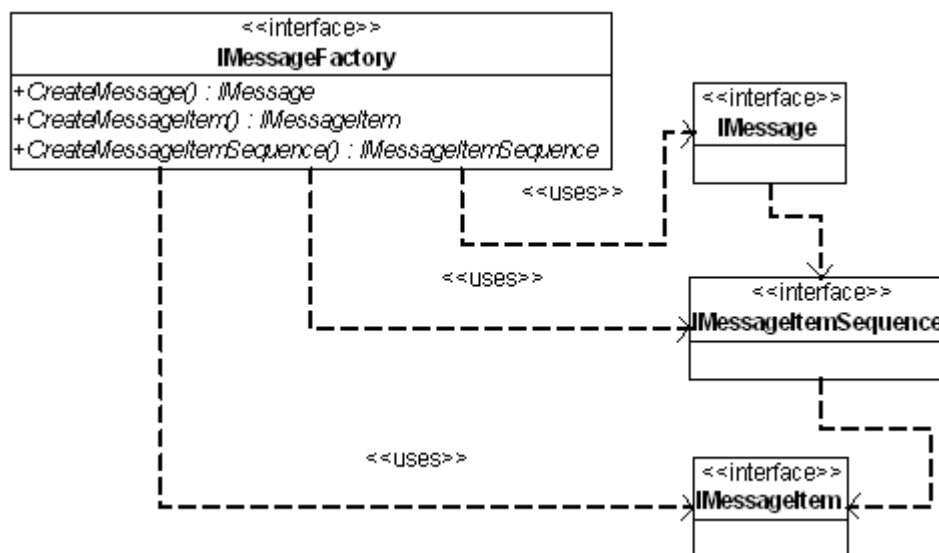
```
void SetValue(string aName, string aValue);
}
```

Value 为 string 类型，并利用 HashTable 来存储 Key 和 Value 的键值对。而 Item 则为 IMessageItem 类型，同样的在 IMessageItemSequence 的实现类中，利用 HashTable 存储了 Key 和 Item 的键值对。

IMessageItem 支持了消息体的嵌套。它包含了两部分：SubValue 和 SubItem。实现的方式和 IMessageItemSequence 相似。定义这样的嵌套结构，使得消息的扩展成为可能。一般的结构如下：

```
IMessage——Name
      ——ID
      ——Body (IMessageItemSequence)
            ——Value
            ——Item (IMessageItem)
                  ——SubValue
                  ——SubItem (IMessageItem)
                  ——.....
```

各个消息对象之间的关系如下：



在实现服务进程通信之前，我们必须定义好各个服务或各个业务的消息格式。通过消息体的方法在服务的一端设置消息的值，然后发送，并在服务的另一端获得这些值。例如发送消息端定义如下的消息体：

```
IMessageFactory factory = new MessageFactory();
IMessageItemSequence body = factory.CreateMessageItemSequence();
body.SetValue("name1", "value1");
```

```

body.SetValue("name2", "value2");

IMessageItem item = factory.CreateMessageItem();
item.SetSubValue("subname1", "subvalue1");
item.SetSubValue("subname2", "subvalue2");

IMessageItem subItem1 = factory.CreateMessageItem();
subItem1.SetSubValue("subsubname11", "subsubvalue11");
subItem1.SetSubValue("subsubname12", "subsubvalue12");
IMessageItem subItem2 = factory.CreateMessageItem();
subItem1.SetSubValue("subsubname21", "subsubvalue21");
subItem1.SetSubValue("subsubname22", "subsubvalue22");

item.SetSubItem("subitem1", subItem1);
item.SetSubItem("subitem2", subItem2);

body.SetItem("item", item);

//Send Request Message
MyServiceClient service = new MyServiceClient("Client");
IMessageItemSequence reply = service.SendRequest("TestService", "Test1", body);

```

在接收消息端就可以通过获得 body 的消息体内容，进行相关业务的处理。

二、.Net Remoting 服务

在 .Net 中要实现进程间的通信，主要是应用 Remoting 技术。根据前面对消息的定义可知，实际上服务的实现，可以认为是对消息的处理。因此，我们可以对服务进行抽象，定义接口 IService:

```

public interface IService
{
    IMessage Execute(IMessage aMessage);
}

```

Execute()方法接受一条 Request Message，对其进行处理后，返回一条 Reply Message。在整个分布式处理架构中，可以认为所有的服务均实现该接口。但受到 Remoting 技术的限制，如果要想实现服务，则该服务类必须继承自 MarshalByRefObject，同时必须在服务端被 Marshal。随着服务类的增多，必然要在服务两端都要对这些服务的信息进行管理，这加大了系统实现的难度与管理的开销。如果我们从另外一个角度来分析服务的性质，基于消息处理而言，所有服务均是对 Request Message 的处理。我们完全可以定义一个 Request 服务负责此消息的处理。

然而，Request 服务处理消息的方式虽然一致，但毕竟服务实现的业务，即对消息处理

的具体实现，却是不相同的。对我们要实现的服务，可以分为两大类：业务服务与 Request 服务。实现的过程为：首先，具体的业务服务向 Request 服务发出 Request 请求，Request 服务侦听到该请求，然后交由其侦听的服务来具体处理。

业务服务均具有发出 Request 请求的能力，且这些服务均被 Request 服务所侦听，因此我们可以为业务服务抽象出接口 IListenService：

```
public interface IListenService
{
    IMessage OnRequest(IMessage aMessage);
}
```

Request 服务实现了 IService 接口，并包含 IListenService 类型对象的委派，以执行 OnRequest()方法：

```
public class RequestListener : MarshalByRefObject, IService
{
    public RequestListener(IListenService listenService)
    {
        m_ListenService = listenService;
    }

    private IListenService m_ListenService;

    #region IService Members

    public IMessage Execute(IMessage aMessage)
    {
        return this.m_ListenService.OnRequest(aMessage);
    }

    #endregion

    public override object InitializeLifetimeService()
    {
        return null;
    }
}
```

在 RequestListener 服务中，继承了 MarshalByRefObject 类，同时实现了 IService 接口。通过该类的构造函数，接收 IListenService 对象。

由于 Request 消息均由 Request 服务即 RequestListener 处理，因此，业务服务的类均应包含一个 RequestListener 的委派，唯一的区别是其服务名不相同。业务服务类实现 IListenService 接口，但不需要继承 MarshalByRefObject，因为被 Marshal 的是该业务服务内

部的 RequestListener 对象，而非业务服务本身：

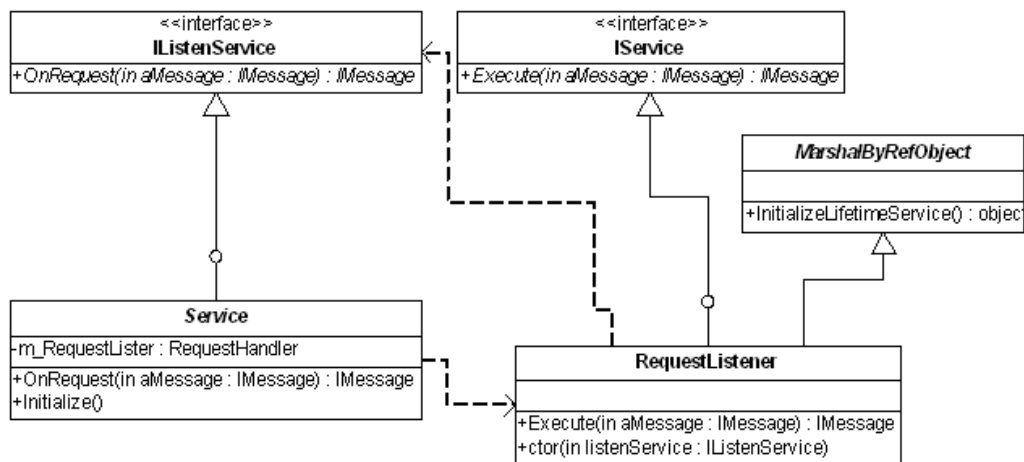
```
public abstract class Service : IListenService
{
    public Service(string serviceName)
    {
        m_ServiceName = serviceName;
        m_RequestListener = new RequestListener(this);
    }

    #region IListenService Members
    public IMessage OnRequest(IMessage aMessage)
    {
        //.....
    }

    #endregion

    private string m_ServiceName;
    private RequestListener m_RequestListener;
}
```

Service 类是一个抽象类，所有的业务服务均继承自该类。最后的服务架构如下：



我们还需要在 Service 类中定义发送 Request 消息的行为，通过它，才能使业务服务被 RequestListener 所侦听。

```
public IMessageItemSequence SendRequest(string aServiceName, string aMessageName,
IMessageItemSequence aMessageBody)
{

```

```

    IMessage message = m_Factory.CreateMessage();
    message.SetMessageName(aMessageName);
    message.SetMessageID("");
    message.SetMessageBody(aMessageBody);

    IService service = FindService(aServiceName);
    IMessageItemSequence replyBody = m_Factory.CreateMessageItemSequence();
    if (service != null)
    {
        IMessage replyMessage = service.Execute(message);
        replyBody = replyMessage.GetMessageBody();
    }
    else
    {
        replyBody.SetValue("result", "Failure");
    }
    return replyBody;
}

```

注意 `SendRequest()`方法的定义，其参数包括服务名，消息名和被发送的消息主体。而在实现中最关键的一点是 `FindService()`方法。我们要查找的服务正是与之对应的 `RequestListener` 服务。不过，在此之前，我们还需要先将服务 `Marshal`：

```

public void Initialize()
{
    RemotingServices.Marshal(this.m_RequestListener, this.m_ServiceName +
        ".RequestListener");
}

```

我们 `Marshal` 的对象，是业务服务中的 `Request` 服务对象 `m_RequestListener`，这个对象在 `Service` 的构造函数中被实例化：

```
m_RequestListener = new RequestListener(this);
```

注意，在实例化的时候是将 `this` 作为 `IService` 对象传递给 `RequestListener`。因此，此时被 `Marshal` 的服务对象，保留了业务服务本身即 `Service` 的指引。可以看出，在 `Service` 和 `RequestListener` 之间，采用了“双重委派”的机制。

通过调用 `Initialize()`方法，初始化了一个服务对象，其类型为 `RequestListener`（或 `IService`），其服务名为：`Service` 的服务名 + `".RequestListener"`。而该服务正是我们在 `SendRequest()`方法中要查找的 `Service`：

```
IService service = FindService(aServiceName);
```


下面我们来看看 FindService()方法的实现:

```
protected IService FindService(string aServiceName)
{
    lock (this.m_Services)
    {
        IService service = (IService)m_Services[aServiceName];
        if (service != null)
        {
            return service;
        }
        else
        {
            IService tmpService = GetService(aServiceName);
            AddService(aServiceName, tmpService);
            return tmpService;
        }
    }
}
```

可以看到, 这个服务是被添加到 m_Service 对象中, 该对象为 SortedList 类型, 服务名为 Key, IService 对象为 Value。如果没有找到, 则通过私有方法 GetService()来获得:

```
private IService GetService(string aServiceName)
{
    IService service = (IService)Activator.GetObject(typeof(RequestListener),
        "tcp://localhost:9090/" + aServiceName + ".RequestListener");
    return service;
}
```

在这里, Channel、IP、Port 应该从配置文件中获取, 为简便起见, 这里直接赋为常量。

再分析 SendRequest 方法, 在找到对应的服务后, 执行了 IService 的 Execute()方法。此时的 IService 为 RequestListener, 而从前面对 RequestListener 的定义可知, Execute()方法执行的其实是其侦听的业务服务的 OnRequest()方法。

我们可以定义一个具体的业务服务类, 来分析整个消息传递的过程。该类继承于 Service 抽象类:

```
public class MyService : Service
{
    public MyService(string aServiceName)
        : base(aServiceName)
    { }
}
```

```
}

```

假设把进程分为服务端和客户端，那么对消息处理的步骤如下：

- 1、 在客户端调用 MyService 的 SendRequest()方法发送 Request 消息；
- 2、 查找被 Marshal 的服务，即 RequestListener 对象，此时该对象应包含对应的业务服务对象 MyService；
- 3、 在服务端调用 RequestListener 的 Execute()方法。该方法则调用业务服务 MyService 的 OnRequest()方法。

在这些步骤中，除了第一步在客户端执行外，其他的步骤均是在服务端进行。

三、业务服务对于消息的处理

前面实现的服务架构，已经较为完整地实现了分布式的服务处理。但目前的实现，并未体现对消息的处理。我认为，对消息的处理，等价与具体的业务处理。这些业务逻辑必然是在服务端完成。每个服务可能会处理单个业务，也可能会处理多个业务。并且，服务与服务之间仍然存在通信，某个服务在处理业务时，可能需要另一个服务的业务行为。也就是说，每一种类的消息，处理的方式均有所不同，而这些消息的唯一标识，则是在 SendRequest()方法已经有所体现的 aMessageName。

虽然，处理的消息不同，所需要的服务不同，但是根据我们对消息的定义，我们仍然可以将这些消息处理机制抽象为一个统一的格式；在 .Net 中，体现这种机制的莫过于委托 delegate。我们可以定义这样的一个委托：

```
public delegate void RequestHandler(string aMessageName, IMessageItemSequence aMessageBody, ref
IMessageItemSequence aReplyMessageBody);
```

在 RequestHandler 委托中，它代表了这样一族方法：接收三个入参，aMessageName，aMessageBody，aReplyMessageBody，返回值为 void。其中，aMessageName 代表了消息名，它是消息的唯一标识；aMessageBody 是待处理消息的主体，业务所需要的所有数据都存储在 aMessageBody 对象中。aReplyMessageBody 是一个引用对象，它存储了消息处理后的返回结果，通常情况下，我们可以用<"result","Success">或<"result","Failure">来代表处理的结果是成功还是失败。

这些委托均在服务初始化时被添加到服务类的 SortedList 对象中，键值为 aMessageName。所以我们可以抽象类中定义如下方法：

```
protected abstract void AddRequestHandlers();
protected void AddRequestHandler(string aMessageName, RequestHandler handler)
{
    lock (this.m_EventHandlers)
    {
```

```

        if (!this.m_EventHandlers.Contains(aMessageName))
        {
            this.m_EventHandlers.Add(aMessageName, handler);
        }
    }
}

protected RequestHandler FindRequestHandler(string aMessageName)
{
    lock (this.m_EventHandlers)
    {
        RequestHandler handler = (RequestHandler)m_EventHandlers[aMessageName];
        return handler;
    }
}

```

AddRequestHandler() 用于添加委托对象与 aMessageName 的键值对，而 FindRequestHandler() 方法用于查找该委托对象。而抽象方法 AddRequestHandlers() 则留给 Service 的子类实现，简单的实现如 MyService 的 AddRequestHandlers() 方法：

```

public class MyService : Service
{
    public MyService(string aServiceName)
        : base(aServiceName)
    { }

    protected override void AddRequestHandlers()
    {
        this.AddRequestHandler("Test1", new RequestHandler(Test1));
        this.AddRequestHandler("Test2", new RequestHandler(Test2));
    }

    private void Test1(string aMessageName, IMessageItemSequence aMessageBody, ref
IMessageItemSequence aReplyMessageBody)
    {
        Console.WriteLine("MessageName: {0} \n", aMessageName);
        Console.WriteLine("MessageBody: {0} \n", aMessageBody);
        aReplyMessageBody.SetValue("result", "Success");
    }

    private void Test2(string aMessageName, IMessageItemSequence aMessageBody, ref
IMessageItemSequence aReplyMessageBody)
    {
        Console.WriteLine("Test2" + aMessageBody.ToString());
    }
}

```

```
}
}
```

Test1 和 Test2 方法均为匹配 RequestHandler 委托签名的方法，然后在 AddRequestHandlers() 方法中，通过调用 AddRequestHandler() 方法将这些方法与 MessageName 对应起来，添加到 m_EventHandlers 中。

需要注意的是，本文为了简要的说明这种处理方式，所以简化了 Test1 和 Test2 方法的实现。而在实际开发中，它们才是实现具体业务的重要方法。而利用这种方式，则解除了服务之间依赖的耦合度，我们随时可以为服务添加新的业务逻辑，也可以方便的增加服务。

通过这样的设计，Service 的 OnRequest()方法的最终实现如下所示：

```
public IMessage OnRequest(IMessage aMessage)
{
    string messageName = aMessage.GetMessageName();
    string messageId = aMessage.GetMessageID();
    IMessage message = m_Factory.CreateMessage();

    IMessageItemSequence replyMessage = m_Factory.CreateMessageItemSequence();
    RequestHandler handler = FindRequestHandler(messageName);
    handler(messageName, aMessage.GetMessageBody(), ref replyMessage);

    message.SetMessageName(messageName);
    message.SetMessageID(messageID);
    message.SetMessageBody(replyMessage);

    return message;
}
```

利用这种方式，我们可以非常方便的实现服务间通信，以及客户端与服务端间的通信。例如，我们分别在服务端定义 MyService(如前所示)和 TestService:

```
public class TestService : Service
{
    public TestService(string aServiceName)
        : base(aServiceName)
    { }

    protected override void AddRequestHandlers()
    {
        this.AddRequestHandler("Test1", new RequestHandler(Test1));
    }
}
```

```

private void Test1(string aMessageName, IMessageItemSequence aMessageBody, ref
IMessageItemSequence aReplyMessageBody)
{
    aReplyMessageBody = SendRequest("MyService", aMessageName, aMessageBody);
    aReplyMessageBody.SetValue("result2", "Success");
}
}

```

注意在 TestService 中的 Test1 方法，它并未直接处理消息 aMessageBody，而是通过调用 SendRequest()方法，将其传递到 MyService 中。

对于客户端而言，情况比较特殊。根据前面的分析，我们知道除了发送消息的操作是在客户端完成外，其他的具体执行都在服务端实现。所以诸如 MyService 和 TestService 等服务类，只需要部署在服务端即可。而客户端则只需要定义一个实现 Service 的空类即可：

```

public class MyServiceClient : Service
{
    public MyServiceClient(string aServiceName)
        : base(aServiceName)
    { }

    protected override void AddRequestHandlers()
    { }
}

```

MyServiceClient 类即为客户端定义的服务类，在 AddRequestHandlers()方法中并不需要实现任何代码。如果我们在 Service 抽象类中，将 AddRequestHandlers()方法定义为 virtual 而非 abstract 方法，则这段代码在客户端服务中也可以省去。另外，客户端服务类中的 aServiceName 可以任意赋值，它与服务端的服务名并无实际联系。至于客户端具体会调用哪个服务，则由 SendRequest()方法中的 aServiceName 决定：

```

IMessageFactory factory = new MessageFactory();
IMessageItemSequence body = factory.CreateMessageItemSequence();
//.....
MyServiceClient service = new MyServiceClient("Client");
IMessageItemSequence reply = service.SendRequest("TestService", "Test1", body);

```

对于 service.SendRequest()的执行而言，会先调用 TestService 的 Test1 方法；然后再通过该方法向 MyService 发送，最终调用 MyService 的 Test1 方法。

