第 22 章 .NET Remoting

22

22.1 简介

22.1.1 什么是.NET Remoting

在4.1.1节，我们把AppDomain的概念定义为运行期间程序集的容器。需要提醒的是，一个进程包含着一个或多个AppDomain，CLR将这些AppDomain相互隔离开来。实际上，包含.NET平台基础类型的mscorlib程序集会被装载到进程中一块不属于任何AppDomain的区域。AppDomain之间的隔离通常是在类型、安全和异常管理层面完成的，而不是在线程层面。

如果掌握了AppDomain的概念，那么应该很容易理解下面对.NET Remoting的定义。

.NET Remoting是.NET平台上允许存在于不同AppDomain中的对象相互了解对方并相互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。

两个不同的应用程序域可能存在于：

- 同一进程；
- 同一机器上的两个不同进程；
- 不同机器上的两个不同进程。

.NET Remoting对开发人员隐藏了这三个方面的问题。对象的定位通常是在源代码编译之后的应用程序的安装期间完成的。更确切地说，“某台机器上的某个对象”这样的信息不应该在源代码中出现。而应该存放在客户端对象能够访问得到的配置文件中。

对于曾经使用过微软技术的开发人员来说，.NET Remoting可以看作是DCOM的继任者。

22.1.2 FAQ

问题：我们可以选择用于两个对象之间通讯的底层协议吗？

回答：可以。在.NET Remoting下，每个通信协议都封装成一个信道对象。.NET Framework已经实现了封装HTTP、TCP以及用于同一机器上的进程通信协议（IPC是Inter Process Communication的缩写）的信道。也可以开发用于其他协议的信道。

问题：方法调用所需的数据应以何种形式在网络中传输？

回答：方法调用所需的数据主要包含输入参数的值、方法的标识符、调用前对象的标识符和由调用产生的输出参数。在.NET Remoting中，负责包装这些数据的对象称为格式化程序。.NET Framework已经实现了把数据包装成二进制形式或者名为SOAP的XML格式形式的格式化程序。不过，也可以开发自己的格式化程序以满足特定需求（如数据加密、冗余消除等）。

问题：当客户端和服务器对象处在同一AppDomain或同一进程时，还要用到使用这些信道与格式化对象的体系结构吗？

回答：很明显，这两种情况没有必要使用信道，并且.NET Remoting能够自动识别出这两种情况，并采取相应的对策。对于在同一寻址空间完成调用这种情况，.NET Remoting会采用一个特殊的信道。

问题：当客户端代码与表示服务器对象的类不处在同一程序集时，它是如何知晓这些类的？

回答：在.NET Remoting下，至少有三个技术是可行的。客户端程序集在编译期间引用服务器程序集。这种情况下，服务器程序集就必须安装在客户端了。因此，该技术使得两者紧密耦合在一起。我们更倾向于第二个技术，它把接口封装到一个隔层程序集中，并同时部署在客户端和服务器上。第三个技术允许通过服务器程序集自动生成该隔层程序集，其中需要用到一个专门的工具，该工具随.NET Framework一起提供。

问题：与每次调用方法都使用网络相比，在客户端的AppDomain中获取服务器对象的副本然后使用该副本不是来得更简单吗？

回答：.NET Remoting提供了这种可能。该问题的答案是：视具体情况而定。这意味着包含服务器对象的类的程序集能被客户端所访问，而这往往不是我们所希望的。另外，一些服务器对象不能移动到另一AppDomain中。这通常是由于服务器对象引用了域中其他不能移动的对象。举个例子，Threat类的实例就是不能移动的。最后，当客户端创建对象副本时，我们也无法在多个客户端之间以并发的方式使用对象了。

问题：谁负责创建服务器对象？客户端还是服务器自己？

回答：在.NET Remoting下，两种情况都有可能。在服务器负责创建对象的情况下，对象通过URI来标识。客户端通过该URI与对象联系，就像你通过电话号码与朋友联系一样。在这种情况下，对象会在客户端第一次调用时创建。

问题：谁负责销毁服务器对象？客户端还是服务器自己？

回答：不管是否存在ping机制，DCOM认为在要销毁服务器对象时不要信任客户端。DCOM认为这样的ping机制会对性能造成影响。在.NET Remoting下，服务器对象会在没有被使用的一段时间之后自动销毁。客户端试图与不存在的对象联系时将会收到异常。

问题：.NET Remoting支持异步调用吗？

回答：支持。.NET Remoting支持5.12节所述的异步调用机制。在异步调用期间，你可以选择让服务器在你的调用执行完毕时向你发出通知。接着，可以以异步的方式获取服务器处理后的结果。

22.2　按引用封送

.NET Remoting提供两种体系架构不同的方案让客户端调用远程对象的方法。这两个方案是*按值封送*（MBV，Marshalling By Value）和*按引用封送*（MBR，Marshalling By Reference）。默认情况下，类的实例不支持远程调用。

按引用封送（MBR）是指在客户端的AppDomain中获取一个名为*透明代理*的新对象。对于客户端代码来说，透明代理就像一个普通的对象引用，透明这一概念也因此而来。事实上，即使csc.exe编译器也不知道某些引用在运行期间会是透明代理。

要想以透明代理的方式而不是普通引用的方式使用程序集中的类型，在编译程序集时必须提供该类型的元数据。往后会介绍几种能够满足这些条件的方法。此时，好奇的读者很可能会提出如下问题。

- 谁负责远程对象的创建？
- 如何获取透明代理？
- 代理类如何处理好在网络上传输输入和输出数据？

所有这些问题的答案都能在本章找到。下面这个程序的分析将为这些问题提供一些线索。下面看看远程对象的类和客户端在同一程序集中并且在同一AppDomain中的情况。

例22-1　MBRTest.cs

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Runtime.Remoting;
using System.Threading;

public class Foo : MarshalByRefObject {
   public void DisplayInfo(string s) {
      Console.WriteLine(s);
      Console.WriteLine("  Name of the domain: " +
                        AppDomain.CurrentDomain.FriendlyName);
      Console.WriteLine("  ThreadID      : " +
                        Thread.CurrentThread.ManagedThreadId);
   }
}

public class Program {
   static void Main() {
      // obj1
      Foo obj1 = new Foo();
      obj1.DisplayInfo("obj1:" );
      Console.WriteLine("  IsObjectOutOfAppDomain(obj1)=" +
         RemotingServices.IsObjectOutOfAppDomain( obj1 ) );
      Console.WriteLine("  IsTransparentProxy(obj1)=" +
         RemotingServices.IsTransparentProxy( obj1 ) );

      // obj2
      AppDomain appDomain=AppDomain.CreateDomain("Another AppDomain.");
      Foo obj2 = (Foo) appDomain.CreateInstanceAndUnwrap(
            "MBRTest",  // Name of the assembly that contains the type.
            "Foo");     // Name of the type.

      obj2.DisplayInfo("obj2:"); // <- Here, the client is not aware
                      // that he is working with a transparent proxy.
      Console.WriteLine("  IsObjectOutOfAppDomain(obj2)=" +
         RemotingServices.IsObjectOutOfAppDomain( obj2 ) );
      Console.WriteLine("  IsTransparentProxy(obj2)=" +
         RemotingServices.IsTransparentProxy( obj2 ) );
   }
}
```

程序输出：

```
obj1:
  Name of the domain: MBRTest.exe
  ThreadID      : 6116
  IsObjectOutOfAppDomain(obj1)=False
  IsTransparentProxy(obj1)=False
obj2:
  Name of the domain: Another AppDomain.
  ThreadID      : 6116
  IsObjectOutOfAppDomain(obj2)=True
  IsTransparentProxy(obj2)=True
```

留意对RemotingServices类的IsObjectOutOfAppDomain()和IsTransparentProxy()两个静态方法的使用。

图22-1展示了CLR运行该程序所用到的体系架构。上下文的概念将在本章稍后做出解释。

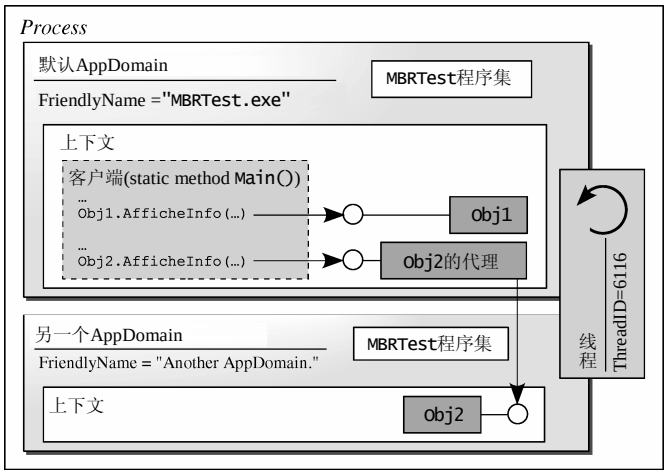

图22-1　按引用封送

你肯定留意到了上面这个程序中Foo类继承自System.MarshalByRefObject类。当JIT编译器遇到引用的类型是MarshalByRefObject的派生类时，它会认为这个引用最后将可能是个透明代理。这样做的结果就是，JIT编译器会阻止对这种引用上调用的方法进行内联。这种做法会将引用的对象设想成不是远程的，但这种假设通常是不成立的。换句话说，让类继承自MarshalByRefObject将告诉JIT，该类的实例可能会通过透明代理以远程方式使用。而在该实例及其客户端处在同一AppDomain中时，客户端将通过引用而不是透明代理使用该实例。

22.3　按值封送和二进制序列化

按值封送（MBV）是指在客户端的AppDomain中构建远程对象的副本。CLR使这个副本拥有和远程对象完全一样的状态。更确切的说，在某一特定时刻一个对象的状态是该对象所包含的值类型字段。视应用程序而定，对象的状态还包括对象中引用类型字段所引用的对象的状态。

副本并非远程对象。客户端无需使用透明代理来访问它。注意，副本不会调用构造函数。之所以这样很容易理解，因为副本必须与原对象完全一样，而原对象的构造函数又已经调用过了。

使用MBV的必要条件是客户端的AppDomain必须能够装载包含远程对象的类的程序集。甚至客户端和这个类可能在同一程序集中。另一个必要条件是原先的远程对象不能包含指向不支持克隆的对象的引用。实际上，对某些对象进行克隆是没意义的。举个例子说，为什么要在包含某个线程的进程外面克隆指向该线程的Thread类的实例呢？

CLR在内部用二进制数据流表示对象的状态，然后把该数据流发送到客户端的AppDomain。在客户端，CLR接收该数据流并在副本内部重新构建对象的状态。这些操作称作对象的*序列化*和*反序列化*。在.NET里，要使对象支持序列化，必须用System.Serializable Attribute标记它的类，或者让它的类实现System.Runtime.Serialization.ISerializable接口。在第一种情况中，会通知CLR使用默认的序列化机制。在第二种情况中，你可以实现你自己的序列化机制。默认情况下，所有基本类型都支持序列化。

一旦对象在客户端的AppDomain中克隆完毕，副本的状态和原先的远程对象的状态就没有关联了。对其中一方所施加的修改不会影响另一方。而MBValue这个术语的内涵也因此得以实现。实际上，这样的效果类似于向方法传递值类型参数。在该参数上所做的修改不会影响调用方的初始对象。图22-2演示了这点。

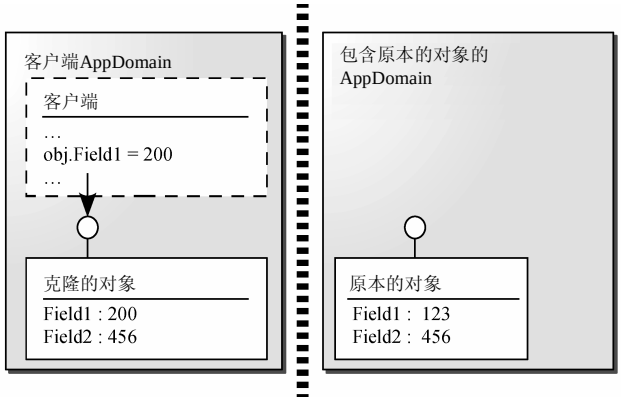

图22-2　按值封送

例22-1中让人感兴趣的是如何修改Foo类以使其支持MBV。

例22-2　MBVTest.cs

```
...
[Serializable]     // <- Added
public class Foo{  // : MarshalByRefObject  <- Commented
...
```

我们得到如下输出：

```
obj1:
  Nom of the domain: MBVTest.exe
  ThreadID      : 3620
  IsObjectOutOfAppDomain(obj1)=False
  IsTransparentProxy(obj1)=False
obj2:
  Nom of the domain: MBVTest.exe
  ThreadID      : 3620
  IsObjectOutOfAppDomain(obj2)=False
  IsTransparentProxy(obj2)=False
```

这证明了在客户端的AppDomain中实际上已经克隆出远程对象的一个新对象了。

专门讨论泛型的13.11.1节展示了二进制序列化懂得如何处理泛型类型。

版本容错序列化

在使用对象序列化时，我们注意到一个常见的问题是由要序列化的类的演化引起的，尤其是向其添加新的字段。实际上，在更新应用程序之后，当我们试图反序列化对象时，相关类的新字段的数据缺失将引发异常。要解决这个问题，可以在新字段上应用System.Runtime.Serialization.OptionalFieldAttribute。这样，相关字段将采用所属类型的默认值。

System.Runtime.Serialization命名空间还包含如下四个attribute，它们用于标记一个方法以便可以在序列化/反序列化过程中的某一步调用。可以使用下面这些方法为新字段赋值。

- OnDeserializingAttribute。该方法会在待反序列化对象的状态恢复和设置之前调用。
- OnDeserializedAttribute。该方法会在对象的状态反序列化和应用完毕之后调用。
- OnSerializingAttribute。该方法会在对象的状态序列化开始之前调用。
- OnSerializedAttribute。该方法会在对象的状态序列化完毕之后调用。

22.4 ObjectHandle类

在介绍MBR那节的代码中，我们可以用如下代码代替CreateInstanceAndUnwrap()方法的使用。

```
...
ObjectHandle hObj2 = appDomain.CreateInstance( "Remoting1" , "Foo" );
Foo obj2 = (Foo) hObj2.Unwrap();
...
```

System.Runtime.Remoting.ObjectHandle类的实例包含了使用远程对象的必要信息。通过调用UnWrap()方法，如果远程对象是MBR的话，可以获取它的透明代理，如果是MBV，则得到它的副本。

.NET把对远程对象的获取分为两步：获取ObjectHandle类的实例，然后解包该实例，这可以作为优化应用程序的一个根据。装载描述远程对象的类的类型元数据仅在第二阶段完成，即解包阶段。如果当前客户端并不使用该远程对象，而是仅仅把它传给应用程序的另一部分，那么就没有必要执行解包了。这样我们就免去了类型元数据的装载，也免去了程序集的装载。下面这个程序示范了这一点。注意，该程序利用了这样一个事实：CLR会在把相关类的类型元数据装载到AppDomain时调用类构造函数。

例22-3　WrapTest.cs

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

[Serializable]
public class Foo {
   // Class Constructor.
   static Foo() {
      Console.WriteLine( "Loading 'Foo class' metadata in the domain : " +
                         AppDomain.CurrentDomain.FriendlyName);
   }
   // Instance Constructor.
   public Foo() {
      Console.WriteLine( "'Foo' ctor called in the domain : " +
                         AppDomain.CurrentDomain.FriendlyName);
   }
}

public class Program {
   static void Main() {
      Console.WriteLine("-->About to call CreateDomain()");
      AppDomain appDomain = AppDomain.CreateDomain( "Another AppDomain." );
      Console.WriteLine("-->About to call CreateInstance()");
      ObjectHandle hObj = appDomain.CreateInstance( "WrapTest", "Foo" );
      Console.WriteLine("-->About to call UnWrap()");
      Foo obj = (Foo) hObj.Unwrap();
      Console.WriteLine("-->UnWrap() called");
   }
}
```

下面是程序的输出：

```
-->About to call CreateDomain()
-->About to call CreateInstance()
Loading 'Foo class' metadata in the domain : Another AppDomain
'Foo' ctor called in the domain : Another AppDomain
-->About to call UnWrap()
Loading 'Foo class' metadata in the domain : WrapTest.exe
-->UnWrap() called
```

这个程序还说明了在构建副本期间没有调用实例构造函数。当把Foo变成一个MBR类并运行这个程序时，倒数第二行将不显示。这清楚地说明了代理对象和远程对象并非同一类型。

22.5　对象的激活

如果读了前一节，那么就可以进入本话题的核心了，换句话说：在MBR的情况下，让对象调用穿越进程之间的虚拟边界和机器之间的物理边界，而在MBV的情况下则让对象直接穿越这些边界。

22.5.1　分布式体系的组件

在使用.NET Remoting的MBR服务器对象的分布式应用程序体系结构中，有四种重要的组件，如下所示。

- 调用服务器对象的客户端；
- 承载服务器对象的宿主；
- 服务器对象的类型元数据；
- 服务器对象的实现。

一般说来，我们把上述的每个组件独立到一个或多个程序集中。这种做法并不是强制的。实际上，可以把所有组件混合到一个单独的程序集中。但你会在接下来的这几页里找到这样做的好处。另外，我们会解释为何和如何从服务器对象的类型实现中分离出它们的类型元数据。图22-3展示了在标准的分布式应用程序中如何划分程序集。

图22-3　分布式体系结构的组件

22.5.2　宿主概览

宿主的职责有如下几点。

- 创建一个或多个信道；
- 向客户端暴露出可通过URI访问的类或服务器对象；
- 维护包含服务器对象的进程。

可能你会对宿主不但能暴露出对象还可以暴露出类感到惊讶。这是由于服务器对象可以由服务器或者客户端来激活的原因。对于第一种情况，客户端必须知晓那个对象以便使用它，而对于第二种情况，客户端必须知晓那个类以便激活它的实例。