我们还需要另外定义一个类，负责添加服务，并初始化这些服务：

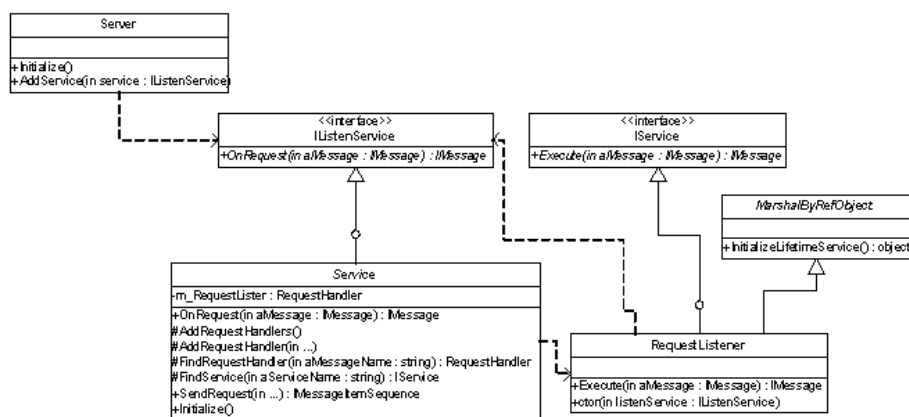
```
public class Server
{
    public Server()
    {
        m_Services = new ArrayList();
    }

    private ArrayList m_Services;
    public void AddService(ILogService service)
    {
        this.m_Services.Add(service);
    }

    public void Initialize()
    {
        IDictionary tcpProp = new Hashtable();
        tcpProp["name"] = "tcp9090";
        tcpProp["port"] = 9090;

        TcpChannel channel = new TcpChannel(tcpProp, new BinaryClientFormatterSinkProvider(),
new BinaryServerFormatterSinkProvider());
        ChannelServices.RegisterChannel(channel);
        foreach (Service service in m_Services)
        {
            service.Initialize();
        }
    }
}
```

同理，这里的 Channel，IP 和 Port 均应通过配置文件读取。最终的类图如下所示：



在服务端，可以调用 Server 类来初始化这些服务：

```
static void Main(string[] args)
```

```
{  
    MyService service = new MyService("MyService");  
    TestService service1 = new TestService("TestService");  
  
    Server server = new Server();  
    server.AddService(service);  
    server.AddService(service1);  
  
    server.Initialize();  
    Console.ReadLine();  
}
```

四、结论

利用这个基于消息与 .Net Remoting 技术的分布式架构，可以将企业的业务逻辑转换为对消息的定义和处理。要增加和修改业务，就体现在对消息的修改上。服务间的通信机制则完全交给整个架构来处理。如果我们将每一个委托所实现的业务（或者消息）理解为 **Contract**，则该结构已经具备了 **SOA** 的雏形。当然，该架构仅仅处理了消息的传递，而忽略了对底层事件的处理（类似于 **Corba** 的 **Event Service**），这个功能我想留待后面实现。

唯一遗憾的是，我缺乏验证这个架构稳定性和效率的环境。应该说，这个架构是我们在企业项目解决方案中的一个实践。但是解决方案则是利用了 **CORBA** 中间件，在 **Unix** 环境下实现并运行。本架构仅仅是借鉴了核心的实现思想和设计理念，从而完成的在 .Net 平台下的移植。由于 **Unix** 与 **Windows Server** 的区别，其实际的优势还有待验证。

1.2.11 .Net Remoting测试小技巧

其实也不能说是小技巧，只是大家可能会没有重视。这是需要先说明的，否则会被我的标题迷惑，因为我要描述的内容，真是寻常无比。

编写 **Remoting** 程序，通常分为三部分：远程对象、服务端程序、客户端程序。如果不考虑元数据的安全性，我们会把远程对象的 **dll** 生成相同的两份，分别放到服务端和客户端。**Remoting** 在客户端的调用是很简单的，但调试起来就没有那么容易。因为客户端和服务端分别属于不同的应用程序域，无法设置断点进行单步调试。如果了解 **NUnit**，大家会知道 **NUnit** 也是不支持分布式应用程序的调试的，至少是支持得不够好。

所以，在实际做项目的过程中，我更倾向于先调用本地的对象，等调试成功后，再打开 **Remoting** 服务，调用远程对象，验证是否正确。举例来说，我要提供访问数据库的远程对象。我会先让该对象在本地运行，并调用其方法。如果一切正常，说明数据库的配置和连接均是正确的。然后再将该调用替换为远程对象。如果程序出错，则可以肯定是 **Remoting** 提供的服务出错了，或者是远程对象未按照 **Remoting** 的规定，没有派生 **MarshalByRefObject**，

或者未提供序列化特性。

最初我使用了最愚蠢的办法，就是写两行调用，一个调用本地对象，一个调用远程对象。然后根据实际的情况，酌情考虑注释某一行代码。如此这般用了一段时间，终于觉得麻烦，迫使我改变方法了。其实很简单，就是为客户端程序的主类中，多写一个构造函数而已，呵呵：)

例如，远程对象是一个访问数据库的 Remoting 服务，派生 MarshalByRefObject 的主类名为 DBAccessService。那么我首先定义一个枚举，分别标明是属于本地调用还是远程调用：

```
public enum InvokeMode
{Local=0,Remoting}
```

对于客户端程序，如果主类为 DataBaseOperate，那么就需要增加一个构造函数和远程对象字段：

```
public DataBaseOperate(InvokeMode invokeMode)
{
    switch (invokeMode)
    {
        case InvokeMode.Local:
            serviceObj = new DBAccessService();
            break;
        case InvokeMode.Remoting:
            serviceObj =
                (DBAccessService)Activator.GetObject(typeof(DBAccessService),
                    "tcp://localhost:8080/DBAccessService");
            break;
    }
}

private DBAccessService serviceObj = null;
```

如此这般，在客户端调用对象时，就可根据设置构造函数中的参数枚举值，灵活地改变客户端调用对象的方式。

其实这种办法也可以用简单工厂模式来完成。不过这个简单工厂生产的产品和通常意义的工厂模式有点不一样哦，因为他们的产品其实是完全相同的。不同的仅仅是创建对象的方式而已。

```
public class SimpleFactory
{
    public static DBAccessService CreateInstance(InvokeMode invokeMode)
    {
```



```
switch (invokeMode)
{
    case InvokeMode.Local:
        return new DBAccessService();

        break;
    case InvokeMode.Remoting:
        return (DBAccessService)Activator.GetObject(typeof(DBAccessService),
"tcp://localhost:8080/DBAccessService");

        break;
}
}
```

然后再调用工厂的静态方法，来获得该对象即可。

两种方法都是一种思想：就是根据需要，选择不同的创建方式（其实前一种方法也可以看作是简单工厂模式的一种变种）。如果只有一个要创建的对象，选前者更为方便；如果需要创建多个对象，用工厂方法提供多个静态方法，应该要灵活一些。

通过上述方法，不仅便于调试，也便于以后代码的修改。如果取消使用 **Remoting**，只需改变参数枚举值即可，其他代码通通不用改变。这个技巧也算是设计模式的一种最简单应用吧。

1.2.12 .NET Remoting中的通道注册

今天我的同事使用 **Remoting** 注册一个新通道。奇怪的是，通道始终无法注册，总是报告异常“该通道已被占用”。我明白这个异常出现的原因，但不明白的是此时系统并未使用任何一个通道，为何会有这个异常呢？即使重新启动计算机也是如此，莫非有一个我们无法探测到的 **Remoting** 服务在顽强且隐匿的在吞噬着通道？无论是 **tcp** 通道和 **http** 通道均是如此，真是奇怪啊。

当然要解决这个问题是非常 **easy** 的，只需要在注册新通道前加上如下几行代码就 **OK** 了：

```
if (ChannelServices.RegisteredChannels.GetLength(0) > 0)
{
    foreach (IChannel channel in ChannelServices.RegisteredChannels)
    {
        ChannelServices.UnregisterChannel(channel);
    }
}
```

```
//再注册新通道;  
TcpChannel newChannel = new TcpChannel(8080);  
//
```

这也提示我们，在写 Remoting 的应用程序时，如果要注册新通道，似乎有检查已有注册通道的必要哦。

1.2.13 在Remoting客户端激活用替换类以分离接口与实现

在我的博客里，《Microsoft .Net Remoting[基础篇]》一文提到了用替代类来取代远程对象元数据的方法。有朋友对此有些疑惑不解，所以专门 Post 本贴以解惑。

我们在部署 Remoting 分布式应用程序时，通常要求接口与实现完全分离，这样可以保证元数据的安全。也就是说，在客户端部署程序集时，不应包括远程对象的实现。然而，如果我们采用的是客户端激活方式，这个目的就很难实现。为什么呢？我们首先来看看客户端激活方式与服务端激活方式在客户端的区别：

如果是服务端激活方式，方法如下：

```
RemoteObject obj = (RemoteObject)Activator.GetObject(typeof(RemoteObject),ObjectUri);
```

其中 RemoteObject 为远程对象，ObjectUri 是知名对象的 Uri。从其实现来看，它需要知道远程对象的类型，但并不需要实例化。如果我们令该远程对象实现某接口，那么在 GetObject 方法中，完全可以传递接口的类型，并返回接口类型对象。

再看客户端激活方式，方法有两种：

一是调用 Activator.CreateInstance()方法，很显然，这种方法要求的类型必须是类对象，因为它要创建具体实例。

第二种方法是采用 RemotingConfiguration.RegisterActivatedClientType()方法。虽然在该方法中，也只需要传递其类型即可。但这种方法要求在注册了远程对象后，必须实例化该远程对象：

```
RemoteObject obj = new RemoteObject;
```

显然要求的类型也必须是类对象，道理不言而喻。

那么，当我们采用客户端激活方式时，怎样让接口与实现分离呢？在拙文《Microsoft .Net Remoting[基础篇]》中，提到有两种方法。一是使用工厂模式，再以服务端 SingleTon 激活模式来模拟客户端激活方式；另一种就是使用替代类。

所谓替代类，就是仍然创建和远程对象类相似的类对象。所谓的类似，是指远程对象类的命名空间、公共方法、字段和属性均相同，但方法和属性的具体实现却采用空语句或无关实现的方式。我们来看看远程对象类和替代类的代码：

远程对象类：

```
public class RemotingObj : MarshalByRefObject
{
    public void Foo()
    {
        //在这里可以放上许多业务;
        Console.WriteLine("Just a test.");
    }
}
```

替代类：

```
public class RemotingObj : MarshalByRefObject
{
    public void Foo()
    {
        return;
    }
}
```

从代码中可以看到，远程对象类的方法 `Foo()` 中，有具体的业务实现，而在其对应的替代类 `Foo()` 方法中，却只有一条简单的 `Return` 语句。

在部署的时候，我们将实际的远程对象类放到服务端，而把替代类放到客户端，这样就可以保护远程对象类的元数据安全了。

为什么可以这样呢？我们要知道，在 `Remoting` 中采用客户端激活方式时，虽然在客户端创建了远程对象实例，实际上此时只是创建了一个代理类而已，它真正指向的还是在服务端创建的远程类对象。我们提供这样一个替代类，就是用一种“欺骗”的方式，瞒过客户端，使其可以顺利地实例化这个代理类。而当我们在客户端调用这个对象的方法：

`obj.Foo();`

此时它将通过代理类去读取远程服务端的真实地远程对象。也就是说，此时的 `Foo()` 方法并非替代类的 `return` 语句，而是真正远程对象类的业务实现。

这种方法其实就是一个 `Trick`，看似巧妙，不过在大型的应用时，其缺陷也是很明显的。由于我们要为每个远程对象都要实现该具体类，因此当远程对象众多时，带来的工作量也是可观的。这种方法也还算不上是真正的分布式应用。因此，大家如果要使用客户端激活方式，同时要求接口和实现分离，最好还是采用工厂模式，并以服务端 `SingleTon` 激活模式来模拟

客户端激活方式。

附示例源代码下载：<http://files.cnblogs.com/wayfarer/替代类.rar>

2.1、版权声明

文章出处：<http://www.cnblogs.com/lovecherry/archive/2005/05/24/161437.html>

文章作者：lovecherry

2.2、内容详情

2.2.1 一步一步学Remoting之一：从简单开始

一、Remoting 的优缺点？

优点：

- 1、能让我们进行分布式开发
- 2、Tcp 通道的 Remoting 速度非常快
- 3、虽然是远程的，但是非常接近于本地调用对象
- 4、可以做到保持对象的状态
- 5、没有应用程序限制，可以是控制台，winform，iis，windows 服务承载远程对象

缺点：

- 1、非标准的应用因此有平台限制
- 2、脱离 iis 的话需要有自己的安全机制

二、Remoting 和 Web 服务的区别？

ASP.NET Web 服务基础结构通过将 SOAP 消息映射到方法调用，为 Web 服务提供了简单的 API。通过提供一种非常简单的编程模型（基于将 SOAP 消息交换映射到方法调用），它实现了此机制。ASP.NET Web 服务的客户端不需要了解用于创建它们的平台、对象模型或编程语言。而服务也不需要了解向它们发送消息的客户端。唯一的要求是：双方都要认可正在创建和使用的 SOAP 消息的格式，该格式是由使用 WSDL 和 XML 架构 (XSD) 表示的 Web 服务合约定义来定义的。

.NET Remoting 为分布式对象提供了一个基础结构。它使用既灵活又可扩展的管线向远程进程提供 .NET 的完全对象语义。ASP.NET Web 服务基于消息传递提供非常简单的编程模型，而 .NET Remoting 提供较为复杂的功能，包括支持通过值或引用传递对象、回调，以及多对象激活和生命周期管理策略等。要使用 .NET Remoting，客户端需要了解所有这些详细信息，简而言之，需要使用 .NET 建立客户端。.NET Remoting 管线还支持 SOAP 消息，但必须注意这并没有改变其对客户端的要求。如果 Remoting 端点提供 .NET 专用的

对象语义，不管是否通过 SOAP，客户端必须理解它们。

三、最简单的 Remoting 的例子

1、远程对象：

建立类库项目：RemoteObject

```
using System;

namespace RemoteObject
{
    public class MyObject:MarshalByRefObject
    {
        public int Add(int a,int b)
        {
            return a+b;
        }
    }
}
```

2、服务端

建立控制台项目：RemoteServer

```
using System;
using System.Runtime.Remoting;

namespace RemoteServer
{
    class MyServer
    {
        [STAThread]
        static void Main(string[] args)
        {
            RemotingConfiguration.Configure("RemoteServer.exe.config");
            Console.ReadLine();
        }
    }
}
```

建立配置文件：app.config

```
<configuration>
  <system.runtime.remoting>
```

```

<application name="RemoteServer">
    <service>
        <wellknown type="RemoteObject.MyObject, RemoteObject"
            objectUri="RemoteObject.MyObject"
            mode="Singleton" />
    </service>
    <channels>
        <channel ref="tcp" port="9999"/>
    </channels>
</application>
</system.runtime.remoting>
</configuration>

```

3、客户端：

建立控制台项目：RemoteClient

```

using System;

namespace RemoteClient
{
    class MyClient
    {
        [STAThread]
        static void Main(string[] args)
        {
            RemoteObject.MyObject app =
            (RemoteObject.MyObject)Activator.GetObject(typeof(RemoteObject.MyObject), System.Configuration.
            n.ConfigurationSettings.AppSettings["ServiceURL"]);
            Console.WriteLine(app.Add(1, 2));
            Console.ReadLine();
        }
    }
}

```

建立配置文件：app.config

```

<configuration>
    <appSettings>
        <add key="ServiceURL" value="tcp://localhost:9999/RemoteObject.MyObject"/>
    </appSettings>
</configuration>

```

4、测试

在最后编译的时候会发现编译报错：

- 1、找不到 app.Add()
 - 2、找不到 RemoteObject

这是因为客户端 `RemoteClient` 没有添加 `RemoteObject` 的引用，编译器并不知道远程对象存在哪些成员所以报错，添加引用以后 `vs.net` 会在客户端也保存一个 `dll`，可能大家会问这样如果对远程对象的修改是不是会很麻烦？其实不麻烦，对项目编译一次 `vs.net` 会重新复制 `dll`。

然后直接运行客户端会出现“目标主机拒绝”的异常，也说明了通道没有打开运行服务端再运行客户端出现“找不到程序集 `RemoteObject`”！回头想想可以发现我们并在服务端对 `RemoteObject` 添加引用，编译的时候通过是因为这个时候并没有用到远程对象，大家可能不理解运行服务端的时候也通过？这是因为没有这个时候还没有激活远程对象。理所当然，对服务端要添加引用远程对象，毕竟我们的对象是要靠远程承载的。

现在再先后运行服务端程序和客户端程序，客户端程序显示 3，测试成功。

四、结束语

我们通过一个简单的例子实现了最简单的 `remoting`，对其实质没有做任何介绍，我想通过例子入门才是最简单的。

2.2.2 一步一步学Remoting之二：激活模式

远程对象的激活模式分**服务端激活**和**客户端激活**两种，（也就是对象分服务端激活对象或者说是知名对象和客户端激活对象两种）先看看 `msdn` 怎么描述服务端激活的：

服务器激活的对象是其生存期由服务器直接控制的对象。服务器应用程序域只有在客户端在对象上进行方法调用时才创建这些对象，而不会在客户端调用 `new` 或 `Activator.GetObject` 时创建这些对象；这节省了仅为创建实例而进行的一次网络往返过程。客户端请求服务器激活的类型实例时，只在客户端应用程序域中创建一个代理。然而，这也意味着当您使用默认实现时，只允许对服务器激活的类型使用默认构造函数。若要发布其实例将使用带参数的特定构造函数创建的类型，可以使用客户端激活或者动态地发布您的特定实例。

服务器激活的对象有两种激活模式（或 `WellKnownObjectMode` 值）：`Singleton` 和 `SingleCall`。

`Singleton` 类型任何时候都不会同时具有多个实例。如果存在实例，所有客户端请求都由该实例提供服务。如果不存在实例，服务器将创建一个实例，而所有后继的客户端请求都将由该实例来提供服务。由于 `Singleton` 类型具有关联的默认生存期，即使任何时候都不会

有一个以上的可用实例，客户端也不会总接收到对可远程处理的类的同一实例的引用。

SingleCall 远程服务器类型总是为每个客户端请求设置一个实例。下一个方法调用将改由其他实例进行服务。从设计角度看，**SingleCall** 类型提供的功能非常简单。这种机制不提供状态管理，如果您需要状态管理，这将是一个不利之处；如果您不需要，这种机制将非常理想。也许您只关心负载平衡和可伸缩性而不关心状态，那么在这种情况下，这种模式将是您理想的选择，因为对于每个请求都只有一个实例。如果愿意，开发人员可以向 **SingleCall** 对象提供自己的状态管理，但这种状态数据不会驻留在对象中，因为每次调用新的方法时都将实例化一个新的对象标识。

首先对于服务端激活的两种模式来做一个试验，我们把远程对象做如下的修改：

```
using System;

namespace RemoteObject
{
    public class MyObject:MarshalByRefObject
    {
        private int i=0;
        public int Add(int a,int b)
        {
            return a+b;
        }

        public int Count()
        {
            return ++i;
        }
    }
}
```

对客户端做以下修改：

```
RemoteObject.MyObject app =
(RemoteObject.MyObject)Activator.GetObject(typeof(RemoteObject.MyObject),System.Configur
ation.ConfigurationSettings.AppSettings["ServiceURL"]);
    Console.WriteLine(app.Count());
    Console.ReadLine();
```