22.5.3　信道概览

信道是一个用于在网络中协助通讯的对象，不过也有专门用于进程间通讯的信道。信道包含三个主要的参数，它们是所使用的网络端口、所使用的通讯协议（TCP、HTTP、IPC）和为了使得数据可以通过网络进行传输而对其进行格式化的方式（二进制格式、XML、SOAP标准……）。默认情况下，二进制格式将用于TCP和IPC协议，而SOAP格式将用于HTTP协议，但是更改这些配置也是很容易的。要使用.NET Remoting，需要两个使用相同通讯协议和相同数据格式化方式的信道：一个在客户端上，另一个在服务器端上。阅读本节后面的内容并不需要对信道有更多的了解，但我们将会在本章稍后专门用一节的内容来讨论信道这个话题。

22.5.4　同步方式、异步方式和单向方式调用

调用远程对象的方式可以是同步方式、异步方式或者没有返回值的异步方式（即单向方式）。值得注意的是，无论对象是否为远程对象，以异步方式调用它的技术都是一样的。该技术在5.12节有述。

22.5.5　对象激活与对象创建

需要提醒的是，在.NET Remoting中我们通常说*对象激活*而不是对象创建。这是由于在创建一个将以远程方式使用的对象的过程中，在发出创建请求到最后新的对象可用之间可能执行了大量的操作。这些操作就是我们所说的激活了。

22.6　Well-Known 对象的激活

设计分布式体系架构时的一个重要步骤是为每个远程对象指定是由客户端还是由服务器端（即宿主）来激活。.NET Remoting允许客户端在服务器上激活一个远程对象或者使用已存在于服务器上的对象。需要强调的是，当对象由服务器激活时，客户端无需调用这个对象的类的构造函数。这可能会成为选择服务器端激活还是客户端激活的决定性因素。下面我们将讨论对象由服务器激活的情况，而对象由客户端激活的情况将稍后讨论。

在客户端使用由服务器激活的远程对象的情况下，最好不要让客户端知道远程对象的实例所属的类。要做到这一点，最好把远程对象的类所支持的接口定义在一个专门的程序集中。纵观分布式体系架构的组件图（在图22-3中），这个程序集将充当服务器对象的类型元数据的角色。具体来说，这个程序集将同时出现在客户端和服务器的AppDomain中。举个例子，下面是这样一个程序集的代码：

例22-4　Interface.cs

```
namespace RemotingInterfaces {
   public interface IAdder {
      double Add(double d1, double d2);
   }
}
```

现在让我们来看看宿主和服务器对象的类。我们会把这两个实体放在同一个程序集中。然而，我们将会看到存在一种标准的宿主，要想使用它们，服务器对象的类必须分离到一个单独的程序集中。下面是包含服务器对象的实现和一个自定义宿主的程序集代码。留意宿主代码中这三项任务的顺序：创建信道、激活服务器对象和保留进程。

例22-5　Server.cs

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using RemotingInterfaces;

namespace RemotingServer {
   // Implementation of server objects.
   public class Adder : MarshalByRefObject, IAdder {
      public Adder() {
         Console.WriteLine( “Adder ctor” );
      }
      public double Add( double d1, double d2 ) {
         Console.WriteLine( “Adder Add( {0} + {1} )”, d1, d2 );
         return d1 + d2;
      }
```

```
    }

    // Implementation of the host.
    class Program {
       static void Main() {
          // 1) Creating a HTTP channel on the port 6510O.
          //    Register this channel in the current AppDomain.
          HttpChannel channel = new HttpChannel( 6510O );
          ChannelServices.RegisterChannel( channel, false );

          // 2) Register Well_Known Objects of type IAdder
          //    at the endpoint 'AddService'.
          RemotingConfiguration.RegisterWellKnownServiceType(
             typeof(Adder),
             "AddService",
             WellKnownObjectMode.SingleCall);

          // 3) Maintain the server process.
          Console.WriteLine( "Press a key to stop the server..." );
          Console.Read();
       }
    }
}
```

留意把“AddService”终结点关联到被激活的对象上的做法。通过组合协议、机器、端口以及终结点等信息，我们获得一个URI，通过它我们可以定位到由服务器激活的对象。在本例中，这个URI是http://localhost:65100/AddService。由于这个缘故，我们把well-known对象说成是由服务器激活并且有一个终结点的对象。从现在开始，我们将使用WKO（Well-Known对象）这个首字母缩略词来指代由服务器激活的对象。当我们深入讨论.NET Remoting的内部机制时，我们将会发现服务器可以不使用终结点而激活并发布一个对象。

接下来要处理的就是客户端程序集的代码了。需要做的就是创建一个信道然后获取与http://localhost:65100/AddService这个URI相关联的远程对象的透明代理。现在，我们可以使用这个通过非远程对象的引用获得的透明代理了。

例22-6　Client.cs

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

using RemotingInterfaces;

namespace RemotingClient {
   class Program {
      static void Main() {
         // Create an HTTP channel and register it in the current AppDomain
         // (the value O means that we delegate the choice of the port
         //  number to the CLR).
         HttpChannel channel = new HttpChannel( O );
         ChannelServices.RegisterChannel( channel, false );

         // Get a transparent proxy on a remote object from its URI.
         // Cast this transparent proxy to a reference of type IAdder.
         MarshalByRefObject objRef = (MarshalByRefObject)
                         RemotingServices.Connect(
```

```
                                typeof( IAdder ),
                                "http://localhost:65100/AddService" );
            IAdder obj = objRef as IAdder;

            // Invoke a method on the remote object.
            double d = obj.Add( 3.0, 4.0 );
            Console.WriteLine("Returned value:" + d);
        }
    }
}
```

现在我们有三个C#源代码文件了，通过下面三个命令行操作可以为它们各自生成相应的程序集。

```
>csc.exe /target:library Interface.cs
>csc.exe Server.cs /r:Interface.dll
>csc.exe Client.cs  /r:Interface.dll
```

我们也可以使用Visual Studio在同一个解决方案中分别创建三个项目。现在，只需依次启动Server.exe以及Client.exe。程序的输出如下所示。

Server.exe输出

```
Press a key to stop the server...
Adder ctor
Adder Add( 3 + 4 )
```

Client.exe输出

```
Returned value:7
```

WKO single call 激活方式与 WKO singleton 激活方式

在前一节的例子里，我们故意只调用一次服务器对象的方法。现在让我们来看看如果在客户端连续两次调用Add()将会输出什么。

Client.cs

```
...
   obj.Add( 3.0 , 4.0 );
   obj.Add( 5.0 , 6.0 );
...
```

服务器的输出：

```
Press a key to stop the server...
Adder ctor
Adder Add( 3 + 4 )
Adder ctor
Adder Add( 5 + 6 )
```

根据这个输出，服务器似乎为客户端的每次调用激活了一个对象，因为客户端的两次调用中每次都导致了构造函数的调用。实际情况也确实如此。服务器之所以采取这样的做法是由于当我们写下如下代码时就已经把我们的WKO对象声明为single call模式了。

```
...WellKnownObjectMode.SingleCall...
```

另一种服务器激活对象的调用模式叫做singleton。通过把SingleCall值替换为Singleton就可以选择这种调用模式了。singleton调用模式强制服务器在客户端进行第一次调用时激活对象，然后一直保留该对象，让它服务于来自所有客户端的所有后续调用。这样做的直接后果就是所有的客户端会共

享同一个对象。线程池的多个线程可能会同时执行这个对象的方法。于是就有必要提供同步机制来保证资源的并发访问得以顺利进行。如果我们用singleton调用模式声明我们的对象的话，那么服务器的输出如下所示。

```
Press a key to stop the server...
Adder ctor
Adder Add( 3 + 4 )
Adder Add( 5 + 6 )
```

注意，无论是以singleton模式还是single-call模式激活，直到客户端第一次调用服务器对象时，服务器才会激活对象。

我们更倾向于说这是一个由服务器提供的对象激活服务而不仅仅是由服务器激活一个对象。实际上，客户端并不真正拥有一个指向远程对象的引用，而是拥有一个指向远程对象激活服务的引用。具体说来，如果客户端所用的对象被销毁了，在下一次调用时，客户端将使用另一个由服务器自动激活的对象。

22.7　客户端激活的对象

我们将使用CAO这个首字母缩略词来指代客户端激活的对象。客户端要想激活对象必须知晓对象的类。这点是与由服务器激活对象所采用的体系架构相比较的一个基本差异。实际上，在WKO的情况里，客户端仅需知晓其想要在对象上使用的接口。而对于一个CAO对象，要使对象的类的类型元数据可用，就有必要提供包含对象的类的程序集。这种限制可能使你感到惊讶，不过接下来的章节会解释如何解决这个问题。现在，我们来编写包含Adder类的程序集的C#代码。

例22-7　ObjServer.cs

```
using System;

namespace RemotingObjServer {
   public interface IAdder {
      double Add(double d1, double d2);
   }
   // Implementation of server objects.
   public class Adder : MarshalByRefObject, IAdder {
      public Adder() {
         Console.WriteLine( “Adder ctor” );
      }
      public double Add( double d1, double d2 ) {
         Console.WriteLine( “Adder Add( {0} + {1} )”, d1, d2 );
         return d1 + d2;
      }
   }
}
```

现在，我们来看看包含宿主的程序集的源代码。这个宿主类似于我们用在服务器激活对象中的那个。唯一不同的是我们使用RegisterActivatedServiceType()静态方法表明我们期望工作在CAO模式而不是使用用于指定WKO模式的RegisterWellKnownServiceType()方法。

例22-8　Server.cs

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

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

using RemotingObjServer;

namespace RemotingServer {
   class Program {
      static void Main() {
         HttpChannel channel = new HttpChannel( 65100 );
         ChannelServices.RegisterChannel( channel, false );

         // Register the Adder class as a .NET Remoting CAO class
         // in the current AppDomain.
         RemotingConfiguration.RegisterActivatedServiceType(typeof(Adder));

         Console.WriteLine("Press a key to stop the server...");
         Console.Read();
      }
   }
}
```

客户端代码也与用在服务器激活对象中的那个非常相似。唯一不同的是我们使用Activator.CreateInstance()静态方法来获取我们所激活的对象的透明代理。

例22-9　Client.cs

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Activation;
using RemotingObjServer;

namespace RemotingClient {
   class Program {
      static void Main() {
         HttpChannel channel = new HttpChannel( 0 );
         ChannelServices.RegisterChannel( channel, false );

         // Activate a remote object and get a transparent proxy
         // that references it.
         IAdder obj = Activator.CreateInstance(
             typeof(Adder),
             null,
             new Object[] { new UrlAttribute("http://localhost:65100")})
            as IAdder;

         // Invoke a method on a remote object.
         double d = obj.Add( 3.0, 4.0 );
         Console.WriteLine( "Returned value:" + d );
      }
   }
}
```

现在我们有了三个C#源代码文件，通过下面三个命令行操作可以为每个文件生成对应的程序集。

```
>csc.exe /target:library ObjServer.cs
>csc.exe Server.cs /r:ObjServer.dll
>csc.exe Client.cs  /r:ObjServer.dll
```

现在，只需依次启动Server.exe和Client.exe。你将得到与使用服务器激活时一样的输出。这个示例并不足以展示客户端激活模式和服务器端激活模式的真正差异。然而，与由服务器在singleton模式下激活对象的情况相比，不同之处在于每个客户端将拥有各自不同的对象。与由服务器在single-call模式下激活对象的情况相比，不同之处则在于同一个对象会用于多个调用。

22.7.1　使用 new 关键字激活对象

之前的客户端存在着一个较大的缺点：代码不能使用C#语法的new关键字来激活远程对象。更糟糕的是，客户端不能选择Adder类的哪一个构造函数将为服务器所用。

RemotingConfiguration. RegisterActivatedClientType()静态方法可以解决上述两个问题。它可表示当前AppDomain中每个Adder的实例都将在服务器上激活。

例22-10　Client.cs

```
...
      static void Main() {
         HttpChannel channel = new HttpChannel(0);
         ChannelServices.RegisterChannel( channel, false );

         RemotingConfiguration.RegisterActivatedClientType(
            typeof(Adder),
            "http://localhost:65100");
         IAdder obj = (IAdder) new Adder();

         double d = obj.Add( 3.0, 4.0 );
         Console.WriteLine( "Returned value:" + d );
      }
...
```

22.7.2　潜在的问题

与COM/DCOM的做法相反，在.NET Remoting中客户端并不负责管理它所激活的对象的生存期，而这所导致的不良后果就是客户端可能希望使用它之前激活的一个远程对象，但是这个对象却已经不存在了。在这种情况下，客户端会引发System.Runtime.Remoting.RemotingException类型的异常。由于这个缘故，客户端必须时刻对此类异常有所防备。

服务器用于管理对象生存期的技术将在稍后介绍。

22.8　factory 设计模式和 soapsuds.exe 工具

在介绍客户端激活对象（CAO）的技术时，我们注意到客户端的程序集需要引用包含对象的类的程序集。在WKO的情况里这种做法是可以避免的，因为客户端只需知晓将与它交互的接口。然而，如果服务器对象的类并不支持接口，我们就不得不在客户端的程序集引用中包含这个类的程序集。

一般来说，把包含服务器对象的实现的程序集部署到客户端是不可接受的。因为这个类的实现通常要尽可能的保密。在这个程序集中，客户端仅需服务器对象所在类的类型元数据。解决这个问题有两个方法，factory设计模式（Gof）或者Soapsuds.exe工具。

22.8.1　factory 设计模式

隐藏在factory设计模式背后的思想是通过WKO对象来完成CAO对象的构造，这样宿主就只需将WKO服务暴露出来。正如我们下面看到的，此时客户端也只需要知晓接口，这样问题也就解决了。

让我们修改上面的代码从而在服务器上激活对象。首先，我们向客户端引入一个新的IFactory接口。

例22-11　Interface.cs

```
namespace RemotingInterfaces {
   public interface IAdder {
      double Add( double d1, double d2 );
   }
   public interface IFactory {
      IAdder BuildANewAdder();
   }
}
```

接着，必须提供实现IFactory接口的Factory类，并把Factory对象以singleton模式暴露出来（在这种情况下最好不要使用single-call模式）。

例22-12　Server.cs

```
...
   public class Adder : MarshalByRefObject, IAdder {
      public Adder() {
         Console.WriteLine( "Adder ctor" );
      }
      public double Add( double d1, double d2 ) {
         Console.WriteLine( "Adder Add( {0} + {1} )", d1, d2 );
         return d1 + d2;
      }
   }
   public class Factory : MarshalByRefObject, IFactory {
      public IAdder BuildANewAdder() {
         return (IAdder)new Adder();
      }
   }
   class Program {
      static void Main() {
         HttpChannel channel = new HttpChannel( 65100 );
         ChannelServices.RegisterChannel( channel, false );
         RemotingConfiguration.RegisterWellKnownServiceType(
               typeof( Factory ),
               "FactoryService",
               WellKnownObjectMode.Singleton );
         Console.WriteLine( "Press a key to stop the server..." );
         Console.Read();
      }
   }
...
```

最后，我们能够在不知晓Adder和Factory类的情况下通过客户端激活Adder的实例，这正是这个方法所要实现的目标。

例22-13　Client.cs

```
...
   class Program {
      static void Main() {
         HttpChannel channel = new HttpChannel( 0 );
         ChannelServices.RegisterChannel( channel, false );

         MarshalByRefObject tmpObj = (MarshalByRefObject)
                     RemotingServices.Connect(
                           typeof( IFactory ),
                           "http://localhost:65100/FactoryService" );
```

```
            IFactory factory = tmpObj as IFactory;
            IAdder obj = factory.BuildANewAdder();

            double d = obj.Add( 3.0, 4.0 );
            Console.WriteLine( "Returned value:" + d );
         }
      }
   ...
```

这个设计模式之所以能在这里发挥作用乃是因为Adder和Factory类都继承自MarshalByRefObject类。

22.8.2　soapsuds.exe 工具

.NET Framework提供的Soapsuds.exe工具能够从服务器对象所在类的程序集中提取该类的元数据。Soapsuds.exe既能从元数据中构造出可用于编译的C#文件，也可以构造出仅包含类型元数据的程序集。下面这个命令行用于从名为ObjServer.dll的程序集中创建一个名为ObjectServerMetadataForClient.dll的程序集。

```
>soapsuds /ia:ObjServer /oa:ObjectServerMetadataForClient.dll
```

下面这个命令行用于从这个ObjServer.dll程序集创建名为ObjServer.cs的C#源代码文件。

```
>soapsuds /ia:ObjServer /gc
```

以下是来自ObjServer.cs C#源文件中的一个片段。

ObjServer.cs

```
...
public class Adder : System.Runtime.Remoting.Services.RemotingClientProxy,
                     IAdder {
   // Constructor
   public Adder() { }
   public Object RemotingReference { get { return (_tp); } }
   [SoapMethod(SoapAction = @"http://schemas.microsoft.com/clr/nsassem/
                            RemotingInterfaces.Adder/Interface#Add")]
   public virtual double Add( double d1, double d2 ) {
      return ((Adder)_tp).Add( d1, d2 );
   }
}
...
```

实际上，我们有了一个新的Adder类，它拥有和原来那个一样的公共方法并继承自System.Runtime.Remoting.Services.RemotingClientProxy类。我们建议你查阅MSDN中与这个类相关的文档，因为可以使用它来获得客户端验证机制或者当客户端处在防火墙背后时用它来指定使用哪个代理服务器。不过，也可以通过指定/nowp选项来告知Soapsuds.exe你不希望让类继承自RemotingClientProxy。在这种情况下，它们会直接继承于MarshalByRefObject。

当远程服务器通过HTTP信道暴露服务器的类的实例时，Soapsuds.exe工具也允许你从服务器那里构建包含这些类的元数据的程序集。下面是服务器的代码。

```
...
class Program {
   static void Main() {
      HttpChannel channel = new HttpChannel( 65100 );
      ChannelServices.RegisterChannel( channel, false );
      RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(Adder),
```

```
            "MyDir/AddService.soap",
            WellKnownObjectMode.SingleCall);
        ...
    }
}
...
```

下面是在客户端使用的命令行：

```
>Soapsuds /url:http://localhost:65100/MyDir/AddService.soap?wsdl
 /oa:ObjectServerMetadataForClient.dll
```