第一次打开客户端的时候显示 1，第二次打开的时候显示 2，类推……由此验证了 **Singleton** 类型任何时候都不会同时具有多个实例。如果存在实例，所有客户端请求都由该实例提供服务。如果不存在实例，服务器将创建一个实例，而所有后继的客户端请求都将由该实例来提供服务。

把服务器端的 config 修改一下：

```
<wellknown type="RemoteObject.MyObject, RemoteObject" objectUri="RemoteObject.MyObject"
           mode="SingleCall" />
```

（这里注意大小写，大写的 C）

再重新运行服务端和客户端，打开多个客户端发现始终显示 1。由此验证了 SingleCall 类型对于每个客户端请求都会重新创建实例。下一个方法调用将由另一个服务器实例提供服务。

下面再说一下客户端的激活模式，msdn 中这么写：

客户端激活的对象是其生存期由调用应用程序域控制的对象，正如对象对于客户端是本地对象时对象的生存期由调用应用程序域控制一样。对于客户端激活，当客户端试图创建服务器对象的实例时发生一个到服务器的往返过程，而客户端代理是使用对象引用 (ObjRef) 创建的，该对象引用是从在服务器上创建远程对象返回时获取的。每当客户端创建客户端激活的类型的实例时，该实例都将只服务于该特定客户端中的特定引用，直到其租约到期并回收其内存为止。如果调用应用程序域创建两个远程类型的新实例，每个客户端引用都将只调用从其中返回引用的服务器应用程序域中的特定实例。

理解一下，可以归纳出

1、客户端激活的时间是在客户端请求的时候，而服务端激活远程对象的时间是在调用对象方法的时候

远程对象修改如下：

```
using System;

namespace RemoteObject
{
    public class MyObject:MarshalByRefObject
    {
        private int i=0;

        public MyObject()
        {
            Console.WriteLine("激活");
        }

        public int Add(int a, int b)
        {
            return a+b;
        }
    }
}
```

```

        public int Count()
        {
            return ++i;
        }
    }
}

```

服务端配置文件：

```

<configuration>
  <system.runtime.remoting>
    <application name="RemoteServer">
      <service>
        <activated type="RemoteObject.MyObject, RemoteObject"/>
      </service>
      <channels>
        <channel ref="tcp" port="9999"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

客户端程序：

```

using System;

namespace RemoteClient
{
    class MyClient
    {
        [STAThread]
        static void Main(string[] args)
        {
            //RemoteObject.MyObject app =
            (RemoteObject.MyObject)Activator.GetObject(typeof(RemoteObject.MyObject), System.Configuration.
            n.ConfigurationSettings.AppSettings["ServiceURL"]);
            RemoteObject.MyObject
            app=(RemoteObject.MyObject)Activator.CreateInstance(typeof(RemoteObject.MyObject), null, new
            object[] {new
            System.Runtime.Remoting.Activation.UrlAttribute(System.Configuration.ConfigurationSettings.A
            ppSettings["ServiceURL"])});
            //Console.WriteLine(app.Count());
        }
    }
}

```

```

        Console.ReadLine();
    }
}

```

客户端配置文件：

```

<configuration>
  <appSettings>
    <add key="ServiceURL" value="tcp://localhost:9999/RemoteServer"/>
  </appSettings>
</configuration>

```

（这里的 uri 按照服务端配置文件中 application 元素定义的 RemoteServer 来写）

运行程序可以看到，在客户端启动的时候服务端就输出了“激活”，我们再转回知名模式进行测试发现只有运行了方法才会在服务端输出“激活”。

2、客户端激活可以调用自定义的构造方法，而不像服务端激活只能使用默认的构造方法

把客户端代码修改如下：

```

RemoteObject.MyObject app=
(RemoteObject.MyObject)Activator.CreateInstance(typeof(RemoteObject.MyObject),
new object[]{10},
new object[]{new System.Runtime.Remoting.Activation.UrlAttribute(
    System.Configuration.ConfigurationSettings.AppSettings["ServiceURL"])});
    Console.WriteLine(app.Count());

```

这里看到我们在 CreateInstance 方法的第二个参数中提供了 10 作为构造方法的参数。在服务端激活模式我们不能这么做。

远程对象构造方法修改如下：

```

public MyObject(int k)
{
    this.i=k;
    Console.WriteLine("激活");
}

```

毫无疑问，我们运行客户端发现输出的是 11 而不是 1 了。

3、通过上面的例子，我们运行多个客户端发现出现的永远是 11，因此，客户端激活模式一旦获得客户端的请求，将为每一个客户端都建立一个实例引用。

总结：

1、Remoting 支持两种远程对象：知名的和客户激活的。知名的远程对象使用了 uri 作为标识，客户程序使用这个 uri 来访问那些远程对象，也正式为什么称作知名的原因。对知名的对象来说 2 种使用模式：SingleCall 和 Singleton，对于前者每次调用都会新建对象，因此对象是无状态的。对于后者，对象只被创建一次，所有客户共享对象状态，因此对象是有状态的。另外一种客户端激活对象使用类的类型来激活，uri 再后台被动态创建，并且返回给客户程序。客户激活对象是有状态的。

2、对于 Singleton 对象来说需要考虑伸缩性，Singleton 对象不能在多个服务器上被部署，如果要跨服务器就不能使用 Singleton 了。

备注：个人习惯原因，在我的例子中服务端的配置都是用 config 文件的，客户端的配置都是基本用程序方式的

使用配置文件的优点：无需重新编译就可以配置通道和远程对象，编写的代码量比较少
使用程序定制的优点：可以获得运行期间的信息，对程序调试有利。

2.2.3 一步一步学Remoting之三：复杂对象

这里说的复杂对象是比较复杂的类的实例，比如说我们在应用中经常使用的 DataSet，我们自己的类等，通常我们会给远程的对象传递一些自己的类，或者要求对象返回处理的结果，这个时候通常也就是需要远程对象有状态，上次我们说了几种激活模式提到说只有客户端激活和 Singleton 是有状态的，而客户端激活和 Singleton 区别在于 Singleton 是共享对象的。因此我们可以选择符合自己条件的激活方式：

	状态	拥有各自实例
Singleton	有	无
SingleCall	无	有
客户端激活	有	有

在这里，我们先演示自定义类的传入传出：

先说一个概念：**MBV** 就是按值编码，对象存储在数据流中，用于在网络另外一端创建对象副本。**MBR** 就是按引用编组，在客户机上创建代理，远程对象创建 ObjRef 实例，实例被串行化传递。

我们先来修改一下远程对象：

```
using System;
```

```
namespace RemoteObject
{
    public class MyObject:MarshalByRefObject
    {
        private MBV _mbv;
        private MBR _mbr;
        public int Add(int a,int b)
        {
            return a+b;
        }

        public MBV GetMBV()
        {
            return new MBV(100);
        }

        public MBR GetMBR()
        {
            return new MBR(200);
        }

        public void SetMBV(MBV mbv)
        {
            this._mbv=mbv;
        }

        public int UseMBV()
        {
            return this._mbv.Data;
        }

        public void SetMBR(MBR mbr)
        {
            this._mbr=mbr;
        }

        public int UseMBR()
        {
            return this._mbr.Data;
        }
    }

    [Serializable]
    public class MBV
```

```
{  
    private int _data;  
    public MBV(int data)  
    {  
        this._data=data;  
    }  
    public int Data  
    {  
        get  
        {  
            return this._data;  
        }  
        set  
        {  
            this._data=value;  
        }  
    }  
}  
  
public class MBR:MarshalByRefObject  
{  
    private int _data;  
    public MBR(int data)  
    {  
        this._data=data;  
    }  
    public int Data  
    {  
        get  
        {  
            return this._data;  
        }  
        set  
        {  
            this._data=value;  
        }  
    }  
}  
}
```

Get 方法用来从服务器返回对象，Set 方法用于传递对象到服务器，Use 方法用来测试远程对象的状态是否得到了保存。

我们先来测试一下客户端激活模式：（服务器端的设置就不说了）

```

RemoteObject.MyObject app=
(RemoteObject.MyObject)Activator.CreateInstance(typeof(RemoteObject.MyObject), null,
new object[] {
new System.Runtime.Remoting.Activation.UrlAttribute(
System.Configuration.ConfigurationSettings.AppSettings["ServiceURL"]));
    RemoteObject.MBV mbv=app.GetMBV();
    Console.WriteLine(mbv.Data);
    RemoteObject.MBR mbr=app.GetMBR();
    Console.WriteLine(mbr.Data);
    mbv=new RemoteObject.MBV(100);
    app.SetMBV(mbv);
    Console.WriteLine(app.UseMBV());
    //mbr=new RemoteObject.MBR(200);
    //app.SetMBR(mbr);
    //Console.WriteLine(app.UseMBR());
    Console.ReadLine();

```

依次显示：100，200，100

前面 2 个 100，200 说明我们得到了服务器端返回的对象（分别是 MBV 和 MBR 方式的），后面一个 100 说明我们客户端建立了一个 MBV 的对象传递给了服务器，因为客户端激活模式是有状态的所以我们能使用这个对象从而输出 100，最后我们注释了几行，当打开注释运行后出现异常“由于安全限制，无法访问类型 System.Runtime.Remoting.ObjRef。”这个在【通道】一节中会讲到原因。

好了，我们再来测试一下 Singleton（别忘记修改客户端配置文件中的 URI 哦）

```

RemoteObject.MyObject app =
(RemoteObject.MyObject)Activator.GetObject(typeof(RemoteObject.MyObject),
System.Configuration.ConfigurationSettings.AppSettings["ServiceURL"]);
//RemoteObject.MyObject
app=(RemoteObject.MyObject)Activator.CreateInstance(typeof(RemoteObject.MyObject), null, new
object[] {new
System.Runtime.Remoting.Activation.UrlAttribute(System.Configuration.ConfigurationSettings.A
ppSettings["ServiceURL"]));

```

后面的语句省略，运行后同样出现 100，200，100——Singleton 也是能保存状态的。SingleCall 呢？修改一下服务端的 Config 再来一次，在“Console.WriteLine(app.UseMBV());”出现了“未将对象引用设置到对象的实例。”因为服务端没有能够保存远程对象的状态，当然出错。

再看一下 .net 内置的一些复杂对象，比如 DataSet，可能传入传出 DataSet 和 DataTable 在应用中比较普遍，一些不可序列化的类我们不能直接传递，比如 DataRow 等，要传递的时候可以考虑放入 DataTable 容器中。

远程对象修改如下：

```
using System;
using System.Data;

namespace RemoteObject
{
    public class MyObject:MarshalByRefObject
    {
        public DataSet Method(DataSet ds)
        {
            DataTable dt=ds.Tables[0];
            foreach(DataRow dr in dt.Rows)
            {
                dr["test"]=dr["test"]+"_ok";
            }
            return ds;
        }
    }
}
```

客户端修改如下：

```
RemoteObject.MyObject app =
(RemoteObject.MyObject)Activator.GetObject(typeof(RemoteObject.MyObject), System.Configuration.
n.ConfigurationSettings.AppSettings["ServiceURL"]);
DataSet ds=new DataSet();
DataTable dt=new DataTable();
dt.Columns.Add(new DataColumn("test", typeof(System.String)));
DataRow dr=dt.NewRow();
dr["test"]="data";
dt.Rows.Add(dr);
ds.Tables.Add(dt);
ds=app.Method(ds);
Console.WriteLine(ds.Tables[0].Rows[0]["test"].ToString());
Console.ReadLine();
```

运行后发现输出 data_ok 了。在这里不管用哪种模式来激活都会得到 data_ok，因为我们并没有要求远程对象来保存状态。

总结：

所有必须跨越应用程序域的本地对象都必须按数值来传递，并且应该用 [serializable]

自定义属性作标记，否则它们必须实现 `ISerializable` 接口。对象作为参数传递时，框架将该对象序列化并传输到目标应用程序域，对象将在该目标应用程序域中被重新构造。无法序列化的本地对象将不能传递到其他应用程序域中，因而也不能远程处理。通过从 `MarshalByRefObject` 导出对象，可以使任一对象变为远程对象。当某个客户端激活一个远程对象时，它将接收到该远程对象的代理。对该代理的所有操作都被适当地重新定向，使远程处理基础结构能够正确截取和转发调用。尽管这种重新定向对性能有一些影响，但 JIT 编译器和执行引擎 (EE) 已经优化，可以在代理和远程对象驻留在同一个应用程序域中时，防止不必要的性能损失。如果代理和远程对象不在同一个应用程序域中，则堆栈中的所有方法调用参数会被转换为消息并被传输到远程应用程序域，这些消息将在该远程应用程序域中被转换为原来的堆栈帧，同时该方法调用也会被调用。从方法调用中返回结果时也使用同一过程。

2.2.4 一步一步学Remoting之四：承载方式（1）

在实际的应用中我们通常只会选择用 windows 服务和 iis 来承载远程对象。选择 windows 服务的原因是能自启动服务，服务器重启后不需要再去考虑启动 service。选择 iis 的理由是我们能使用集成验证等一些 iis 的特性。

在 msdn 中可以找到相关文章：

<http://www.microsoft.com/china/msdn/library/architecture/architecture/architecturetopic/BuildSuccApp/BSAAsecmodsecmod29.msp>

<http://msdn.microsoft.com/library/chs/default.asp?url=/library/CHS/cpguide/html/cpconRemotingExampleHostingInIIS.asp>

可能大家会觉得这个过程将是一个复杂的过程，其实不然，下面说一下实现方法，步骤非常少。

先来建立远程对象

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace RemoteObject
{
    public class MyObject:MarshalByRefObject
    {
        public DataSet GetData()
        {
            SqlConnection conn=new
SqlConnection(System.Configuration.ConfigurationSettings.AppSettings["strconn"]);
            SqlDataAdapter da=new SqlDataAdapter("select * from UBI_ProvinceMaster", conn);
```

```

        DataSet ds=new DataSet();
        da.Fill(ds);
        return ds;
    }
}

```

客户端仍然是一个控制台来进行测试：

```

RemoteObject.MyObject app =
(RemoteObject.MyObject)Activator.GetObject(typeof(RemoteObject.MyObject),System.Configuration.
n.ConfigurationSettings.AppSettings["ServiceURL"]);
DataTable dt=app.GetData().Tables[0];
foreach(DataRow dr in dt.Rows)
{
    Console.WriteLine(dr["iPrMId"]+" "+dr["vPrMName"]);
}
Console.ReadLine();

```

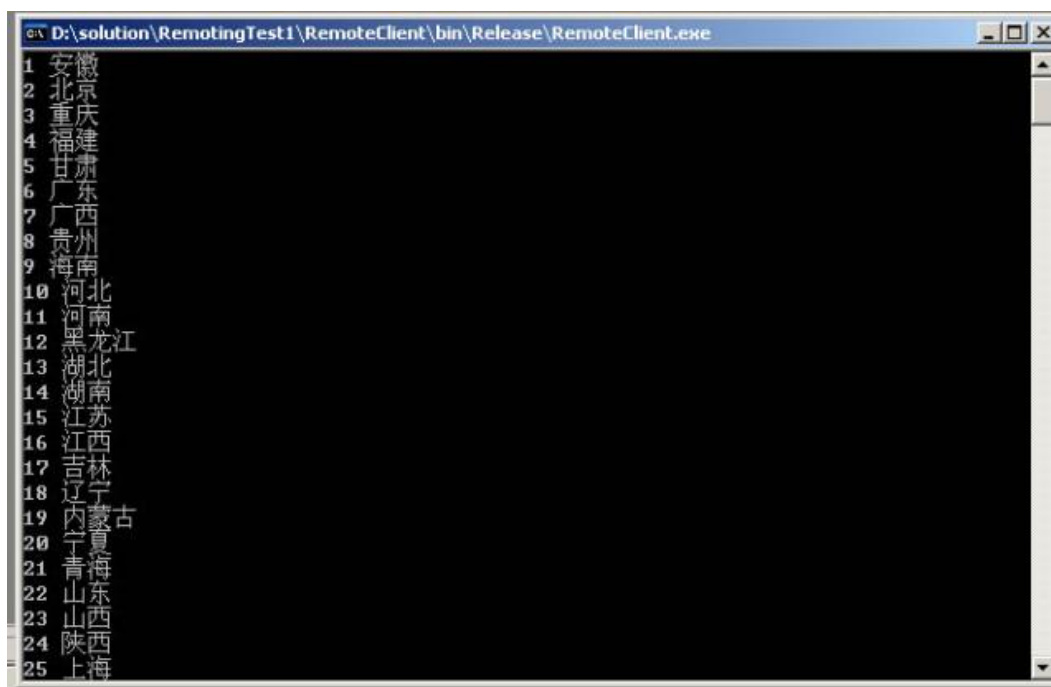
服务端配置文件：

```

<configuration>
  <appSettings>
    <add key="strconn" value="server=(local);uid=sa;pwd=;database=UBISOFT" />
  </appSettings>
  <system.runtime.remoting>
    <application name="RemoteServer">
      <service>
        <wellknown type="RemoteObject.MyObject, RemoteObject"
objectUri="RemoteObject.MyObject"
mode="SingleCall" />
      </service>
      <channels>
        <channel ref="tcp" port="9999"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

运行程序，我们得到的是一个省市的列表：



一、windows 服务承载

用 vs.net 制作一个 windows 服务的过程基本不超过 10 个步骤，所以我们不需要害怕。

- 1、建立一个新的 windows 服务项目 RemoteServer1
- 2、打开 Service1 代码视图，找到 OnStart 部分，加入代码