注意把soap扩展名关联到终结点上，并在URL中使用?wsdl后缀，这将告知服务器我们希望获得什么（即获得类型元数据）。

最后需要提醒的是soapsuds.exe工具并不考虑带参数的构造函数。因此，对于必须使用带参数的构造函数初始化对象的情况，使用factory设计模式会更合适。

22.9　Well-Known 和客户端激活的对象的生命周期

谈论single-call模式下WKO的生存期是没有意义的，因为对象在第一次调用后马上就销毁了。我们将介绍.NET Remoting在决定由远程实体激活的对象何时销毁时所采用的解决方案。该解决方案将建立在租约机制的基础上。有一点需要注意，该解决方案同时考虑了CAO和singleton模式下的WKO。同样需要注意的是，我们在这里介绍的仅适用于继承自MarshalByRefObject类的对象并且它们是被所在的AppDomain以外的实体访问的。记住，不符合这些条件的对象的生存期将由垃圾收集器管理。

在AppDomain中并不存在一个强引用指向那些可供位于AppDomain以外的实体访问的对象。为了防止垃圾收集器回收这样一个对象，CLR让每个AppDomain包含一个租约管理器。租约管理器在每个以远程方式使用的MBR对象被激活时为其分配生存期。租约管理器会周期性地检查这些对象中每个对象的租约。租约过期的对象会在下次垃圾回收时自动销毁。

不过，.NET Remoting赋予了这个租约机制更多的灵活性。具体来说，对象的租约期可以通过以下三种方式延长。

- 租约期会在每次调用对象时自动延长。
- 租约期可以通过直接调用某个方法来延长。
- 最后，当租约管理器发现某个租约过期了，它会在决定销毁相应的对象之前依次询问它的主办方，而主办方具有延长将被销毁的对象的租约的能力。主办方的类必须继承自MarshalByRefObject，因此它也可以是一个远程对象。如果对某个主办方的询问超过一定时间，租约管理器将停止向它询问并继续询问下一个主办方。

System.Runtime.Remoting. Lifetime命名空间包含名为ILease和ISponsor的两个接口，它们是专门为管理我们刚才所提到的机制而设计的：

```
interface System.Runtime.Remoting.Lifetime.ILease{
   // The LeaseState enumeration has values :
   //  Null      Not intialized.
   //  Initial   Currently initialized.
   //  Active    Initialized and valid.
   //  Renewing  Not valid anymore and sponsors are currently consulted.
   //  Expired   Expired.
   LeaseState   CurrentState{ get; }

   // Initial lease duration.
   // This property can be set only during lease initialization.
```

```
    TimeSpan   InitialLeaseTime{ get; set; }

    // The duration by witch a call to a remote object renews
    // the current lease time.
    TimeSpan   RenewOnCallTime{ get; set; }

    TimeSpan   CurrentLeaseTime{ get; }

    // Renew the current lease time.
    TimeSpan   Renew( TimeSpan );

    // Methods to manage sponsors of the object.
    void       Register( ISponsor );
    void       Register( ISponsor, TimeSpan );
    void       UnRegister( ISponsor) ;

    // Maximum amount of time to wait for a sponsor to return
    // its renewal time.
    TimeSpan   SponsorshipTimeout{ get; set; }
}

interface System.Runtime.Remoting.Lifetime.ISponsor{
    Timespan Renewal(ILease);
}
```

正如你所看到的，这些接口的使用是直观的。你可以通过在对象上调用object MarshalByRefObject.GetLifetimeService()方法或者通过调用object RemotingServices.GetLifetimeService(object)静态方法获取对象的租约。举个例子：

```
...
ILease lease = (ILease) obj.GetLifetimeService();
lease.Renew( TimeSpan.FromSeconds(30) );
...
```

租约是由mscorlib.dll中一个名为Lease的内部类实现的，该类无法访问。

对象租约的三个参数是租约的初始时间、当调用对象的某个方法时租约的延长时间和询问主办方的最长等待时间。对于一个给定的对象，这三个参数会在机器配置文件里获得它们的默认值。默认时间分别为5分钟、2分钟和2分钟。System.Runtime.Remoting.LifetimeServices类的静态属性可用来在当前AppDomain中设置这些默认值。而要在类的实例上设置这些值，必须重写object MarshalByRefObject.GetLifetimeService()虚方法。例如：

例22-14

```
...
using System.Runtime.Remoting.Lifetime;
...
   public class Adder : MarshalByRefObject, IAdder {
      public override object InitializeLifetimeService() {
         ILease lease = (ILease) base.InitializeLifetimeService();
         if ( lease.CurrentState == LeaseState.Initial ) {
            lease.InitialLeaseTime   = TimeSpan.FromSeconds( 50 );
            lease.RenewOnCallTime    = TimeSpan.FromSeconds( 20 );
            lease.SponsorshipTimeout = TimeSpan.FromSeconds( 20 );
         }
         return lease;
      }
   ...
}
...
```

通过把TimeSpan.Zero值赋给租约的InitialLeaseTime属性，你可以无限延长租约期。

可以定义你自己的主办方类。需要提醒的是，这样的类必须继承自MarshalByRefObject并实现ISponsor接口。主办方类在指定分布式体系架构的期间能够发挥其作用，此时对象的生存期在逻辑上依赖于某种情况。最常见的情况是某些客户端一直处于活动状态。这样，只需为每个客户端准备一个主办方，只要关联到主办方的客户端仍然需要该对象就为它延长租约期。不过，需要小心这类ping机制。DCOM技术就采用了这种机制，但实践证明一旦应用程序的客户端数目超出极限，该机制的效率就无法接受了。

22.10　配置.NET Remoting

用来展示远程对象激活机制的示例存在一个主要的问题：它们是不可配置的。换句话说，一旦编译好了，我们就不能改变端口号或者服务器的名称。下面，我们将演示如何令服务器以及客户端上的配置参数能够被我们读取并修改。你可能已经猜到这些参数会在一份XML文档中制定。为了演示这些配置功能，我们将使用如下所示的定义在一个新的程序集中的三个类。

例22-15　ObjServer.cs

```
using System;

namespace RemotingObjServer {
   public interface IAdder {
      double Add(double d1, double d2);
   }
   public class Adder : MarshalByRefObject, IAdder {
      public Adder() {
         Console.WriteLine( “Adder ctor” );
      }
      public double Add( double d1, double d2 ) {
         Console.WriteLine(“Adder Add( {0} + {1} )”, d1, d2 );
         return d1 + d2;
      }
   }
   public interface IMultiplier {
      double Mult( double d1, double d2 );
   }
   public class Multiplier : MarshalByRefObject, IMultiplier {
      public Multiplier() {
         Console.WriteLine( “Multiplier ctor” );
      }
      public double Mult( double d1, double d2 ) {
         Console.WriteLine( “Multiplier Mult( {0} + {1} )”, d1, d2 );
         return d1 * d2;
      }
   }
   public interface IDivider {
      double Div( double d1, double d2 );
   }
   public class Divider : MarshalByRefObject, IDivider {
      public Divider() {
         Console.WriteLine( “Divider ctor” );
      }
      public double Div( double d1, double d2 ) {
         Console.WriteLine( “Divider Div( {0} + {1} )”, d1, d2 );
         return d1 + d2;
      }
   }
}
```

22.10.1　配置宿主

以下的XML文档允许你以WKO的singleton模式暴露Adder类并以WKO的single-call模式暴露Multiplier类。与此同时，Divider类也暴露出来以便让远程对象激活。这三个对象都可以通过HTTP信道的65100端口访问。如果希望使用TCP或者IPC信道，可以把ref属性的值设置为tcp或者ipc。对于使用IPC的情况，我们无需指定端口号。

每个暴露出来的类型都必须在前面加上命名空间。另外，类的名字后面也必须紧跟包含它的程序集的名字。如果这个程序集带有强名称，则必须使用完整的强名称。

例22-16　Host.config

```
<configuration>
   <system.runtime.remoting>
      <application name = “Server”>
         <service>
            <wellknown type=”RemotingObjServer.Adder,ObjServer”
                       mode =”Singleton” objectUri=”Service1.rem” />
            <wellknown type=”RemotingObjServer.Multiplier,ObjServer”

                       mode =”SingleCall” objectUri=”Service2.rem” />
            <activated type=”RemotingObjServer.Divider,ObjServer” />
         </service>
         <channels>
            <channel port=”65100” ref=”http” />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

由于void RemotingConfiguration.Configure(string)静态方法接受XML文档的名字作为参数，所以宿主的代码量得以减至最少。

例22-17　Server.cs

```
...
   class Program {
      static void Main() {
         RemotingConfiguration.Configure( “Host.config”, false );
         Console.WriteLine( “Press a key to stop the server...” );
         Console.Read();
      }
   }
...
```

22.10.2　配置客户端

下面的XML文档使得客户端可以使用由先前的服务器暴露的两个WKO服务和一个类：

例22-18　Client.config

```
<configuration>
   <system.runtime.remoting>
      <application name = “Client”>
         <client>
            <wellknown type=”RemotingObjServer.Adder,ObjServer”
                       url=”http://localhost:65100/Service1.rem” />
            <wellknown type=”RemotingObjServer.Multiplier,ObjServer”
                       url=”http://localhost:65100/Service2.rem” />
         </client>
```

```
            <client url="http://localhost:65100/">
               <activated type ="RemotingObjServer.Divider,ObjServer"/>
            </client>
         </application>
      </system.runtime.remoting>
   </configuration>
```

指定的URL以http://访问模式开始，但如果接收信道是TCP或者IPC的话可以把它们改为以tcp://或者ipc://开始。在使用IPC信道的情况中，可以用底层命名管道的名称替代机器名称的宿主/端口部分（在这里是localhost:65100）。这个名称可通过服务器配置文件中信道声明的portName属性提供。

与宿主类似，我们也是通过void RemotingConfiguration.Configure(string)静态方法把参数从这个XML文件装载到客户端的AppDomain中的。

例22-19

```
...
using RemotingObjServer;
...
   class Program {
      static void Main() {
         RemotingConfiguration.Configure("Client.config", false);

         Adder objA = new Adder();
         double dA = objA.Add( 3.0, 4.0 );

         Multiplier objM = new Multiplier();
         double dM = objM.Mult( 3.0, 4.0 );

         Divider objD = new Divider();
         double dD = objD.Div( 3.0, 4.0 );
      }
   }
...
```

调用WKO对象的构造函数仅仅为了获取指向对应类型的引用（实际上是一个透明代理）。具体地说，它们没有导致任何网络访问。

也可以在服务器配置文件中配置租约管理的参数。关于这个话题的更多内容可以在MSDN上名为“<lifetime> Element”的文章中找到。

最初因为我们要调用构造函数，所以也就无法使用接口。factory设计模式也无法使用，因为即使是服务器激活的情况，客户端仅使用接口也是无法完成激活的。这样，就必须使用Soapsuds.exe工具来避免向客户端提供包含服务器对象的实现的程序集。然而，我们将介绍一个诀窍使得可以在WKO模式中使用接口，从而可以在CAO模式中使用factory设计模式。

22.10.3 联合使用接口和配置文件

首先，让我们把接口放在一个程序集中，这个程序集将同时被客户端和服务器端引用。

例22-20 Interface.cs

```
namespace RemotingInterfaces {
   public interface IAdder {
      double Add( double d1, double d2 );
   }
   public interface IMultiplier {
      double Mult( double d1, double d2 );
   }
```

```
   public interface IDivider {
      double Div( double d1, double d2 );
   }
   public interface IFactory {
      IAdder       BuildNewAdder();
      IMultiplier  BuildNewMultiplier();
      IDivider     BuildNewDivider();
   }
}
```

接着，这个诀窍在于一张表的使用，这个表把服务器配置的WKO服务关联到配置在客户端上的接口。下面是客户端的代码，它管理着名为dicoTypes的表。

例22-21　Client.cs

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Activation;
using System.Collections;

using RemotingInterfaces;

namespace RemotingClient {
   class CustomActivator {
      private static bool bInit;
      // Associating interfaces with services distants WKO.
      private static IDictionary dicoTypes;

      public static Object GetObject( Type type ) {
         if ( !bInit )
            InitdicoTypes();
         WellKnownClientTypeEntry entry = (WellKnownClientTypeEntry)
                                          dicoTypes[type];
         return Activator.GetObject( entry.ObjectType, entry.ObjectUrl );
      }

      private static void InitdicoTypes() {
         bInit = true;
         dicoTypes = new Hashtable();
         foreach ( WellKnownClientTypeEntry entry in
            RemotingConfiguration.GetRegisteredWellKnownClientTypes() )
               dicoTypes.Add( entry.ObjectType, entry );
      }
   }

   class Program {
      static void Main() {
         RemotingConfiguration.Configure( “Client.config”, false );

         IAdder objA = (IAdder)CustomActivator.GetObject( typeof(IAdder) );
         double dA = objA.Add( 3.0, 4.0 );

         IMultiplier objM = (IMultiplier)
            CustomActivator.GetObject( typeof( IMultiplier ) );
         double dM = objM.Mult( 3.0, 4.0 );
```

```
            // Factory design pattern for a Client Activated Object.
            IFactory factory = (IFactory)
               CustomActivator.GetObject( typeof( IFactory ) );
            IDivider objD = factory.BuildNewDivider();
            double dD = objD.Div( 3.0, 4.0 );
         }
      }
   }
```

客户端的配置文件如下所示。

例22-22　Client.config

```
<configuration>
   <system.runtime.remoting>
      <application name = "Client">
         <client>
            <wellknown type="RemotingInterfaces.IAdder,Interface"
                       url="http://localhost:65100/Service1.rem" />
            <wellknown type="RemotingInterfaces.IMultiplier,Interface"
                       url="http://localhost:65100/Service2.rem" />
            <wellknown type="RemotingInterfaces.IFactory,Interface"
                       url="http://localhost:65100/Service3.rem" />
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

注意，接口/WKO服务关联表是通过这个配置文件明确地描述出来的。这些内容可以通过RemotingConfiguration.GetRegisteredWellKnownClientTypes()方法在客户端读取。

这个服务器配置文件展示了三个WKO服务。需要提醒的是，第三个服务使得客户端可以通过factory设计模式使用CAO对象。

例22-23　Host.config

```
<configuration>
   <system.runtime.remoting>
      <application name = "Server">
         <service>
            <wellknown type="RemotingServer.Adder,Server"
                       mode ="Singleton" objectUri="Service1.rem" />
            <wellknown type="RemotingServer.Multiplier,Server"

                       mode ="SingleCall" objectUri="Service2.rem" />
            <wellknown type="RemotingServer.Factory,Server"
                       mode ="SingleCall" objectUri="Service3.rem" />
         </service>
         <channels>
            <channel port="65100" ref ="http" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

当然，现在可以把Adder、Multiplier、Factory和Divider类放在服务器的代码中了。

例22-24　Server.cs

```
using System;
```

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

using RemotingInterfaces;

namespace RemotingServer{
   public class Adder : MarshalByRefObject, IAdder { ... }
   public class Multiplier : MarshalByRefObject, IMultiplier { ... }
   public class Divider : MarshalByRefObject, IDivider { ... }
   public class Factory : MarshalByRefObject, IFactory {
      public IAdder BuildNewAdder() { return new Adder(); }
      public IMultiplier BuildNewMultiplier() { return new Multiplier(); }
      public IDivider BuildNewDivider() { return new Divider(); }
   }
   class Program {
      static void Main() {
         HttpChannel channel = new HttpChannel( 65100 );
         ChannelServices.RegisterChannel( channel, false );
         RemotingConfiguration.RegisterWellKnownServiceType(
               typeof(Factory),
               "FactoryService",
               WellKnownObjectMode.Singleton );
         Console.WriteLine( "Press a key to stop the server..." );
         Console.Read();
      }
   }
}
```

22.11　.NET Remoting 服务器的部署

在部署基于.NET Remoting的分布式应用程序时，最好为每个组件提供至少一个配置文件。这样就可以无需重新编译任何程序集来配置应用程序了。

服务器对象的类的版本问题也可以通过这些配置文件来解决。事实上，当在XML属性<type>中指定类型时，可以给出程序集的强名称。而强名称就包含了程序集的版本号。

本章中的所有宿主都运行在控制台模式。在真实的世界里，我们倾向于使用Windows服务或者ASP.NET来承载我们的服务器对象。

22.11.1　Windows 服务

与控制台应用程序相比，Windows服务的主要优点在于用户无需登录机器即可运行由Windows服务承载的应用程序。另一个优点是对正在运行的服务的操作可以通过UI所提供的*开始/停止/暂停/恢复/重启*等命令来完成。还有一个能够说明该方案好的理由是服务可以在机器*重启*后自动运行。我们不会在本书详述如何创建服务。与使用专门的Win32函数相比，使用包含在System.ServiceProcess命名空间里的类，会使这项工作变得异常简单。所有这些都在MSDN的“System.ServiceProcess Namespace”一文中有述。

22.11.2　IIS

IIS是除控制台模式与Windows服务之外承载服务器对象的另一个选择，它具有以下优点。

- 可以使用IIS的安全管理。另外，如果你的IIS服务器支持SSL证书，则可以使用数据加密服务。也可以使用Windows验证机制。
- IIS的使用减轻了编写宿主的负担。具体地说，只需开发继承自MarshalByRefObject的类并提供配置文件即可。

- IIS也考虑了版本问题。它能检测到新版本的安装并负责新版本的过渡。安装新版本仅需替换新的程序集和配置文件而无需停止IIS。当配置文件正被使用时，IIS不会阻止对它们进行的写操作，因为IIS在内部使用的是这些文件的影子副本。

然而，所有这些操作都是有代价的，而IIS的主要问题在于它会对性能产生显著的影响。另外，你无法在IIS上使用TCP或者IPC信道。使用IIS要遵循如下两步。

(1) 在mmc（Windows的通用管理控制台）下的IIS管理单元中创建一个新的IIS虚拟目录。需要说明的是，IIS虚拟目录使得机器上的文件夹可以通过URI来访问。必须在这个文件夹中放置你的配置文件，并且该文件的名称必须是Web.Config。

(2) 在第一步所创建的文件夹里创建一个新的名为bin的子文件夹。把包含你的服务器对象所在类的程序集放在这个文件夹里。另一个选择是把你的程序集放在GAC文件夹里。为此，你的程序集必须具有强名称。

当使用这种部署方式时配置文件中的<application>元素不能有Name属性。此外，也不能在配置文件中指定端口以免影响了IIS的端口管理。

22.12　安全的.NET Remoting 信道

22.12.1　安全的 TCP 信道

如果使用TCP信道，那么可以使用NTLM和Kerberos协议来验证运行客户端的Windows用户的身份并加密客户端与服务器端之间所交换的数据。TCP信道内部使用NegociatedStream类来提供这个功能，该类使用了与这些协议相关的被称为SSPI的Win32 API服务。这些协议以及该类在17.10节有述。

要使用这个功能，只需在客户端和服务器端的配置文件中把TCP信道的secure属性设为true就可以了。