```
System.Runtime.Remoting.RemotingConfiguration.Configure(AppDomain.CurrentDomain.BaseDirectory + "RemoteServer1.exe.config");
```

(不要遗漏 AppDomain.CurrentDomain.BaseDirectory +)

config 和控制台方式的 config 是一样的，我们让这个 windows 服务做的仅仅是从 config 文件读出配置信息进行配置通道。别忘记添加配置文件。

- 3、切换到设计视图，右键—添加安装程序
 - 4、切换到新生成的 ProjectInstaller.cs 设计视图，找到 serviceProcessInstaller1 对 Account 属性设置为 LocalSystem，对 serviceInstaller1 的 ServiceName 属性设置为 RemoteServer1（服务的名字），StartType 属性设置为 Automatic（系统启动的时候自动启动服务）
 - 5、别忘记对添加 RemoteObject 的引用
 - 6、建立一个新的安装项目 RemoteServerSetup（我们为刚才那个服务建立一个安装项目）
 - 7、右键—添加—项目输出—主输出—选择 RemoteService1—确定
 - 8、右键—视图—自定义操作—自定义操作上右键—添加自定义操作—打开应用程序文件夹—选择刚才那个主输出—确定
 - 9、重新生成这个安装项目—右键—安装
 - 10、在服务管理器中（我的电脑—右键—管理—服务和应用程序—服务）找到 RemoteServer1 服务，启动服务
- 现在就可以打开客户端测试了！

一些 FAQ:

1、启动服务的时候系统说了类似“服务什么都没有做，服务已经被停止”表示什么？
表示 windows 服务出错了，一般是服务的程序有问题，检查服务做了什么？在我们这个程序中仅仅添加了一行代码，一般不会出现这个错误。

2、运行客户端出现“服务器无响应”？

先检查 windows 服务配置文件是不是正确设置了激活方式和激活对象，客户端服务端端口号是否统一？

3、运行客户端出现“无法找到程序集”？

检查 windows 服务配置文件是否正确配置了激活对象的类型和 uri？服务是否添加了远程对象引用？

4、远程对象类中有用到 `System.Configuration.ConfigurationSettings.AppSettings["strconn"]`，但是远程对象并没有配置文件，它从哪里读取这个 config 的？

因为远程对象不是独立存在的，它被 windows 服务承载的，因此它从 windows 服务的配置文件中读取一些配置信息，远程对象本身不需要配置文件。

5、安装的时候是不是要卸载服务？

不需要，安装程序会 停止服务端—》卸载服务—》安装服务

6、在正式使用的时候怎么部署我们的系统？

如果客户端是程序仅仅只要把安装项目下面 3 个文件传到服务器进行安装，配置好 config 文件（比如连接字符串），开启服务即可。如果客户端是网站，同样把服务在服务器安装，配置好 config 文件（比如连接字符串），开启服务，最后把网站传到 web 服务器（可能和 service 不是同一个服务器）。

7、部署的时候需要传远程对象 dll 吗？

不需要，可以看到安装项目中已经自动存在了这个 dll。

8、这样的系统有什么特点？

一个 web 服务器，多个 service 服务器，多个 sqlservice 服务器，web 服务器负担比较小，所有的逻辑代码都分布到不同的 service 服务器上面。

最后说一个测试的 tip:

如果我们远程调用对象进行测试程序非常麻烦，我们需要这么做

修改了远程对象—》重新编译安装程序—》在自己机器重新安装服务—》启动服务—》查看结果

其实可以这么做：

1、修改远程对象中的连接数据库字符串，由于不是远程对象了，我们必须从本地读取连接字符串，比如上列我们直接修改为：

```
SqlConnection conn=new SqlConnection("server=(local);uid=sa;pwd=;database=UBISOFT");
```

2、修改客户端代码，直接实例化远程对象

```
//RemoteObject.MyObject app =
(RemoteObject.MyObject)Activator.GetObject(typeof(RemoteObject.MyObject),System.Configuration.
n.ConfigurationSettings.AppSettings["ServiceURL"]);
```

```
RemoteObject.MyObject app = new RemoteObject.MyObject();
```

等到正式部署的时候我们还原数据库连接字符串从 config 文件中读取,还原远程对象从远程读取即可。

如果对 windows 服务还不是很清楚, 请看以下文章:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vbwlkwalkthroughcreatingwindowsserviceapplication.asp>

<http://www.cnblogs.com/lovecherry/archive/2005/03/25/125527.html>

2.2.5 一步一步学Remoting之四：承载方式（2）

这里来说一下 iis 承载方式, 顺便简单说一下 remoting 的通道和【复杂对象】中的遗留问题。

首先明确一点: iis 来承载的话只能是 http 通道方式的。

我们来建立一个 web 项目, 比如叫 remoting, 删除项目中的所有 webform, 把远程对象 dll—RemoteObject.dll 复制到项目的 dll 文件夹下面, 然后打开 web.config 进行服务端设置:

```
<configuration>
  <appSettings>
    <add key="strconn" value="server=(local);uid=sa;pwd=;database=UBISOFT" />
  </appSettings>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown type="RemoteObject.MyObject, RemoteObject"
          objectUri="MyObject.soap"
          mode="SingleCall" />
      </service>
      <channels>
        <channel ref="http" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

来分析一下这个 config:

- 1、可能大家还不是很理解 type 属性，其实 type 属性分两部分<命名空间.类名>,<程序集>
- 2、objectUri 是用来表示对象的 uri 的，到时候我们用这个 uri 来连接到服务端
- 3、我们需要为 uri 指定 soap（soap 格式化）或者 rem（二进制格式化）后缀

要进行测试其实很简单，我们在浏览器输入：

`http://localhost/remoting/MyObject.soap?wsdl`

进行测试，如果发生问题基本就是配置文件的问题或者对象 dll 没有正确复制到 dll 目录

接下来修改一下客户端的配置文件就可以了，主要是修改地址。

```
<configuration>
  <appSettings>
    <add key="ServiceURL" value="http://localhost/remoting/MyObject.soap"/>
  </appSettings>
</configuration>
```

iis 承载方式默认是 80 端口，我们不需要在端口上做任何设置。还需要注意到的是 iis 方式，我们使用这样的格式作为地址：

`http://ip 地址/虚拟目录/远程对象.soap`

运行了客户端以后如果我们的数据量比较大的话，就算是本机我们也能感受到延迟，比 tcp 方式延迟厉害很多很多，其实 http 方式的 remoting 效率比 webservice 还要差，具体选择 http 方式的 remoting 还是 webservice 还是要看我们是不是对对象的状态有需求。

iis 的部署也是自动启动服务的，还有一个优点就是可以结合 iis 的 windows 身份认证，这个参照一些 iis 的配置文章，这里就不说了。

下面还是要来看一下两种【通道】：

默认情况下，HTTP 通道使用 SOAP 格式化程序，因此，如果客户端需要通过 Internet 访问对象，则可以使用 HTTP 通道。由于这种方法使用 HTTP，所以允许客户端通过防火墙远程访问 .NET 对象。将这些对象集成在 IIS 中，即可将其配置为 Web 服务对象。随后，客户端就可以读取这些对象的 WSDL 文件，以便使用 SOAP 与 Remoting 对象通信。

默认情况下，TCP 通道使用二进制格式化程序。此格式化程序以二进制格式进行数据的序列化，并使用原始套接字在网络中传送数据。如果对象部署在受防火墙保护的封闭环境中，则此方法是理想的选择。该方法使用套接字在对象之间传递二进制数据，因此性能更好。由于它使用 TCP 通道来提供对象，因此具有在封闭环境中开销较小的优点。由于防火墙和配置问题，此方法不能在 Internet 上使用。

因此我们也需要更根据自己的需求来选择通道！看看 remoting 有这么多可以选择的方式：选择激活模式，选择通道，选择承载方式，如此多的选择给了我们灵活的同时也增加了理解 remoting 的难度。

msdn相关章节：<http://msdn.microsoft.com/library/CHS/cpguide/html/cpconChannels.asp>

最后说一下前面的遗留问题，为什么会发生这个安全异常？

<http://www.cnblogs.com/lovecherry/archive/2005/05/20/159335.html>

msdn 说：

依赖于运行时类型验证的远程处理系统必须反序列化一个远程流，然后才能开始使用它，未经授权的客户端可能会试图利用反序列化这一时机。为了免受这种攻击，.NET 远程处理提供了两个自动反序列化级别：Low 和 Full。Low（默认值）防止反序列化攻击的方式是，在反序列化时，只处理与最基本的远程处理功能关联的类型，如自动反序列化远程处理基础结构类型、有限的系统实现类型集和基本的自定义类型集。**Full 反序列化级别支持远程处理在所有情况下支持的所有自动反序列化类型。**

我们首先来修改服务端的配置文件：

```
<configuration>
  <system.runtime.remoting>
    <application name="RemoteServer">
      <service>
        <wellknown type="RemoteObject.MyObject, RemoteObject"
          objectUri="RemoteObject.MyObject"
          mode="Singleton" />
      </service>
      <channels>
        <channel ref="tcp" port="9999"/>
        <serverProviders>
          <provider ref="wsdl" />
          <formatter ref="soap" typeFilterLevel="Full" />
          <formatter ref="binary" typeFilterLevel="Full" />
        </serverProviders>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

当然也可以用程序进行设置：

```
using System;
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
```



```
using System.Runtime.Serialization.Formatters;
```

```
RemotingConfiguration.RegisterWellKnownServiceType(typeof(RemoteObject.MyObject),  
"RemoteObject.MyObject", WellKnownObjectMode.Singleton);  
BinaryServerFormatterSinkProvider serverProvider = new BinaryServerFormatterSinkProvider();  
BinaryClientFormatterSinkProvider clientProvider = new BinaryClientFormatterSinkProvider();  
serverProvider.TypeFilterLevel = TypeFilterLevel.Full;  
IDictionary props = new Hashtable();  
props["port"] = 9999;  
TcpChannel channel = new TcpChannel(props, clientProvider, serverProvider);  
ChannelServices.RegisterChannel(channel);  
Console.ReadLine();
```

客户端还要用程序进行调整：

若要使用配置文件设置反序列化级别，必须显式指定 `<formatter>` 元素的 `typeFilterLevel` 属性。虽然这通常是在服务器端指定的，**但您还必须为注册来侦听回调的客户端上的任何信道指定这一属性**，以控制其反序列化级别

在程序前面加上和服务端基本相同的代码：

```
BinaryServerFormatterSinkProvider serverProvider = new BinaryServerFormatterSinkProvider();  
BinaryClientFormatterSinkProvider clientProvider = new BinaryClientFormatterSinkProvider();  
serverProvider.TypeFilterLevel = TypeFilterLevel.Full;  
IDictionary props = new Hashtable();  
props["port"] = 0;  
TcpChannel channel = new TcpChannel(props, clientProvider, serverProvider);  
ChannelServices.RegisterChannel(channel);
```

这样就可以了，注意：如果在同一个机器上面测试端口号应设为不同于服务器端设置的端口号，推荐设置为 0（远程处理系统自动选择可用端口）

.NET Remoting 自身不提供安全模型。然而，通过将远程对象驻留在 ASP.NET 中并使用 HTTP 通道进行通信，远程对象可以使用 IIS 和 ASP.NET 提供的基本安全服务。比较而言，TCP 通道和自定义的主机可执行文件能够提供更高的性能，但这种组合不提供内置的安全功能。

- 若要对客户端进行身份验证，请使用 HTTP 通道，在 ASP.NET 中驻留对象，以及在 IIS 中禁用匿名访问。
- 如果您不担心客户端身份验证问题，请使用 TCP 通道，它可以提供更高的性能。
- 如果您使用 TCP 通道，请使用 IPSec 保护客户端和服务端之间的通信通道。使用 SSL 来保护 HTTP 通道。

- 如果您需要对远程资源进行受信任的调用，请将组件驻留在 Windows 服务中，而不是驻留在控制台应用程序中。
- 始终不要向 Internet 公开远程对象。在这种情况下，请使用 Web 服务。应该仅在 Intranet 中使用 .NET Remoting。应该使用内部方式从 Web 应用程序访问对象。即使对象驻留在 ASP.NET 中，也不要向 Internet 客户端公开它们，因为客户端必须是 .NET 客户端。

最后，让我们来看一篇 msdn 有关 remoting 安全的文章：

<http://www.microsoft.com/china/msdn/library/architecture/architecture/architecturetopic/BUILDSucApp/BSAAsecmod11.aspx>

说到这里大家可能对 remoting 的一些基本知识稍微有点概念了，后续文章会继续不断强化这些概念！

2.2.6 一步一步学Remoting之五：异步操作

如果你还不知道什么是异步也不要紧，我们还是来看实例，通过实例来理解才是最深刻的。

在 Remoting 中，我们可以使用以下几种异步的方式：

- 1、普通异步
- 2、回调异步
- 3、单向异步

一个一个来说，首先我们这么修改我们的远程对象：

```
public int ALongTimeMethod(int a, int b, int time)
{
    Console.WriteLine("异步方法开始");
    System.Threading.Thread.Sleep(time);
    Console.WriteLine("异步方法结束");
    return a + b;
}
```

这个方法传入 2 个参数，返回 2 个参数和表示方法执行成功，方法需要 time 毫秒的执行时间，这是一个长时间的方法。

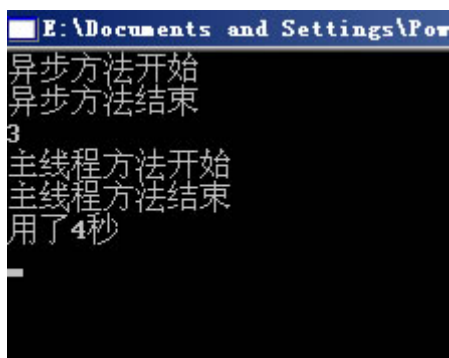
如果方法我们通过异步远程调用，这里需要注意到这个方法输出的行是在服务器端输出的而不是客户端。因此，为了测试简单，我们还是在采用本地对象，在实现异步前我们先来看看同步的调用方法，为什么说这是一种阻塞？因为我们调用了方法主线程就在等待了，看看测试：

```
DateTime dt=DateTime.Now;
RemoteObject.MyObject app=new RemoteObject.MyObject();
Console.WriteLine(app.ALongTimeMethod(1,2,1000));
Method();
Console.WriteLine("用了"+((TimeSpan)(DateTime.Now-dt)).TotalSeconds+"秒");
Console.ReadLine();
```

假设 method 方法是主线程的方法，需要 3 秒的时间：

```
private static void Method()
{
    Console.WriteLine("主线程方法开始");
    System.Threading.Thread.Sleep(3000);
    Console.WriteLine("主线程方法结束");
}
```

好了，现在开始运行程序：



用了 4 秒，说明在我们的方法开始以后本地就一直在等待了，总共用去的时间=本地方法+远程方法，对于长时间方法调用这显然不科学！我们需要改进：

1、普通异步：

首先在 main 方法前面加上委托，签名和返回类型和异步方法一致。

```
private delegate int MyDelegate(int a, int b, int time);
```

main 方法里面这么写：

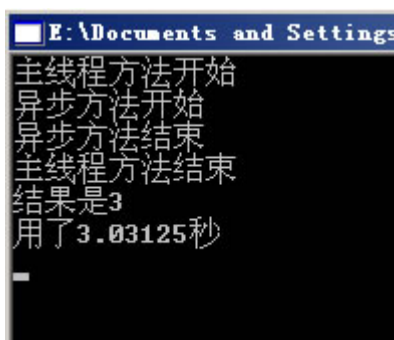
```
DateTime dt=DateTime.Now;
RemoteObject.MyObject app=new RemoteObject.MyObject();
MyDelegate md=new MyDelegate(app.ALongTimeMethod);
IAsyncResult Iar=md.BeginInvoke(1,2,1000,null,null);
Method();
if(!Iar.IsCompleted)
```

```

{
    Iar.AsyncWaitHandle.WaitOne();
}
else
{
    Console.WriteLine("结果是"+md.EndInvoke(Iar));
}
Console.WriteLine("用了"+((TimeSpan)(DateTime.Now-dt)).TotalSeconds+"秒");
Console.ReadLine();

```

来看一下执行结果：



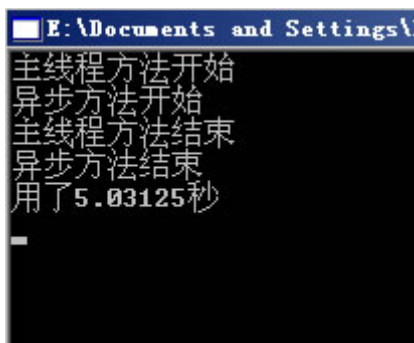
```

E:\Documents and Settings\
主线程方法开始
异步方法开始
异步方法结束
主线程方法结束
结果是3
用了3.03125秒

```

现在总共执行时间接近于主线程的执行时间了，等于是调用方法基本不占用时间。

分析一下代码：Iar.AsyncWaitHandle.WaitOne(); 是阻塞等待异步方法完成，在这里这段代码是不会被执行的，因为主方法完成的时候，异步方法早就 IsCompleted 了，如果我们修改一下代码：IAsyncResult Iar=md.BeginInvoke(1,2,5000,null,null);



```

E:\Documents and Settings\
主线程方法开始
异步方法开始
主线程方法结束
异步方法结束
用了5.03125秒

```

可以看到，主线程方法结束后就在等待异步方法完成了，总共用去了接近于异步方法的时间：5 秒。

在实际的运用中，主线程往往需要得到异步方法的结果，也就是接近于上述的情况，我们在主线程做了少量时间的工作以后最终要是需要 WaitOne 去等待异步操作返回的结果，才能继续主线程操作。看第二个图可以发现，异步操作仅仅用了 1 秒，但是要等待 3 秒的主线程方法完成后再返回结果，这还是不科学啊。因此，我们要使用委托回调的异步技术。

2、回调异步：

```
class MyClient
```

```

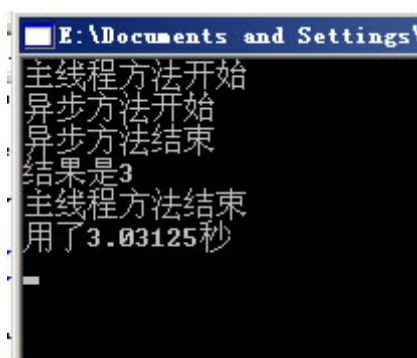
{
    private delegate int MyDelegate(int a, int b, int time);
    private static MyDelegate md;

    [STAThread]
    static void Main(string[] args)
    {
        DateTime dt = DateTime.Now;

        RemoteObject.MyObject app = new RemoteObject.MyObject();
        md = new MyDelegate(app.ALongTimeMethod);
        AsyncCallback ac = new AsyncCallback(MyClient.CallBack);
        IAsyncResult Iar = md.BeginInvoke(1, 2, 1000, ac, null);
        Method();
        Console.WriteLine("用了" + ((TimeSpan)(DateTime.Now - dt)).TotalSeconds + "秒");
        Console.ReadLine();
    }
    public static void CallBack(IAsyncResult Iar)
    {
        if (Iar.IsCompleted)
        {
            Console.WriteLine("结果是" + md.EndInvoke(Iar));
        }
    }
    private static void Method()
    {
        Console.WriteLine("主线程方法开始");
        System.Threading.Thread.Sleep(3000);
        Console.WriteLine("主线程方法结束");
    }
}

```

可以看到我上面的注释行，去掉远程调用的注释，对下面的本地调用注释，编译后启动服务端，再启动客户端就是远程调用了。



异步调用结束，立即就能显示出结果，如果开启远程方法的话，可以看的更加清晰：
 客户端：主线程方法开始—》服务端：异步方法开始—》服务端：异步方法结束—》客户端：
 结果是 3—》客户端：主线程方法结束—》客户端：用了 3.03125 秒。

3、单向异步就是像同步调用方法那样调用方法，方法却是异步完成的，但是不能获得方法的返回值而且不能像同步方法那样取得所调用方法的异常信息！对于不需要返回信息的长时间方法，我们可以放手让它去干就行了：

远程对象：

```
using System;
using System.Runtime.Remoting.Messaging;

namespace RemoteObject
{
    public class MyObject:MarshalByRefObject
    {
        [OneWay]
        public void ALongTimeMethodOneWay(int time)
        {
            Console.WriteLine("异步方法开始");
            System.Threading.Thread.Sleep(time);
            Console.WriteLine("异步方法结束");
        }
    }
}
```

[OneWay]属性是 Remoting.Messaging 的一部分，别忘记 using，下面看看客户端代码：

```
using System;

namespace RemoteClient
{
    class MyClient
    {
        [STAThread]
        static void Main(string[] args)
        {
            DateTime dt=DateTime.Now;
            RemoteObject.MyObject
app=(RemoteObject.MyObject)Activator.GetObject(typeof(RemoteObject.MyObject), System.Configuration.ConfigurationSettings.AppSettings["ServiceURL"]);
            //RemoteObject.MyObject app=new RemoteObject.MyObject();
            app.ALongTimeMethodOneWay(1000);
        }
    }
}
```

```
Method();  
Console.WriteLine("用了"+((TimeSpan)(DateTime.Now-dt)).TotalSeconds+"秒");  
Console.ReadLine();  
}  
  
private static void Method()  
{  
    Console.WriteLine("主线程方法开始");  
    System.Threading.Thread.Sleep(3000);  
    Console.WriteLine("主线程方法结束");  
}  
}
```

这次我们仅仅只能在远程调试,我们先让异步方法去做,然后就放心的做主线程的事情,其他不管了。

运行结果我描述一下:

客户端: 主线程方法开始—》服务端: 异步方法开始—》服务端: 异步方法结束—》客户端: 主线程方法结束—》客户端: 用了 3.8 秒。

上面说的三种方法,只是异步编程的一部分,具体怎么异步调用远程方法要结合实际例子,看是否需要用到方法的返回和主线程方法的运行时间与远程方法运行时间等结合起来考虑,比如上述的 `WaitHandle` 也可以用轮询来实现:

`while(!ar.IsCompleted==false) System.Threading.Thread.Sleep(10);`总的来说远程对象的异步操作和本地对象的异步操作是非常接近。

还可以参考 msdn 相关文章:

<http://msdn.microsoft.com/library/chs/default.asp?url=/library/CHS/cpguide/html/cpconasynchronousprogrammingdesignpattern2.asp>

2.2.7 一步一步学Remoting之六：事件（1）

周末又过去了要上班了,一大早起来继续写。

概念就不说了,具体参见 msdn 相关章节:

<http://msdn.microsoft.com/library/CHS/cpguide/html/cpconEvents.asp>

我们先来改造一下上次的程序,为上次的主线程方法添加事件,能不断的引发事件来汇报处理的进度:

```
public class MyEventArgs
{
    private int _rate;

    public int Rate
    {
        get
        {
            return _rate;
        }
    }

    public MyEventArgs(int rate)
    {
        this._rate = rate;
    }
}

public class MyObject
{
    public delegate void MyEventHandler(object sender, MyEventArgs e);
    public event MyEventHandler MyEvent;

    public void ALongTimeMethod(int time)
    {
        Console.WriteLine("主线程方法开始");
        for (int i = 0; i < 100; i++)
        {
            System.Threading.Thread.Sleep(time);
            OnMyEvent(new MyEventArgs(i));
        }
        Console.WriteLine("主线程方法结束");
    }

    protected void OnMyEvent(MyEventArgs e)
    {
        if (MyEvent != null)
        {
            MyEvent(this, e);
        }
    }
}
```

再来为事件添加处理程序：

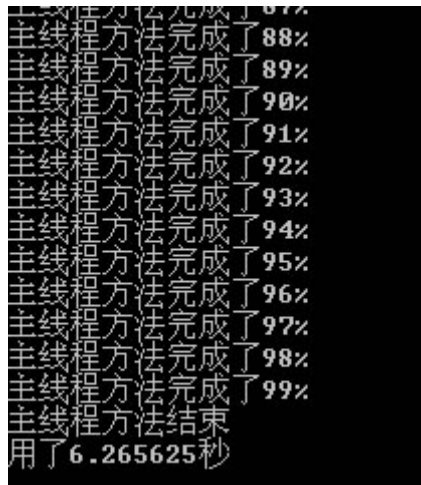

```

class MyClient
{
    [STAThread]
    static void Main(string[] args)
    {
        DateTime dt = DateTime.Now;
        MyObject obj = new MyObject();
        obj.MyEvent += new MyObject.MyEventHandler(obj_MyEvent);
        obj.ALongTimeMethod(50);
        Console.WriteLine("用了" + ((TimeSpan)(DateTime.Now - dt)).TotalSeconds + "秒");
        Console.ReadLine();
    }

    public static void obj_MyEvent(object sender, EventArgs e)
    {
        Console.WriteLine("主线程方法完成了" + e.Rate + "%");
    }
}

```

运行程序可以看到：



```

主线程方法完成了87%
主线程方法完成了88%
主线程方法完成了89%
主线程方法完成了90%
主线程方法完成了91%
主线程方法完成了92%
主线程方法完成了93%
主线程方法完成了94%
主线程方法完成了95%
主线程方法完成了96%
主线程方法完成了97%
主线程方法完成了98%
主线程方法完成了99%
主线程方法结束
用了6.265625秒

```

这个是本地的，远程对象的事件也这么简单吗？其实没有想象的简单，因为对象是在远程的，服务端的事件客户端怎么捕捉？应该说远程对象的事件可以分成客户端触发—》服务器应答，服务端触发—》客户端应答和客户端触发—》客户端应答，第一种就很简单了，后面2种都需要有一个中间件。下面我们来为远程对象同样来添加一个进度机制，首先来建立我们的远程对象：

```

[Serializable]
public class MyEventArgs : EventArgs
{

```

```
private int _rate;
private string _ip;

public int Rate
{
    get
    {
        return _rate;
    }
}

public string IP
{
    get
    {
        return _ip;
    }
}

public MyEventArgs(int rate, string ip)
{
    this._rate = rate;
    this._ip = ip;
}
}

public class MyObject : MarshalByRefObject
{
    public delegate void MyEventHandler(object sender, MyEventArgs e);
    public event MyEventHandler MyEvent;

    public int ALongTimeMethod(int a, int b, int time, string ip)
    {
        Console.WriteLine("异步方法开始");
        for (int i = 0; i < 10; i++)
        {
            System.Threading.Thread.Sleep(time);
            OnMyEvent(new MyEventArgs(i, ip));
        }
        Console.WriteLine("异步方法结束");
        return a + b;
    }

    protected void OnMyEvent(MyEventArgs e)
```

```
{  
    if (MyEvent != null)  
    {  
        MyEvent(this, e);  
    }  
}  
}
```

为了调试方便，服务器端和客户端这次都用程序实现，下面是服务器端：

```
using System;  
using System.Collections;  
using System.Runtime.Remoting;  
using System.Runtime.Remoting.Channels;  
using System.Runtime.Remoting.Channels.Tcp;  
using System.Runtime.Serialization.Formatters;  
  
namespace RemoteServer  
{  
    class MyServer  
    {  
        [STAThread]  
        static void Main(string[] args)  
        {  
  
            RemotingConfiguration.RegisterWellKnownServiceType(typeof(RemoteObject.MyObject), "RemoteObject.MyObject", WellKnownObjectMode.Singleton);  
  
            BinaryServerFormatterSinkProvider serverProvider =  
            new BinaryServerFormatterSinkProvider();  
  
            BinaryClientFormatterSinkProvider clientProvider =  
            new BinaryClientFormatterSinkProvider();  
  
            serverProvider.TypeFilterLevel = TypeFilterLevel.Full;  
            IDictionary props = new Hashtable();  
            props["port"] = 9999;  
            TcpChannel channel = new TcpChannel(props, clientProvider, serverProvider);  
            ChannelServices.RegisterChannel(channel);  
            Console.ReadLine();  
        }  
    }  
}
```

客户端为了简单一点，我去除了前面做测试的本地事件：

```

using System;
using System.Net;
using System.Collections;
using System.Text;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Serialization.Formatters;

class MyClient
{
    private delegate int MyDelegate(int a, int b, int time, string ip);
    private static MyDelegate md;

    [STAThread]
    static void Main(string[] args)
    {
        DateTime dt=DateTime.Now;
        RemotingConfiguration.RegisterWellKnownClientType(typeof(RemoteObject.MyObject),
"tcp://localhost:9999/RemoteObject.MyObject");
        BinaryServerFormatterSinkProvider serverProvider = new
BinaryServerFormatterSinkProvider();
        BinaryClientFormatterSinkProvider clientProvider = new
BinaryClientFormatterSinkProvider();
        serverProvider.TypeFilterLevel = TypeFilterLevel.Full;
        IDictionary props=new Hashtable();
        props["port"]=0;
        TcpChannel channel = new TcpChannel(props, clientProvider, serverProvider);
        ChannelServices.RegisterChannel(channel);
        RemoteObject.MyObject app=new RemoteObject.MyObject();
        app.MyEvent+=new RemoteObject.MyObject.MyEventHandler(MyEvent);
        md=new MyDelegate(app.ALongTimeMethod);
        AsyncCallback ac=new AsyncCallback(MyClient.CallBack);
        IPHostEntry ipHE=Dns.GetHostByName(Dns.GetHostName());
        IAsyncResult Iar=md.BeginInvoke(1, 2, 300, ipHE.AddressList[0].ToString(), ac, null);
        Method();
        Console.WriteLine("用了"+((TimeSpan)(DateTime.Now-dt)).TotalSeconds+"秒");
        ChannelServices.UnregisterChannel(channel);
        Console.ReadLine();
    }

    public static void CallBack(IAsyncResult Iar)
    {
        if(Iar.IsCompleted)

```

```

    {
        Console.WriteLine("结果是"+md.EndInvoke(Iar));
    }
}

public static void MyEvent(object sender, RemoteObject.MyEventArgs e)
{
    Console.WriteLine("来自"+e.IP+"的异步方法完成了"+e.Rate*10+"%");
}

public static void Method()
{
    Console.WriteLine("主线程方法开始");
    System.Threading.Thread.Sleep(5000);
    Console.WriteLine("主线程方法结束");
}
}

```

代码看上去不错，可是 debug 启动后报错：



这就是我前面提到的问题，远程不可能有本地的程序集，也无法触发本地事件。解决办法就是加一个事件中间件，继承 MarshalByRefObject：

```

public class EventClass : MarshalByRefObject
{
    public void MyEvent(object sender, MyEventArgs e)
    {
        Console.WriteLine("来自" + e.IP + "的异步方法完成了" + e.Rate * 10 + "%");
    }
}

```

然后来修改一下客户端： 把

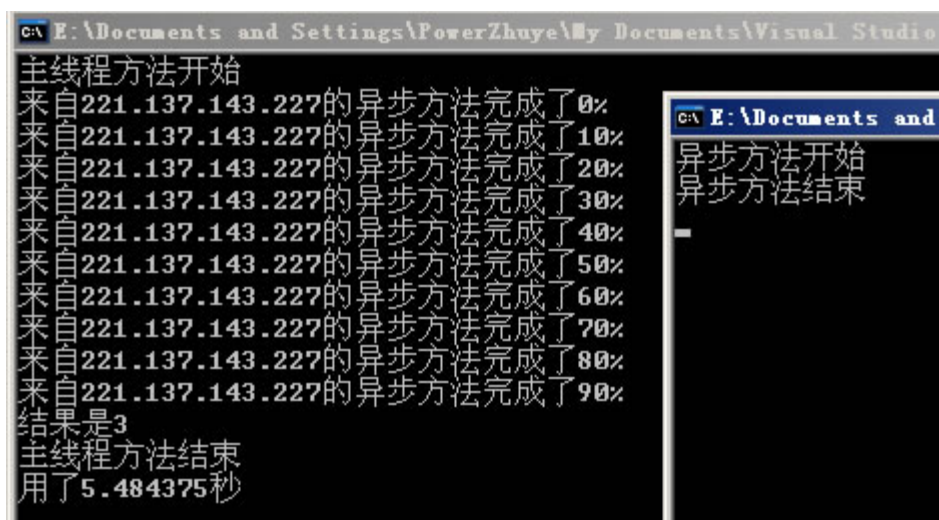
```
app.MyEvent+=new RemoteObject.MyObject.MyEventHandler(MyEvent);
```

修改为

```
RemoteObject.EventClass ec=new RemoteObject.EventClass();
app.MyEvent+=new RemoteObject.MyObject.MyEventHandler(ec.MyEvent);
```

删除客户端的 MyEvent 静态方法。

运行一下程序：



前后两个窗口本别是服务端和客户端的，貌似达到了我们的要求，其实不然，程序有 2 个漏洞：

- 1、客户端关闭以后打开新的程序就出错，因为以前的委托链丢失，服务端程序企图触发事件出错。
- 2、同时打开几个客户端，客户端收到的是所有的进度信息，而不仅仅是自己的，广播性质的消息。

不早了要上班了，在下一节中来改进这 2 点。

2.2.8 一步一步学Remoting之六：事件（2）

到了午休的时间，抓紧时间继续写，上次说有 2 个遗留问题：

（1）关闭一个客户端以后会影响其他的客户端事件

原因：客户端没有取消事件订阅就关闭了，触发事件的时候找不到事件订阅者

解决：遍历委托链，找到异常的对象，从委托链中卸下

（2）服务器端对客户端广播，客户端能收到其他客户端的事件处理信息

原因：使用了 Singleton 模式，共享远程对象

解决：因为需要远程对象有状态且不共享实例，所以只有客户端激活可以选择

修改后的服务端：

```
using System;
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Serialization.Formatters;

namespace RemoteServer
{
    class MyServer
    {
        [STAThread]
        static void Main(string[] args)
        {
            RemotingConfiguration.ApplicationName="RemoteObject.MyObject";

            RemotingConfiguration.RegisterActivatedServiceType(typeof(RemoteObject.MyObject));

            BinaryServerFormatterSinkProvider serverProvider =
                new BinaryServerFormatterSinkProvider();
            BinaryClientFormatterSinkProvider clientProvider =
                new BinaryClientFormatterSinkProvider();
            serverProvider.TypeFilterLevel = TypeFilterLevel.Full;
            IDictionary props = new Hashtable();
            props["port"]=8888;
            TcpChannel channel = new TcpChannel(props, clientProvider, serverProvider);
            ChannelServices.RegisterChannel(channel);
            Console.ReadLine();
        }
    }
}
```

修改后的远程对象：

```
using System;

namespace RemoteObject
{
    [Serializable]
    public class MyEventArgs:EventArgs
```

```
{

    private int _rate;
    private string _ip;

    public int Rate
    {
        get
        {
            return _rate;
        }
    }

    public string IP
    {
        get
        {
            return _ip;
        }
    }

    public MyEventArgs(int rate, string ip)
    {
        this._rate=rate;
        this._ip=ip;
    }
}

public class MyObject:MarshalByRefObject
{
    public delegate void MyEventHandler(object sender, MyEventArgs e);
    public event MyEventHandler MyEvent;
    public string tmp;

    public int ALongTimeMethod(int a, int b, int time, string ip)
    {
        Console.WriteLine("来自"+ip+"的异步方法开始");
        for(int i=1;i<=10;i++)
        {
            System.Threading.Thread.Sleep(time);
            Console.WriteLine("来自"+ip+"的异步方法完成了"+i*10+"%");
            OnMyEvent(new MyEventArgs(i, ip));
        }
        Console.WriteLine("来自"+ip+"的异步方法结束");
        return a+b;
    }
}
```



```

    }

    protected void OnMyEvent(MyEventArgs e)
    {
        if (MyEvent!=null)
        {
            foreach(Delegate d in MyEvent.GetInvocationList())
            {
                try
                {
                    ((MyEventHandler)d)(this, e);
                }
                catch
                {
                    MyEvent-= (MyEventHandler)d;
                }
            }
        }
    }
}

public class EventClass:MarshalByRefObject
{
    public void MyEvent(object sender, MyEventArgs e)
    {
        if(((MyObject)sender).tmp==e.IP)
            Console.WriteLine("异步方法完成了"+e.Rate*10+"%");
    }
}
}

```

修改后的客户端：

```

using System;
using System.Net;
using System.Collections;
using System.Text;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Serialization.Formatters;

class MyClient
{

```

```

private delegate int MyDelegate(int a, int b, int time, string ip);

private static MyDelegate md;

static RemoteObject.MyObject app;
static RemoteObject.EventClass ec;
static DateTime dt;

[STAThread]
static void Main(string[] args)
{
    dt=DateTime.Now;

RemotingConfiguration.RegisterActivatedClientType(typeof(RemoteObject.MyObject), "tcp://local
host:8888/RemoteObject.MyObject");

    BinaryServerFormatterSinkProvider serverProvider =
        new BinaryServerFormatterSinkProvider();
    BinaryClientFormatterSinkProvider clientProvider =
        new BinaryClientFormatterSinkProvider();
    serverProvider.TypeFilterLevel = TypeFilterLevel.Full;
    IDictionary props=new Hashtable();
    props["port"]=0;
    TcpChannel channel = new TcpChannel(props, clientProvider, serverProvider);
    ChannelServices.RegisterChannel(channel);
    app=new RemoteObject.MyObject();
    ec=new RemoteObject.EventClass();
    app.MyEvent+=new RemoteObject.MyObject.MyEventHandler(ec.MyEvent);
    md=new MyDelegate(app.ALongTimeMethod);
    AsyncCallback ac=new AsyncCallback(MyClient.CallBack);
    IPHostEntry ipHE=Dns.GetHostByName(Dns.GetHostName());
    Random rnd=new Random(System.Environment.TickCount);
    string ip=ipHE.AddressList[0].ToString()+"("+rnd.Next(100000000).ToString()+")";
    app.tmp=ip;
    IAsyncResult Iar=md.BeginInvoke(1, 2, 500, ip, ac, null);
    Method();
    Console.WriteLine("用了"+((TimeSpan)(DateTime.Now-dt)).TotalSeconds+"秒");
    ChannelServices.UnregisterChannel(channel);
    Console.ReadLine();
}

public static void CallBack(IAsyncResult Iar)
{
    if(Iar.IsCompleted)
    {
        Console.WriteLine("结果是"+md.EndInvoke(Iar));
        app.MyEvent-=new RemoteObject.MyObject.MyEventHandler(ec.MyEvent);
    }
}

```