```
<configuration>
   <system.runtime.remoting>
      <application name = "XXX">
         <channels>
            <channel port="65100" ref="tcp" secure="true"/>
         </channels>
...
```

也可以通过信道的属性列表以编程的方式把secure属性的值设为true。

例22-25　Client.cs

```
...
      static void Main() {
         IDictionary properties = new Hashtable();
         properties.Add( "secure", true );
         TcpChannel channel = new TcpChannel( properties,null,null );
         ChannelServices.RegisterChannel( channel, true );
...
```

如果希望客户端的请求在服务器上执行时，仍旧使用客户端的用户账号而非服务器账号，那么也可以在服务器端上把TCP信道的tokenImpersonationLevel属性的值设为Impersonation。

22.12.2　安全的 HTTP 信道

如果希望在使用HTTP协议时保护被交换的数据，有两个方案可用。

- 用IIS承载服务器并在IIS层面使用SSL协议（即HTTPS）。
- 创建你自己的HTTP安全信道。

22.13　代理和消息

本节将深入分析透明代理并介绍真实代理和消息拦截器的概念。下面简单回顾一下透明代理。

- 透明代理是在mscorlib.dll程序集中定义的一个内部类的实例。因此，该类不能直接访问。
- 透明代理是由CLR自动创建用于引用远程对象的，这个远程对象的类必须继承自MarshalByRefObject。
- 在程序集中，对象引用的内部管理会因程序集是否装载到对象所在的AppDomain中而有所不同。当程序集和对象不在同一AppDomain时，我们引用一个透明代理。当它们在同一AppDomain时，我们直接引用对象。我们已经知道，bool RemotingServices.IsTransparentProxy(object)静态方法可用于判断对象的引用是透明代理还是直接的引用。
- 当CLR构建远程对象的透明代理时，用于操作远程对象的类型元数据必须装载到AppDomain中。我们已经了解了获取这些元数据的多种方法。

22.13.1　把方法调用转换成消息

正如图22-4所示，标准的方法调用可以被视为触发调用的线程的栈的变换。

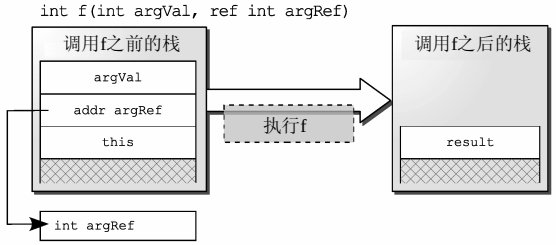

图22-4　方法调用=栈变换

这个变换分成两个步骤进行：在方法执行之前，CLR在栈中取出参数以便用作方法的局部变量，而在执行之后，CLR把返回值压入栈中。

在调用位于另一个AppDomain（甚至另一个进程）中的远程对象的某个方法期间，不可能使用这个基于参数传递技术的栈。事实上，调用方法的线程可能与执行它的线程不同。参数的传递使用了消息交换技术，一是用在调用方法期间（包含方法所需的信息），二是用在方法执行结束时（包含方法所产生的信息）。

透明代理扮演的角色是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，而CLR内部的一个名为栈生成器接收器则负责在服务器端完成与之相反的操作。图22-5中展示了这点。

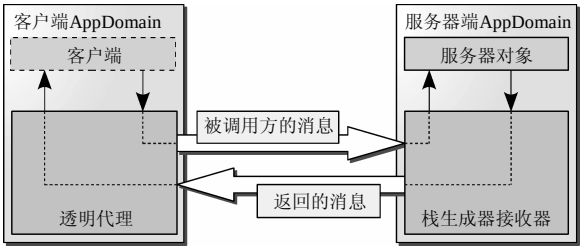

图22-5　透明代理和栈生成器接收器

22.13.2　IMessage接口的层次结构

下一节我们将会看到可以在不同的消息处理层上截获透明代理和栈生成器接收器之间交换的消息。我们将会看到这些消息事实上是.NET对象。现在让我们把注意力集中在用于操作这些消息的接口上。下面是它们的层次结构。

```
System.Runtime.Remoting.Messaging.IMessage
   System.Runtime.Remoting.Messaging.IMethodMessage
      System.Runtime.Remoting.Messaging.IMethodCallMessage
         System.Runtime.Remoting.Activation.IConstructionCallMessage
      System.Runtime.Remoting.Messaging.IMethodReturnMessage
         System.Runtime.Remoting.Activation.IConstructionReturnMessage
```

下面是IMessage接口和IMethodMessage接口的定义。

```
using System.Runtime.Remoting.Messaging;
using System.Collections;
public interface IMessage {
   IDictionary Properties { get; }
}
public interface IMethodMessage : IMessage {
   object               GetArg( int index );
   string               GetArgName( int index );
   int                  ArgCount { get; }
   object[]             Args { get; }
   bool                 HasVarArgs { get; }
   string               MethodName { get; }
   object               MethodSignature { get; }
   string               TypeName { get; }
   string               Uri { get; }
   LogicalCallContext   LogicalCallContext { get; }
   MethodBase           MethodBase { get; }
}
```

IMethodCallMessage接口实质上用于遍历由透明代理所构建的消息中的数据。IMethodReturnMessage接口则用于遍历由栈生成器接收器所构建的消息中的数据。

22.13.3　透明代理、真实代理和ObjRef类

为了不让事情复杂化，一直以来我们都隐藏了真实代理的存在。真实代理是System.Runtime.Remoting.Proxies.RealProxy类（或其派生类）的实例对象。每个透明代理有且仅有一个与之对应的真实代理对象。透明代理和真实代理在调用远程对象期间是一起工作的。正如我们之前看到的，透明代理负责把基于栈的参数传递模式转换成基于消息的参数传递模式。真实代理的一个任务就是寻找用于发送由透明代理创建的消息的合适信道，然后发送消息，而另一个任务则是从信道中获取表示方法调用所产生的返回值的消息，然后把这个返回消息转送给透明代理。与透明代理不同，你可以创建自己的真实代理类，这将在稍后介绍。

1．类型和真实代理

真实代理的另一个责任是获知远程对象的类型。真实代理的构造函数接收一个类型作为参数。该类型必须是一个接口或者继承自MarshalByRefObject的类；否则真实代理的构造函数会抛出异常。当用户对他认为是普通引用但事实上是一个透明代理的东西进行转换时，CLR会自动在真实代理上调用Type GetProxiedType()方法。

2．创建一对透明代理/真实代理和ObjRef类

在一对透明代理/真实代理中，通常会首先创建真实代理，然后CLR会通过System.Runtime.Remoting.ObjRef类的实例或者远程对象的定位信息创建透明代理。这些信息可能包含对象激活服务

的URI。ObjRef类的实例是MBV的，因此可以在AppDomain边界之间交换信息。ObjRef的实例包含真实代理引用远程对象所需的信息。这些信息是：远程对象的类型、远程对象的定位信息、用于联系远程对象的信道类型和一些CLR内部使用的信息。引用远程对象的ObjRef类的实例可以从多个途径获取。

在对象由客户端激活的情况中，客户端会在激活对象时获取ObjRef类的实例。对象的激活可以通过使用new关键字、AppDomain.CreateInstance()方法或者Activator.CreateInstance()方法实现。

在WKO对象的情况中，客户端并不需要服务器初始化的ObjRef实例。这是合理的，因为客户端仅需知道激活服务的URI就可以访问该服务了。这样带来的结果就是，调用Activator.GetObject()或者RemotingServices.Connect()方法不会向网络发送消息，而这可能与我们原先想象的不同。除了优化网络的使用，这样做还可以使得服务器在客户端调用之前无需创建对象，并且该机制对于以singleton模式或以single-call模式激活的对象都适用。与客户端对象激活机制相比，该机制可以减少网络上的一个消息来回。

最后需要知道的是，服务器AppDomain会为每个引用类型的返回值和每个引用类型的返回参数返回一个ObjRef类的实例。当然，这些参数的引用类型必须是接口或者MarshalByRefObject的派生类。

22.13.4　通过 ObjRef 类发布对象

ObjRef类的实例包含了用于定位和标识远程对象的所有必要信息。我们可以由此推断出，把这样一个实例序列化到文件中，我们就有了一个到对象的引用。然后把这个文件传给客户端，例如通过电子邮件。通过反序列化ObjRef类的实例，客户端就获得了一个指向远程对象的引用并可以使用它了。这种做法叫作*发布对象*。由于.NET Remoting提供的一些方法使得发布对象这项工作变得容易。

下面的代码展示了服务器把对象的引用序列化到一个名为Adder.txt的文件中。注意，对象的类必须继承自MarshalByRefObject：

例22-26　Server.cs