```

    }
}

public static void Method()
{
    Console.WriteLine("主线程方法开始");
    System.Threading.Thread.Sleep(5000);
    Console.WriteLine("主线程方法结束");
}
}

```

之所以要在 ip 地址后面跟上随机数，是因为可能在一个机器上会打开多个客户端，需要在这个时候能在服务器端区分多个客户端。

The image displays two side-by-side console windows from a Windows command prompt. The left window, titled 'c:\ D:\solution\RemotingTest1\RemoteServer\bin\Release\RemoteS...', shows a continuous stream of text from the server side. It lists numerous IP addresses and ports (e.g., 10.9.99.240<57240921>) followed by messages like '的异步方法开始' (asynchronous method start) and '的异步方法完成了X%' (asynchronous method completed X%), where X ranges from 10 to 100. The right window, titled 'c:\ D:\solution\RemotingTest1\Remote...', shows the client side. It displays '主线程方法开始' (main thread method start), followed by progress updates for an asynchronous method (10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%), and finally '主线程方法结束' (main thread method end) along with the execution time '用了5.1342824秒' (took 5.1342824 seconds). A second client window below it shows similar progress and a final '结果是3' (result is 3).

备注：我的所有例子都是在客户端和服务端部署远程对象的，其实这个做法不是很好，我们应该仅仅把接口部署在两地，远程对象仅仅部署在服务器端即可。

3.1、版权声明

文章出处：<http://www.cnblogs.com/idor/archive/2007/03/06/665479.html>

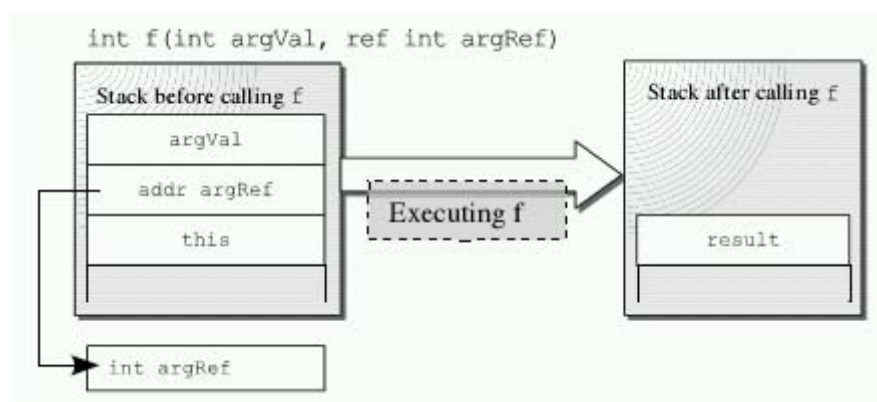
文章作者：idor

3.2、内容详情

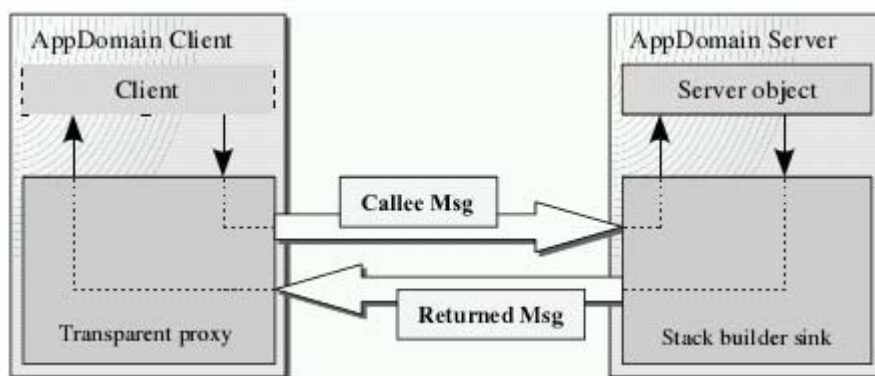
3.2.1 Remoting基本原理及其扩展机制（上）

.NET Remoting 是.NET 平台上允许存在于不同应用程序域中的对象相互知晓对方并进行通讯的基础设施。调用对象被称为客户端，而被调用对象则被称为服务器或者服务器对象。简而言之，它就是.NET 平台上实现分布式对象系统的框架。

传统的方法调用是通过栈实现，调用方法前将 `this` 指针以及方法参数压入线程栈中，线程执行方法时将栈中的参数取出作为本地变量，经过一番计算后，将方法的返回结果压入栈中。这样我们就完成了一次方法调用。如下图所示：



基于栈的方法调用在同一个应用程序域中很容易实现，但是如果调用的方法所属的对象位于另一个应用程序域或另一个进程甚至是另一个机器，又当如何？应用程序域之间是无法共享同一个线程栈的，此时我们将转而使用另一种方法调用机制——基于消息的方法调用机制。在客户端通过代理对象将原先基于栈的方法调用信息（定位远程对象的信息、方法名、方法参数等）封装到一个消息对象中，再根据需要将这个消息对象转化成某个格式的数据流发送到远程对象所在的的应用程序域中。当经过格式化的消息到达服务器后，首先从中还原出消息对象，之后在远程对象所在的的应用程序域中构建出相应方法调用栈，此时就可以按照传统的基于栈的方法调用机制完成方法的调用，而方法返回结果的过程则按照之前的方法反向重复一遍。如下图所示：



在基于消息的远程方法调用中主要有以下几个重要角色：

Client Proxy: 负责在客户端处理基于栈的参数传递模式和基于消息的参数传递模式之间的转换。

Invoker: 与 Client Proxy 的功能相反。

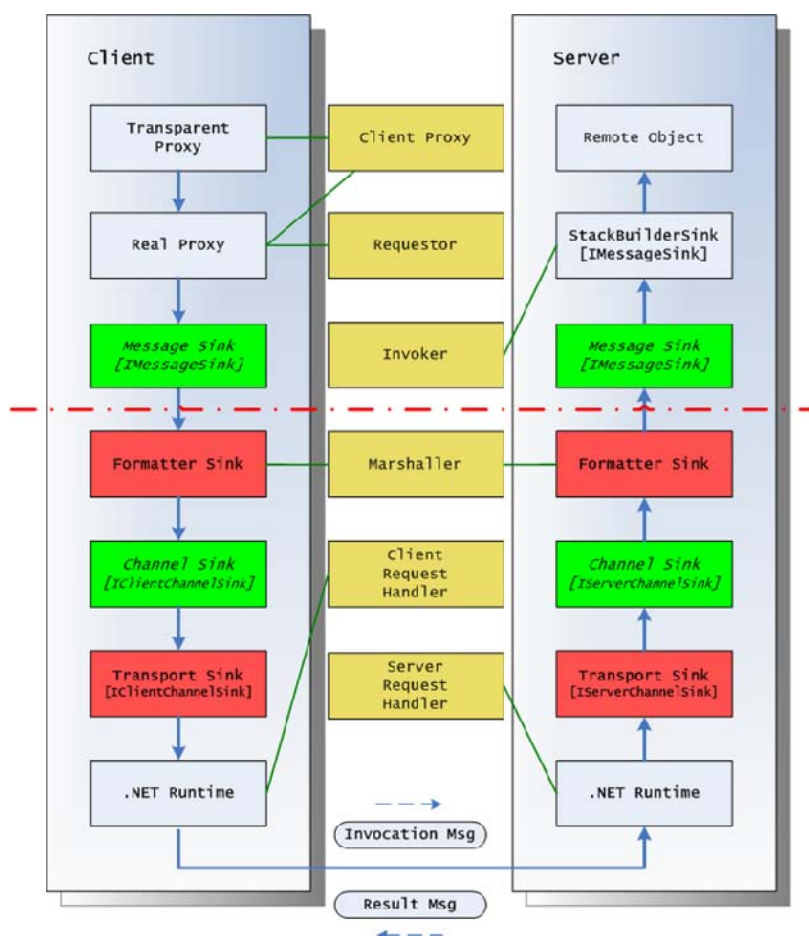
Requestor: 负责将消息对象转换成可在网络上传输的数据流，并将其发送到服务器。

Marshaller: 负责消息对象的序列化与反序列化。

Client Request Handle: 负责以数据流的格式发送客户端的请求消息。

Server Request Handel: 负责接收来自客户端的请求消息。

那么在.NET Remoting 框架下，这些重要角色又各自对应了哪些对象呢？下图是一个 Remoting 框架的示意图：



从中我们可以看到客户端的 **Transparent Proxy** 与服务器端的 **StackBuilderSink** 分别扮演了 **Client Proxy** 与 **Invoker** 的角色。**Remoting** 依靠这两个对象实现了基于栈的方法调用与基于消息的方法调用的转换，并且这一过程对于开发者是完全隐藏的。

Marshaller 的角色由 **Formmatter Sink** 完成，在 **Remoting** 中默认提供了两种 **Fommatter**：一个实现了消息对象与二进制流的相互转换，另一个实现了消息对象与 **Soap** 数据包的相互转换，而支持 **Soap** 格式则说明 **Remoting** 具有实现 **Web Service** 技术的可能。

Client Request Handle 与 **Server Request Handel** 都是 .NET 中实现网络底层通讯的对象，如 **HttpWebRequest**、**HttpServerSocketHandler** 等。在 **Remoting** 中我们并不直接接触这些对象，而是通过 **Channel** 对它们进行管理。在框架中默认提供了三种 **Channel**：**HttpChannel**、**TcpChannel** 与 **IpcChannel**。但是在上面这副图中，我们并没有看到 **Channel** 对象，那么 **Channel** 又是如何影响网络底层的通讯协议的呢？

其实在上面这幅图中，真正能对通讯时所采用的网络协议产生影响的元素是 **Transport Sink**，对应不同的协议 **Remoting** 框架中提供了三种共计六个 **Transport Sink**：**HttpClientTransportSink**、**HttpServerTransportSink** 与 **TcpClientTransportSink**、**TcpServerTransportSink** 以及 **IpcClientTransportSink**、**IpcServerTransportSink**，它们分别放置在客户端与服务器端。既然 **Transport Sink** 才是通讯协议的决定元素，那么 **Channel** 肯定与它有着某种联系，让我们先暂时搁置此话题，留待后面进一步介绍。

观察上面这副图，我们可以发现其中包含了一系列的 **Sink**，而所谓的 **Sink** 就是一个信息接收器，它接受一系列的输入信息，为了达到某种目的对这些信息做一些处理，然后将处理后信息再次输出到另一个 **Sink** 中，这样一个个的 **Sink** 串联起来就构成了一个 **Pipeline**（管道）。**Pipeline** 模式在分布式框架中经常可以看到，应用该模式可以使框架具有良好的灵活性。当我们需要构建一个系统用于处理并转换一串输入数据时，如果通过一个大的组件按部就班的来实现此功能，那么一旦需求发生变化，比如其中的两个处理步骤需要调换次序，或者需要加入或减去某些处理，系统将很难适应，甚至需要重写。而 **Pipeline** 模式则将一个个的处理模块相互分离，各自独立，然后按照需要将它们串联起来即可，此时前者的输出就会作为后者的输入。此时，每个处理模块都可以获得最大限度的复用。当需求发生变化时，我们只需重新组织各个处理模块的链接顺序，或者删除或加入新的处理模块即可。在这些处理模块（**Sink**）中最重要两类是 **Formatter Sink** 与 **Transport Sink**，也就是图 1 中的红色部分，当我们需要通过网络访问远程对象时，首先要将消息转化为可在网络上传播的数据流，然后需要通过特定的网络协议完成数据流的发送与接收，这正是这两类 **Sink** 所负责的功能。虽然它们本身也可以被自定义的 **Sink** 所替换，不过 **Remoting** 中提供的现有实现已经可以满足绝大多数的应用。

那么 .NET **Remoting** 中又是如何创建这个 **Pipeline** 的呢？以 **HttpChannel** 为例，在创建客户端的 **Pipeline** 的时候，会调用它的 **CreateMessageSink** 方法，此时又会进一步调用 **HttpClientChannel** 的构造函数，在构造函数中调用了一个名为 **SetupChannel()** 的私有方法，看一下它的实现：

```
private void SetupChannel()
```



```
{
    if (this._sinkProvider != null)
    {
        CoreChannel.AppendProviderToClientProviderChain(this._sinkProvider,
            new HttpClientTransportSinkProvider(this._timeout));
    }
    else
    {
        this._sinkProvider = this.CreateDefaultClientProviderChain();
    }
}
```

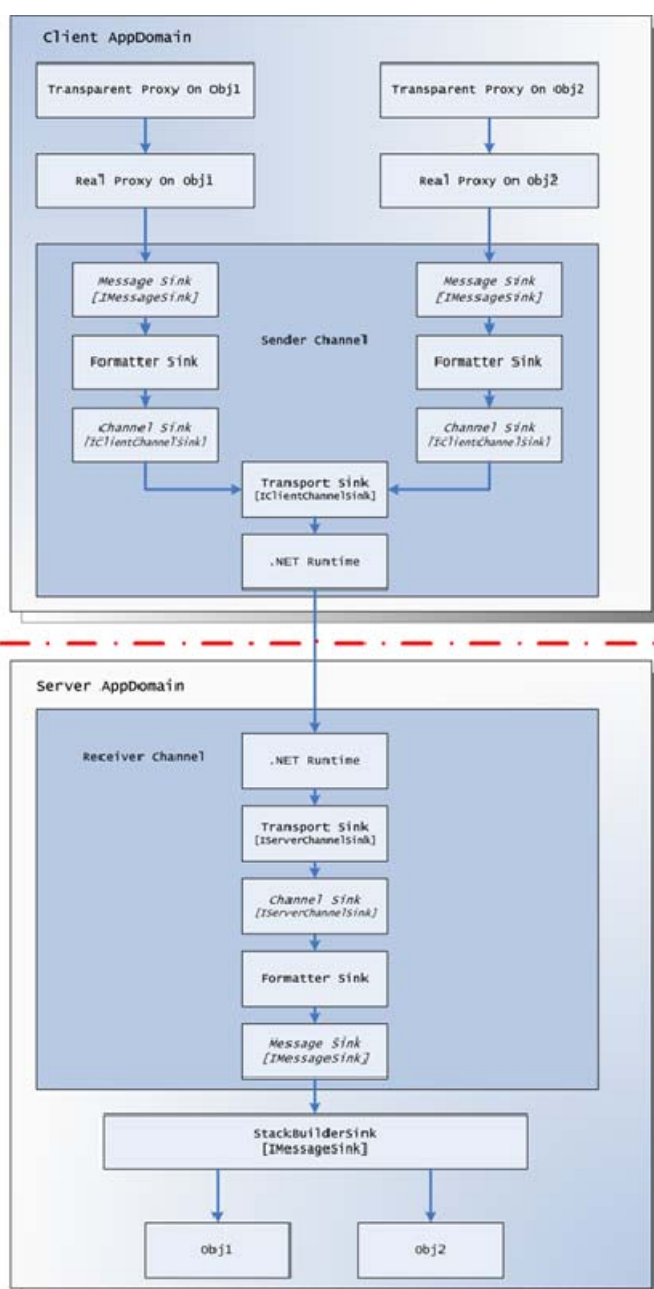
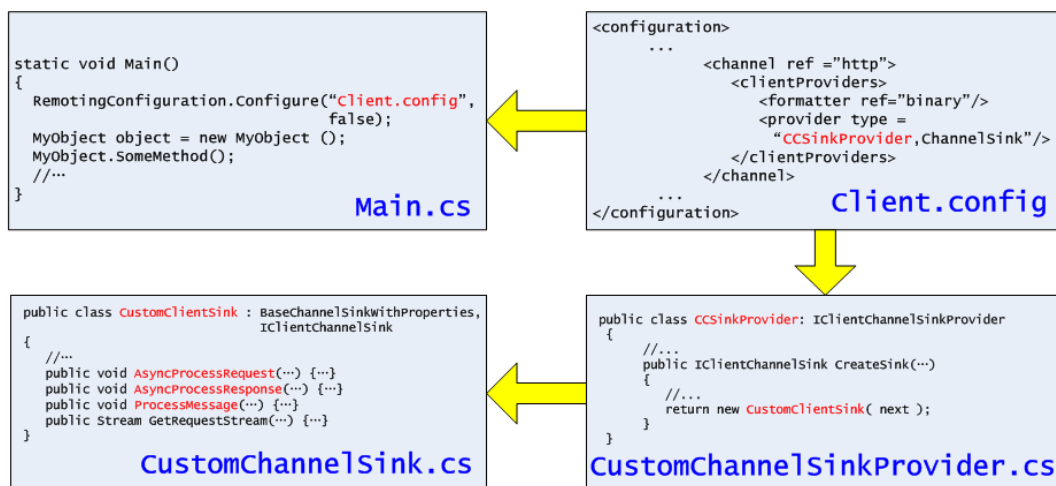
再看看在没有提供自定义 SinkProvider 的默认情况下 CreateDefaultClientProviderChain() 会创建出哪些 Sink

```
private IClientChannelSinkProvider CreateDefaultClientProviderChain()
{
    IClientChannelSinkProvider provider = new SoapClientFormatterSinkProvider();
    provider.Next = new HttpClientTransportSinkProvider(this._timeout);
    return provider;
}
```

其中包含了 SoapClientFormatterSinkProvider 与 HttpClientTransportSinkProvider，自然地，到了创建 Pipeline 的时候，它们会分别创建出 SoapClientFormatterSink 与 HttpClientTransportSink，前者用于实现消息对象的 Soap 格式化，而后者则表示将用 Http 协议来实现消息的通讯。这样我们就明白了为什么使用 HttpChannel 就会采用 Http 协议作为网络通讯协议，并且消息将以 SOAP 格式传递。但是仔细观察 SetupChannel 方法中的下段代码，我们可以发现通过提供自定义的 SinkProvider，我们可以改变消息的编码格式，因为此时只创建了一个 HttpClientTransportSinkProvider，并没有定义 FormatterSinkProvider，而 FormatterSinkProvider 完全可以在自定义的 SinkProvider 中自由设定。

```
if (this._sinkProvider != null)
{
    CoreChannel.AppendProviderToClientProviderChain(this._sinkProvider,
        new HttpClientTransportSinkProvider(this._timeout));
}
```

那么我们又如何能在 .NET Remoting 中定义自己的 SinkProvider 并让它发挥作用呢？下面这幅图演示如果使用自定义 SinkProvider 对 Pipeline 进行定制。



首先在配置文件中引用自定义的 `ChannelSinkProvider` 的类名及其所在程序集，然后编写自定义的 `ChannelSinkProvider` 的具体实现，也就是加入你需要的 `ChannelSink`，最后在自定义的 `ChannelSink` 中实现具体的处理操作。这样我们向 Pipeline 中成功地添加了自定义处理模块。这里需要提醒大家注意，接收方的 Pipeline 信道与发送方 Pipeline 之间存在着一个根本的差异。发送方 Pipeline 为每个远程对象的真实代理创建一个接收器（Sink）链。接收方信道在其创建之时创建了接收器（Sink）链，这条链将为所有通过这个接收方 Pipeline 进行转送的调用所使用，不论与这个调用相关联的对象是哪个。左图展示了这个差异。

细心的读者可能已经注意到之前我们定制的是 `ChannelSink`，为什么要加上一个 Channel 呢？让我们回过头去再来看看图 1，其中有两个绿色的 Sink，他们都是 Remoting 中可选的组件，也是我们对 Pipeline 进行扩展的地方。你可以在这两个地方加入自定义的 Sink，从而对流经 Pipeline 的数据做某些需要的处理，前者需实现

IMessageSink 接口，后者需要实现 IXXXChannelSink 接口，那么它们又有何不同呢？注意看图 1，我们会发现其中隔了一个 Formmatter Sink，而 Formmatter Sink 的作用就是将原来的.NET 消息变成可在网络上传播的数据流（可以是 Binary 或 Soap 格式），这也就表明前者的输入是消息对象，而后者的输入是数据流(Stream)。这也就决定了他们各自所能实现的扩展功能是不同的。比如，操作消息对象，我们可以把远程方法的参数变一变（比如从英文翻成中文），而操作数据流，我们可以实现数据流的加密或压缩之类功能。如何定制新的 ChannelSink 并将其加入到 Pipeline 中已经在图 2 中演示过了，那么 MessageSink 呢？请关注下节内容。

3.2.2 Remoting基本原理及其扩展机制（中）

在上一篇文章我们已经介绍到通过在配置文件中指定自定义的 ChannelSinkProvider，我们可以在 Pipeline 中加入自己的 ChannelSink，此时我们就可以加入自己的信息处理模块，但是这里我们所能操作的对象是已经经过格式化的消息（即数据流），我们看不到原始的消息对象，这也势必影响了我们所能实现的扩展功能。而在上文的图 1 中，我们看到除了 ChannelSink 可以扩展之外，我们还可以加入自定义的 MessageSink，而它是位于格式器之前的，也就是说在 MessageSink 中我们可以直接操作尚未格式化的消息对象。此时，我们就获得一个功能更强大的扩展点。直接操作消息对象，这意味着什么呢？简单来说，我们可以在这里实现方法拦截，我们可以修改方法的参数、返回值，在调用方法前后加入自己的处理逻辑。是不是觉得听上去很耳熟？没错，这就也正是 AOP 所要实现的一个目标。下面，在了解了整个 Remoting 的大背景以及 ChannelSink 的扩展机制后，我们将对 MessageSink 的扩展机制做进一步介绍。

在介绍前，我先提醒各位读者注意以下几点：

1. 确定你确实想深入了解 Remoting 的内部机制；
2. 确定你能很好的理解上一篇文章；
3. 如果说上一篇文章总结归纳的内容较多的话，在本文中出现的內容大多是笔者个人的探索，我想其他资料（包括英文资料）中都不曾介绍过这些内容，所以我不保证所有观点的正确性，如果你觉得哪里有误，也欢迎你在评论中提出你的意见。

下面就让我们开始品尝大餐吧。：)

利用 ChannelSinkProvider 扩展 MessageSink

MessageSink 的扩展有两种实现方法，让我先从简单的开始。在上一篇文章我们已经介绍到通过在配置文件中指定自定义的 ChannelSinkProvider，我们可以在 Pipeline 中加入自己的 ChannelSink。那么有没有一个类似于 IClientChannelSinkProvider 的 IMessageSinkProvider 呢？可惜答案是否定的。那么我们能否通过 IClientChannelSinkProvider 插入一个 MessageSink 呢？插入之后它又能否发挥其功效呢？

首先我们先实现一个自定义的 MessageSink。此时只需新建一个类，并实现 IMessageSink

接口中的 `SyncProcessMessage` 方法(为简单起见我们只考虑同步调用模式)，在方法中我们可以直接操作 `Message` 对象，比如我们可以向 `Message` 中加入额外的属性，如下所示：

```
public class CustomMessageSink : IMessageSink
{
    public IMessage SyncProcessMessage(IMessage msg)
    {
        // Add some custom data into msg.
        ((IMethodMessage)msg).LogicalCallContext.SetData("MyName", "idior");
        return m_NextSink.SyncProcessMessage(msg);
    }
}
```

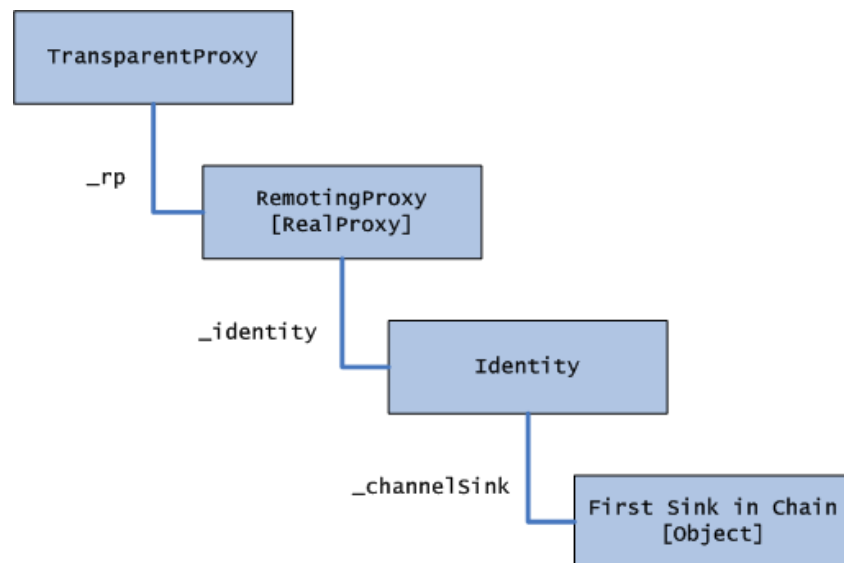
在上一篇文章的图 2 中我们可以看到 `IClientChannelSinkProvider` 是通过下面这个方法创建 `ChannelSink`。

```
public IClientChannelSink CreateSink(IChannelSender channel, string url,
object remoteChannelData) {...}
```

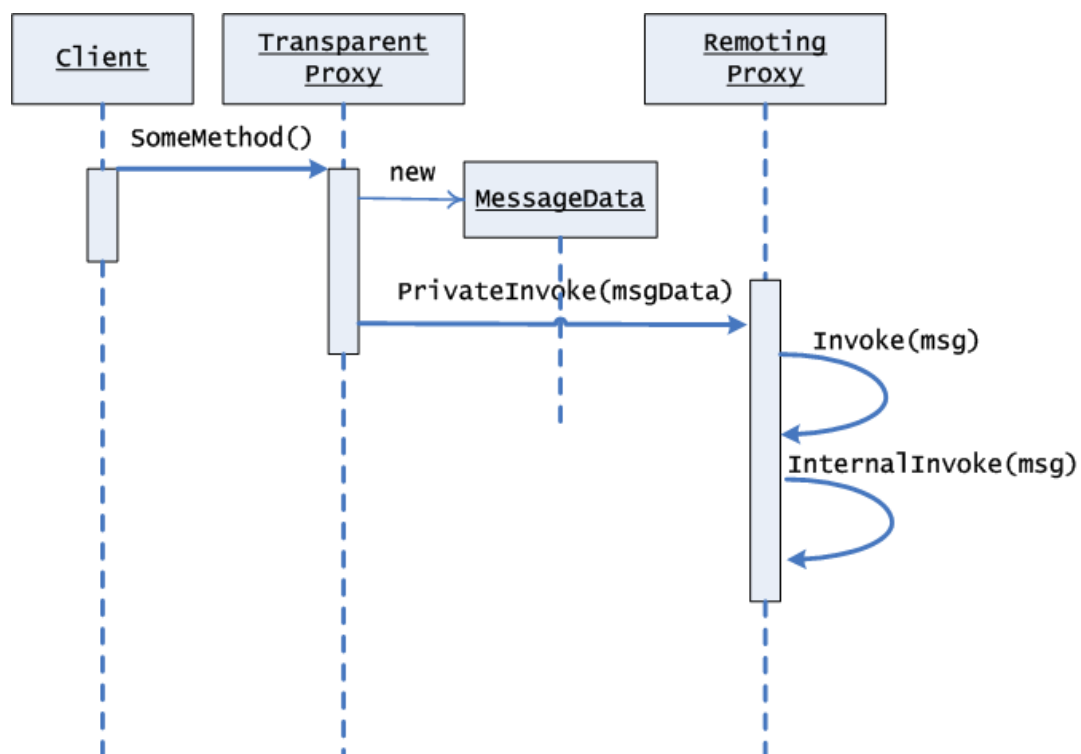
注意它的返回值是 `IClientChannelSink`，而不是 `IMessageSink`，这样我们就无法将仅实现了 `IMessageSink` 接口的 `CustomMessageSink` 插入。为此，我们让 `CustomMessageSink` 也实现 `IClientChannelSink` 接口，只不过在实现 `IClientChannelSink` 接口中的方法时，我们全部抛出异常，以表示这些方法不应该被调用到。这样我们就可以瞒天过海般地利用 `ChannelSinkProvider` 创建出一个 `MessageSink`。现在问题来了，这个 `MessageSink` 虽然创建出来了，但是它被插入 `Pipeline` 了吗？其实，我们在上一篇文章中就漏过了一个问题——那些利用 `ChannelSinkProvider` 创建出来的 `ChannelSink` 是如何被插入到 `Pipeline` 中的，明白了它的原理，就自然解决了上面的问题。

Pipeline

`Pipeline` 是何物？我们并没有解释清楚这个概念，是否存在一个对象它就叫 `Pipeline` 或者类似的名字？遗憾地告诉你，没有！可以说这里的 `Pipeline` 是一个抽象的概念，它表示了当我们调用一个远程对象时从 `RealProxy` 到 `StackBuildSink` 之间所经过的一系列 `Sink` 的集合，但是并不存在一个单独的链表把这些 `Sink` 全部链接起来。也就是说并不存在一个大的 `Sink` 链表，当你触发远程方法后，我们就依次从这个链表中取出一个个的 `Sink`，大家挨个处理一下消息。不过在远程对象的代理中倒是维护了一个由 `ChannelSink` 组成的链表。不过需要注意它并不代表整个 `Pipeline`，而只能算是其中一部分，在后面我们会看到 `Pipeline` 中还包括了很多其他类型的 `Sink`。这个链表保存在 `RealProxy` 的 `_identity` 对象中，链表是通过 `IClientChannelSink` 的 `Next` 属性链接起来的，在 `_identity` 对象中保存链表的第一个元素，其他元素可以通过 `Next` 属性获得，如下图所示：



下面我们来看看这个链表是如何得到的。每当我们通过 TransparentProxy 调用远程方法时，如下图所示，最终会调用到 RemotingProxy 中的 InternalInvoke 方法，它将负责把各个 ChannelSink 创建出来并链接在一起。



```

internal virtual IMessage InternalInvoke(IMethodCallMessage reqMcmMsg,
                                         bool useDispatchMessage, int callType)
{
    //...
    if (identity.ChannelSink == null)
    
```

```

{
    IMessageSink envoySink = null;
    IMessageSink channelSink = null;
    if (!identity.ObjectRef.IsObjRefLite())
    {
        RemotingServices.CreateEnvoyAndChannelSinks(null, identity.ObjectRef,
                                                    out envoySink , out channelSink );
    }
    else
    {
        RemotingServices.CreateEnvoyAndChannelSinks(identity.ObjURI, null,
                                                    out envoySink , out channelSink );
    }
    RemotingServices.SetEnvoyAndChannelSinks(identity, envoySink, channelSink );
    if (identity.ChannelSink == null)
    {
        throw new RemotingException("...");
    }
}
//...
}

```

第一个判断语句（Line 5）说明创建并链接 ChannelSink 的工作只发生在第一次调用，以后的每次调用将重复使用第一次的结果。第二个判断语句（Line 9）暂且不管，我只需知道在下一步将创建出两个 Sink 链，一个是 EnvoySink 链，而另一个是 ChannelSink 链，前者我们也先不去管它（将在下部中介绍）而后者将通过 out 关键字传给局部变量 channelSink。其中 CreateEnvoyAndChannelSinks 方法最终会把 ChannelSink 链的创建任务交给 Channel 对象，至于 Channel 对象是如何配合 ChannelSinkProvider 工作的，我们在上一篇文章中已经介绍过了。

不知你有没有注意到局部变量 channelSink（Line 8）此时的类型是 IMessageSink 而不是 IClientChannelSink。到关键地方了，大家提起精神啊！明明我们创建的是 ChannelSink 链却把头元素的类型设为 IMessageSink。这是为什么？大家知道在采用 HttpChannel 时，ChannelSink 链的一个元素是什么吗？——SoapClientFormatterSink。你认为它应该是一个 Message Sink 还是 Channel Sink？它是负责将消息对象格式为数据流的，操作对象是原始消息，自然应该是一个 MessageSink。呵呵，原来搞了半天 Remoting 本身就有利用 IClientChannelSinkProvider 扩展 MessageSink 的例子（你可以在类库中找到 SoapClientFormatterSinkProvider）。如之前所述，SoapClientFormatterSink 虽然是一个 MessageSink，但是为了利用 IClientChannelSinkProvider 将其插入到 Pipeline 中，它也不得不实现 IClientChannelSink 接口，而且你可以看到它在实现 IClientChannelSink 接口中的方法时，全部抛出异常。如下所示：

```

public class SoapClientFormatterSink :IMessageSink, IClientChannelSink
{

```

```

//...

//Implement method in IMessageSink
public IMessage SyncProcessMessage(IMessage msg)
{
    IMethodCallMessage message1 = (IMethodCallMessage) msg;
    try
    {
        ITransportHeaders headers1;
        Stream stream1;
        Stream stream2;
        ITransportHeaders headers2;
        this.SerializeMessage(message1, out headers1, out stream1);
        this._nextSink.ProcessMessage(msg, headers1, stream1,
                                     out headers2, out stream2);

        if (headers2 == null)
        {
            throw new ArgumentNullException("returnHeaders");
        }
        return this.DeserializeMessage(message1, headers2, stream2);
    }
    catch (Exception exception1)
    {
        return new ReturnMessage(exception1, message1);
    }
    catch
    {
        return new ReturnMessage(new Exception("..."), message1);
    }
}

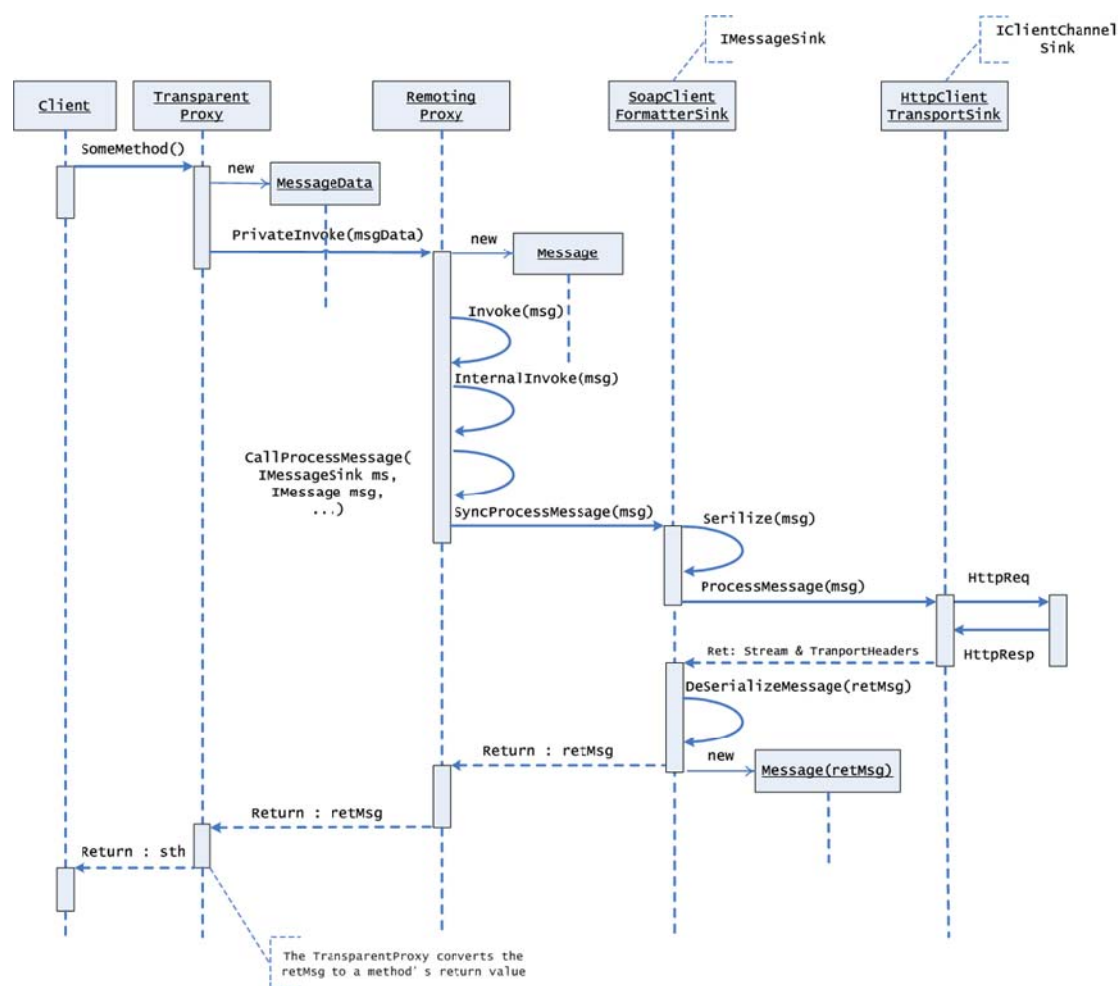
//Implement method in IClientChannelSink
public void ProcessMessage(...)
{
    throw new NotSupportedException();
}

//...
}

```

然后在 `InternalInvoke` 方法中（代码 3），我们又通过 `SetEnvoyAndChannelSinks` 方法（Line19）把之前赋值的局部变量 `channelSink` 赋给 `RemotingProxy` 对象中 `identity` 对象的 `_channelSink` 变量，这样一个个 `ChannelSink` 被链接在一起并能被代理对象所访问。

现在我们可以确定通过 `IClientChannelSinkProvider` 完全可以向 `Pipeline` 中插入新的 `MessageSink`。由于 `SoapClientFormatterSink` 的存在，我们也完全可以相信这个被插入到 `ChannelSink` 链中的 `MessageSink` 能正常的工作（即执行 `IMessageSink` 中的方法，而不是 `IClientChannelSink` 中的方法），不过为了让大家更清楚 `Remoting` 的底层实现，我们还是想探究一下它是如何调用 `ChannelSink` 链中的一个 `Sink` 来处理消息的。下图就是调用一次远程方法所产生的序列图：



在上图中，我们可以看到在 `InternalInvoke` 方法中将调用 `CallProcessMessage` 方法，它会把消息对象交给 `ChannelSink` 链中的第一个 `Sink` 处理。如下所示：

```
RemotingProxy.CallProcessMessage(identity.ChannelSink, reqMsg, ...);
```

而我们在上图中可以发现 `CallProcessMessage` 方法的第一个形参是 `IMessageSink` 类型的。也就是说通过 `IClientChannelSinkProvider` 方式插入到 `Pipeline` 中的第一个 `Sink`，反倒是 `IMessageSink` 类型的，而不是 `IClientChannelSink`。这也为插入到 `ChannelSink` 链中的 `MessageSink` 能正常工作扫清了障碍。正是因为这个原因 `SoapClientFormatterSink` 才能发挥其作用。

另外在利用 `IClientChannelSinkProvider` 插入 `MessageSink` 的时候，必须将它插入到 `FormatterSink` 的前面。因为只有在消息被 `Format` 之前，我们才能通过 `MessageSink` 对它进行处理，`Format` 之后在 `Sink` 对消息的修改就无效了。这点在配置文件中体现为自定义的 `SinkProvider` 必须放在 `Formatter` 前面。不过这是针对客户端而言，服务器端则恰恰与此相反。

```
<channel ref="http">
  <clientProviders>
    <provider type="CustomSinks.CustomSinkProvider,CustomSinks" />
    <formatter ref="soap" />
  </clientProviders>
</channel>
```

而在客户端插入 `ChannelSink` 时，自定义的 `SinkProvider` 都是放在 `Formatter` 后面的。你可以在上一篇文章的图 2 中发现这点。

总结

在本节中主要介绍了如何利用 `IClientChannelSinkProvider` 向 `Pipeline` 中加入 `MessageSink`，从而在远程方法调用中修改消息对象，实现功能更强大的扩展。并由此介绍了 `Remoting` 在实现此功能时，它的内部实现机制，有助于大家更深入地了解 `Remoting` 框架。

下一节将介绍当 `Client` 和 `Server` 对象处在同一个 `Appdomain` 时，如何拦截并修改消息，其中将涉及到更多类型的 `Sink`。

3.2.3 Remoting基本原理及其扩展机制（下）

让我们在开始本节内容之前先了解以下几个基本概念。

应用程序域

应用程序域（通常简称为 `AppDomain`）可以视为一种轻量级进程。一个 `Windows` 进程内可以包含多个 `AppDomain`。`AppDomain` 这个概念的提出是为了实现在一个物理服务器中承载多个应用程序，并且这些应用能够相互独立。`ASP.NET` 中利用 `AppDomain` 在同一个进程内承载了多组 `Web` 应用程序就是一个例子。实际上微软曾进行过在单一进程内承载多达 1000 个简单 `Web` 应用程序的压力测试。

使用 `AppDomain` 所获得的性能优势主要体现在两方面：

- 创建 `AppDomain` 所需要的系统资源比创建一个 `Windows` 进程更少。
- 同一个 `Windows` 进程内所承载的 `AppDomain` 之间可以互相共享资源，如 `CLR`、基本 `.NET` 类型、地址空间以及线程。

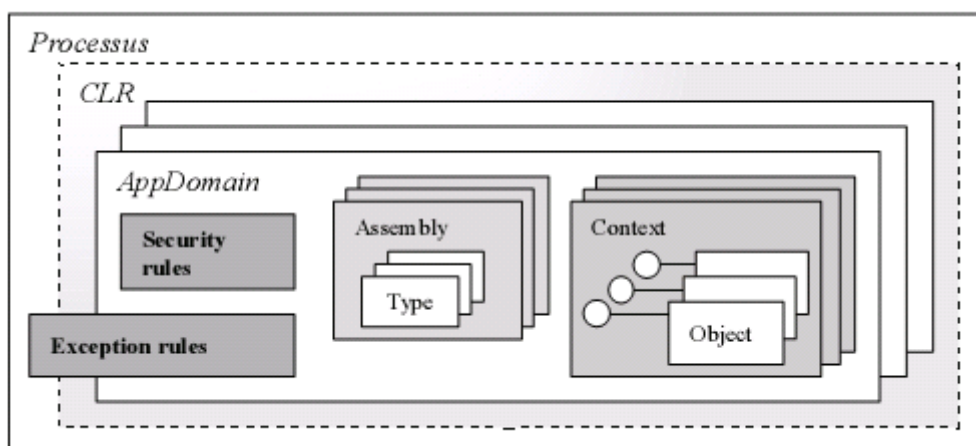
而各个 AppDomain 之间的独立性体现为以下这些特征：

- 一个 AppDomain 可以独立于其他的 AppDomain 而被卸载。
- 一个 AppDomain 无法访问其他 AppDomain 的程序集和对象。
- 若没有发生跨边界的异常抛出，一个 AppDomain 拥有自己独立的异常管理策略。这意味着一个 AppDomain 内出现问题不会影响到同一个进程内中的其他 AppDomain。
- 每个 AppDomain 可以分别定义独自的程序集代码访问安全策略。
- 每个 AppDomain 可以分别定义独自的规则以便 CLR 在加载前定位程序集所在位置。

可以看出应用程序域是进程中的一个子单元，不过在 .NET 中还存在一个比应用程序域还要细粒度的单元——.NET 上下文(Context)。

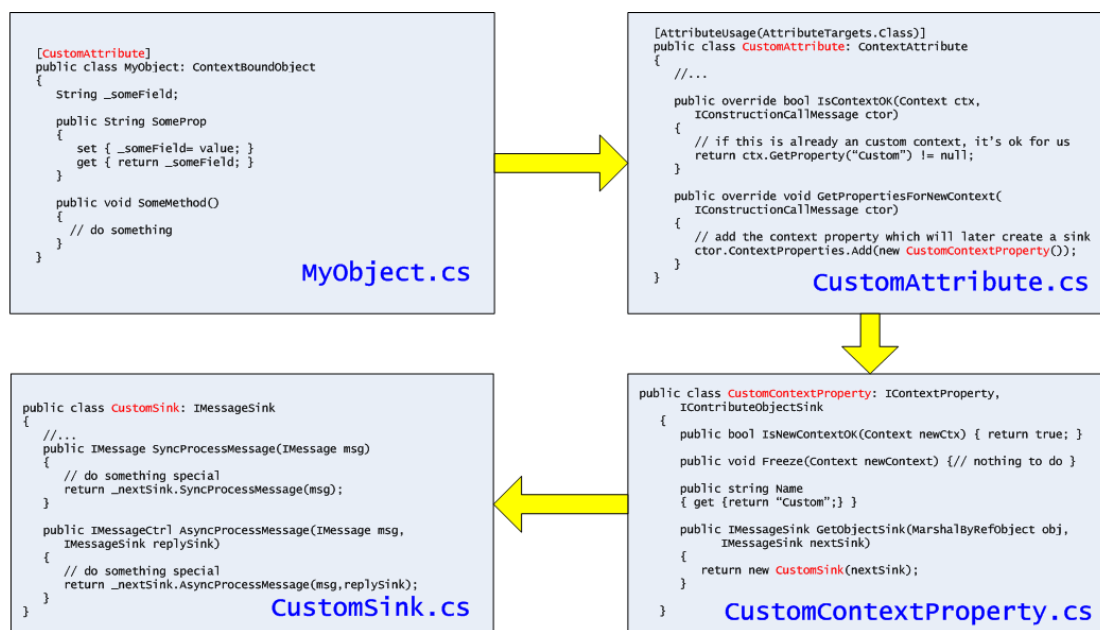
.NET Context

一个 .NET 应用程序域能够包含多个被称为 .NET 上下文的实体。所有 .NET 对象都存在于上下文中，每个应用程序域中至少存在一个上下文。这个上下文称为应用程序域的默认上下文，它在应用程序域创建的时候就创建了。下图总结了它们之间的关系：



那么 MessageSink 与上下文有什么关系呢？我们知道在通常情况下，如果访问同一个 AppDomain 中对象的方法时，会采用基于栈的方式（详见本系列上部）。在这种情况下，我们是无法拦截其中的消息的，因为此时根本不存在消息对象。只有当我们通过 Transparent Proxy 访问另一个对象的方法时，才会采用基于消息的方式。而现在我们只知道当一个对象调用处在另一个 AppDomain 中的远程对象（该对象为 MarshalByRefObject 子类）时，Remoting 才会为调用方创建那个远程对象的 Transparent Proxy。在了解了 .NET 上下文的概念后，你会发现，即使处在同一个 AppDomain 中的两个对象，如果它们所处的上下文不同，在访问对方的方法时，也会借由 Transparent Proxy 实现，即采用基于消息的方法调用方式。此时，我们就可以在上下文中插入 MessageSink 了。那么在上下文中是否存在类似 IClientChannelSinkProvider 的接口呢？很幸运，在经过一番探索后，我们发现确实存在类似的接口，而且还不止一个：IContributeEnvoySink、IContributeServerContextSink、IContributeObjectSink、IContributeClientContextSink 这四个接口中各自包含了一个 GetXXSink 的方法，它们都会返回一个实现了 IMessageSink 接口的对象。我们知道

IClientChannelSinkProvider 接口是配合配置文件最终实现向 Pipeline 中加入 ChannelSink 的，而以上这四个接口并没有配合配置文件使用，不过与 IClientChannelSinkProvider 的使用方式倒也有异曲同工之效。读者可以将下面这幅图与本系列上部中的图 2 比较一番。

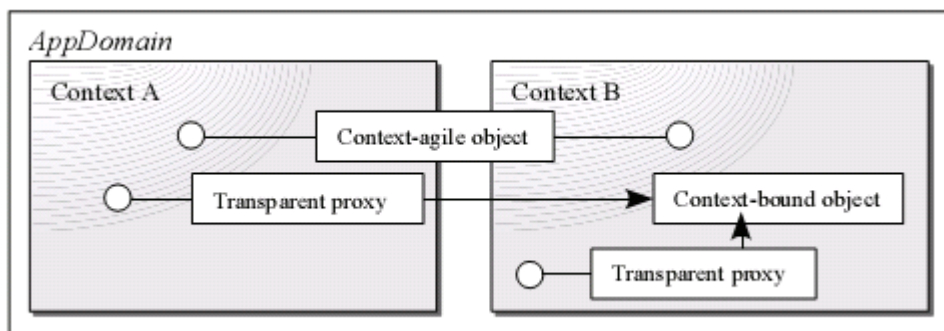


在上图中又出现了很多新的概念，下面将对它们一一做出解释：

ContextBoundObject

上下文可以看作应用程序域中一个包含对象和消息接收器的区域。对上下文里的对象的调用会转换成可以被 MessageSink（消息接收器）拦截和处理的消息。我们知道要把调用转换成消息，必须通过透明代理这个中介。而且，仅当对象是 MarshalByRefObject 的子类的实例并被其所在的应用程序域以外的实体调用时，CLR 才会为它创建透明代理。这里，我们希望对所有调用使用消息接收器机制，即使那些调用是来自同一个应用程序域中的实体。这个时候我们就需要用到 System.ContextBoundObject 类了。继承自 ContextBoundObject 的类的实例同样仅能由透明代理访问。此时，即使在这个类的方法中使用的 this 引用也是透明代理而不是对这个对象的直接引用。我们会发现 ContextBoundObject 类继承自 MarshalByRefObject，这非常合理，因为它很好地强调了该类的特性——它告诉 CLR 这个类将会通过透明代理使用。

ContextBoundObject 的子类的实例被视为上下文绑定的（context-bound）。没有继承自 ContextBoundObject 的类的实例则被视为上下文灵活的（context-agile）。上下文绑定的对象永远在其上下文中执行。只要不是远程对象，上下文灵活的对象总是在执行这个调用的上下文中执行。如下图所示：



ContextAttribute

上下文 attribute 是应用在上下文绑定的类上的.NET attribute。上下文 attribute 类实现了 `System.Runtime.Remoting.Contexts.IContextAttribute` 接口。上下文绑定的类可以应用多个上下文 attribute。在这个类的对象创建期间，这个类的每个上下文 attribute 判断这个对象的创建者所在的上下文是否适用。该操作通过以下方法完成：

```
public bool IContextAttribute.IsContextOK(Context clientCtx,
    IConstructionCallMessage ctorMsg)
```

只要其中一个上下文 attribute 返回 `false`，CLR 就必须创建一个新的上下文来容纳这个新的对象。这样，每个上下文 attribute 可以在这个新的上下文中注入一个或多个上下文属性。这些注入通过以下方法完成：

```
public void IContextAttribute.GetPropertiesForNewContext(
    IConstructionCallMessage ctorMsg)
    IContextProperty
```

上下文属性是实现 `System.Runtime.Remoting.Contexts.IContextProperty` 接口的类的实例。每个上下文可以包含多个属性。上下文属性在上下文创建的时候通过上下文 attribute 注入。一旦每个上下文 attribute 注入了它的属性，就会为每个属性调用下面的方法。此后就无法在这个上下文中注入另外的属性了：

```
public void IContextProperty.Freeze( Context ctx )
```

然后，CLR 通过调用下面的方法判断新的上下文能否满足每个属性：

```
public bool IContextProperty.IsNewContextOK( Context ctx )
```

每个上下文属性都有一个通过 `Name` 属性定义的名称：

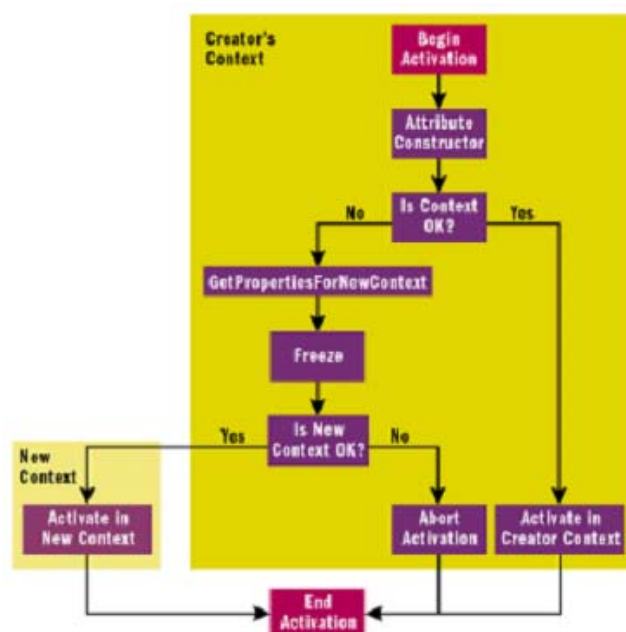
```
public string IContextProperty.Name{ get }
```

上下文中承载的对象的方法可以通过调用下面的方法访问上下文属性：

```
IContextProperty Context.GetProperty( string sPropertyName )
```

这一点很有意思，上下文中的对象通过它们所在的上下文的属性可以共享信息并访问服务。不过，上下文属性的主要作用并不在于此。上下文属性的主要作用在于向相关上下文中的消息接收器区域注入消息接收器（MessageSink）。（消息接收器区域的概念将在后面介绍）

以上注入 MessageSink 的过程可以用下图概括：

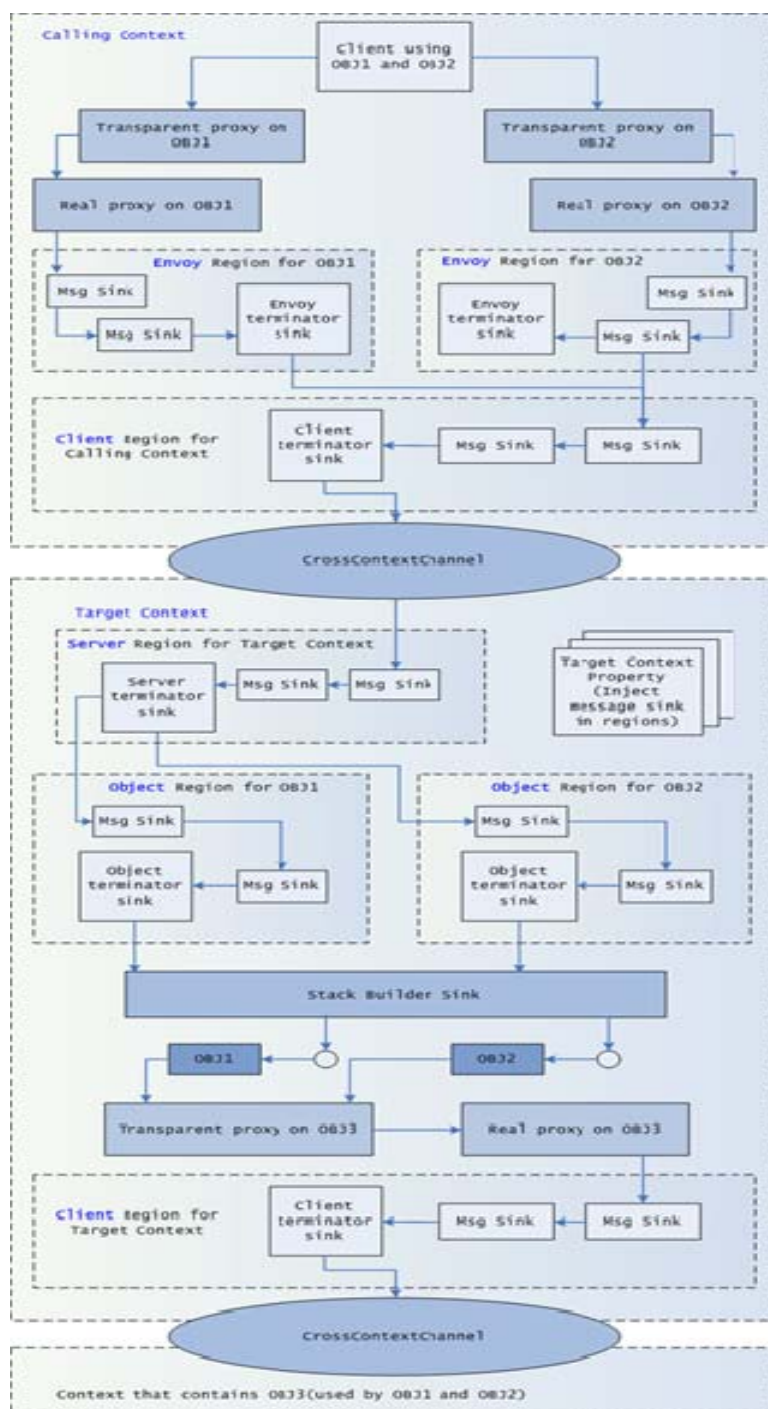


MessageSink Region

不知你是否记得之前提到的四个接口：IContributeEnvoySink、IContributeServerContextSink、IContributeObjectSink、IContributeClientContextSink。其实它们分别代表了四个不同的消息接收器区域：服务器（server）区域、对象（object）区域、信使（envoy）区域和客户端（client）区域。要理解区域概念，你必须考虑上下文绑定的对象是否被位于另一个上下文的实体调用。这个实体可以是一个静态方法或者另一个对象。在我们关于区域的讨论中，我们把这个实体所在的上下文称为调用方上下文（calling context），而把被调用对象所在的上下文称为目标上下文（target context）。目标上下文中的每个属性都可以在这些区域中注入消息接收器。

- 注入服务器区域的消息接收器拦截所有从另一个上下文发往目标上下文中所有对象的调用消息。于是，每个目标上下文有一个服务器区域。
- 注入对象区域的消息接收器拦截所有从另一个上下文发往目标对象中特定对象的调用消息。于是，上下文中每个对象会有一个对象区域。
- 注入信使区域的消息接收器拦截所有从另一个上下文发往目标对象中特定对象的调用消息。信使区域和对象区域的不同点是信使区域位于调用方上下文而不是包含对象的目

- 标上下文。我们使用信使区域把调用方上下文的信息传递给目标上下文的消息接收器。
- 注入客户端区域的消息接收器拦截所有从目标上下文发往位于其他上下文的对象的调用消息。于是，每个目标上下文有一个客户端区域。你可能会对这个区域所处的位置有点困惑，似乎当它位于 Calling context 的信使区域下方时会显得更加对称。之所以会有这样的误解，是因为我们对 Server、Client 的理解有了偏差。你应该记住除了信使区域是位于 Calling context 外，另外三个区域都是处在 Target Context。而所谓的 Server、Client 是针对处在 Target Context 中的对象在某不同时刻所扮演不同角色而言的。当然 Calling context 中也会有客户端区域，不过其中的 MessageSink 不是通过 Target context 的属性注入的，而应该依靠 Calling context 中的上下文属性注入。



上图说明了区域的概念。目标上下文包含名为 **OBJ1** 和 **OBJ2** 的两个对象。我们选择在目标上下文中放置两个对象而不是一个是为了更好地说明对象区域和信使区域是在对象层面与消息的拦截关联起来的，而服务器区域和客户端区域则是在上下文层面与消息的拦截关联起来的。

我们在每个区域中放置了两个自定义消息接收器是为了更好地说明一个区域能包含零个、一个或多个消息接收器。具体地说，所有自定义消息接收器都通过目标上下文的属性注入区域，即使这个区域不属于目标上下文。因为你可以定义你自己的上下文属性类，你可以选择必须注入哪个消息接收器。

你可能注意到每个区域都包含一个用于通知 **CLR** 退出区域的系统终结器接收器(system terminator sink)，它是由 **Remoting** 框架定义的，并且总是位于每个区域的末尾。

当调用方上下文和目标上下文处在同一个应用程序域中时，**CLR** 会使用 **mscorlib.dll** 中 **CrossContextChannel** 内部类的实例作为信道。这个实例会使得当前线程的 **Context** 属性发生切换。图中也展示了这一实例。