```
...
      static void Main() {
         HttpChannel channel = new HttpChannel( 65100 );
         ChannelServices.RegisterChannel( channel, false );

         CAdditionneur obj = new CAdditionneur();
         ObjRef objRef = RemotingServices.Marshal( obj );
         FileStream fStream = new FileStream("Adder.txt", FileMode.Create);
         SoapFormatter soapFormatter = new SoapFormatter();
         soapFormatter.Serialize( fStream, objRef );
         fStream.Close();

         Console.WriteLine( "Press a key to stop the server..." );
         Console.Read();
      }
...
```

下面的代码展示了客户端从*Adder.txt*文件中反序列化远程对象的引用。

例22-27　Client.cs

```
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;
...
      static void Main() {
         HttpChannel channel = new HttpChannel( 0 );
         ChannelServices.RegisterChannel( channel, false );
```

```
            FileStream fStream = new FileStream( "Adder.txt",FileMode.Open );
            SoapFormatter soapFormatter = new SoapFormatter();
            IAdder obj = soapFormatter.Deserialize( fStream ) as IAdder;

            double d = obj.Add( 3.0, 4.0 );
        }
...
```

该示例可用于验证.NET Remoting为每个MarshalByRefObject派生类的实例赋予了一个唯一的标识。换句话说，每个远程对象都有一个独一无二的基于GUID（Global Unique Identity）的URI。这个URI类似下面这样。

```
/6b03659f_0164_43e2_99cf_f36eda31adae/367459709_1.rem
```

当发布的对象被销毁时，引用它的文件就没用了。仅当对象由客户端激活或者由服务器发布时，客户端才能通过这个URI与之通讯。换句话说，如果该对象是一个WKO对象，客户端将无法通过这个URI与之通讯。

对象发布技术是一种介于客户端远程对象激活（CAO）和服务器远程对象激活服务（WKO）之间的解决方案。服务器激活了对象，却由客户端负责对象的识别。我们会在本章的末尾总结这四种激活模式的差异。

对象发布文件是由大约三十行较难读懂的XML代码构成的。下面是一些相关的摘录。

Adder.txt

```
...
<uri id="ref-2">/6b03659f_0164_43e2_99cf_f36eda31adae/367459709_1.rem</uri>
...
<serverType id="ref-5">RemotingServer.Adder, Server,
 Version=1.0.1140.28752, Culture=neutral, PublicKeyToken=null</serverType>
...
<item id="ref-8">RemotingInterfaces.IAdder, Interface,
 Version=1.0.1138.27103, Culture=neutral, PublicKeyToken=null</item>
...
<a3:CrossAppDomainData id="ref-9" xmlns:a3=
 "http://schemas.microsoft.com/clr/ns/System.Runtime.Remoting.Channels">
<_ContextID>1362648</_ContextID>
<_DomainID>1</_DomainID>
<_processGuid id="ref11">faf88595_9b2e_4f23_99ee_1d006915a98</_processGuid>
</a3:CrossAppDomainData>
...
<item id="ref-13">http://213.26.48.1:65100</item>
...
```

22.13.5　消息接收器

总的来说，在.NET Remoting下，方法的调用是通过交换两个消息来完成的，其中一个包含方法执行时所需的信息，而另一个则包含方法执行后所产生的信息。这两个消息中的每一个都经过了位于客户端AppDomain的真实代理和位于服务器端AppDomain的栈生成器接收器之间的一些处理。例如，包含方法输入参数的消息在客户端序列化后在服务器端反序列化。这个处理是由消息接收器对象完成的。具体地说，消息接收器是实现System.Runtime.Remoting.Messaging.IMessageSink接口并应用SerializableAttribute的类的实例。

```
public interface System.Runtime.Remoting.Messaging.IMessageSink {
  IMessageSink NextSink{get;}
  IMessage     SyncProcessMessage(IMessage request);
  IMessageCtrl AsyncProcessMessage(IMessage request,IMessageSin replySink);
}
```

由于NextSink属性的存在，消息接收器可以相互链接起来。你可能已经猜到，同步调用的信息处理是通过SyncProcessMessage()方法完成的，而异步调用的信息处理则是通过AsyncProcessMessage()方法完成的。下面的类实现告诉了我们应该将消息处理代码放置在何处。

你必须为消息接收器的类标记上SerializableAttribute。

```
[Serializable]
public class MonMsgSink : IMessageSink{
   private IMessageSink m_NextSink;
   public  IMessageSink NextSink{ get { return m_NextSink; } }
   public  MonMsgSink( IMessageSink nextSink ) { m_NextSink = nextSink; }

   IMessage SyncProcessMessage( IMessage msgIn ) {
      // Here, you can work on msgIn.
      IMessage msgOut = m_NextSink.SyncProcessMessage( msgIn ) ;
      // Here, you can work on msgOut.
      return msgOut;
   }
   IMessageCtrl AsyncProcessMessage(IMessage msgIn,IMessageSink replySink){
      // Here, you can work with msgIn.
      // You can also add a message sink.
      IMessageCtrl msgCtrl=m_NextSink.AsyncProcessMessage(msgIn,replySink);
      // Here, you can indicate to the CLR that it doesn't need to
      // wait more than 1 second for the asynchronous call return
      // with the line: 'msgCtrl.Cancel(1000);'
      return msgCtrl;
   }
}
```

注意，SyncProcessMessage()方法可以处理输入和输出消息，而AsyncProcessMessage()只能处理输入消息。不过，AsyncProcessMessage()方法还可以参与用于处理方法调用所返回信息的消息接收器链的构建。自然地，这条消息链中的消息接收器的链接顺序将和原先的链接顺序完全相反。

现在你知道如何创建消息接收器了。我们将在本章稍后介绍如何把你自己的消息接收器注入方法调用的消息链中，还有更重要的——这样做可以得到什么好处。但在此之前，我们先介绍如何实现一个自定义的真实代理。接着我们将会接触到第一个消息接收器，它处在用于同步方法调用的消息链中。

22.13.6 为何考虑自定义真实代理

在示范如何创建自定义真实代理之前，最好先看几个如何使用自定义真实代理的例子。

可以使用自定义真实代理来追踪对远程对象的调用。

可以使用自定义真实代理来避免某些可以在本地执行的远程调用。可以实现一个信息缓存系统。当我们向远程对象请求信息时，真实代理可以首先查看一下能是否在本地找到所请求的信息。

可以使用自定义真实代理以一种透明的方式修改方法调用的参数。例如，可以把作为参数传递给方法的字符串从一种语言翻译成另一种语言。

可以使用自定义真实代理把透明代理转换成某个类型，而这个类型不一定是真实代理所持有的。我们不打算详述这个特性，但要做到这样，你的自定义真实代理必须实现System.Runtime.Remoting.IRemotingTypeInfo接口。**MSDN**中对这个接口的描述可以告诉你如何使用它。

还可以使用自定义真实代理来实现服务器之间的负载平衡机制。只需定期获取每台服务器的负载量，然后把对远程对象的调用路由到负载量低的服务器就行了。还可以在评估负载量时考虑每台服务器的处理能力。我们也建议根据每台服务器能完成的工作量比例的分布，随机地把请求路由到多台服

务器上。这种做法通常与采用一个基于每台服务器当前负载量的算法的效果等价。需要指出的是，多台服务器之间的负载平衡仅适用于无状态远程WKO对象的情况。

22.13.7　开发自定义真实代理

实现一个自定义真实代理的类并不复杂。你只需创建一个RealProxy的子类并实现IMessage Invoke(IMessage)方法就行了。当出现以同步方式调用与之相关联的透明代理时，CLR会自动调用这个方法。输入消息就是透明代理所创建的消息。输出消息就是表示方法返回的消息，它会自动传递给透明代理。要在Invoke()方法中传递同步调用，只需在消息链的第一个消息接收器上调用IMessage SyncProcessMessage(IMessage)。

下面是一个自定义真实代理类的示例。在这个例子中，Invoke()方法使用的消息接收器是由信道提供的。然而，我们将会看到在代理和信道之间还可以嵌入其他的消息接收器。在Invoke()方法中，我们只是简单的把方法的输入参数和返回值显示出来。注意，我们把服务器对象激活服务的URI放入了输入消息中。我们可以轻易地修改这一步以实现一个负载平衡系统。

例22-28　Client.cs

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Proxies;
using System.Runtime.Remoting.Messaging;
using System.Collections;
using RemotingInterfaces;
// You must use soapsuds.exe to le the client know
// about the 'Adder' class.
using RemotingServer;

public class CustomRealProxy : RealProxy {
   // URI for server acticvated object.
   String m_Uri;
   // References the first message sink of the HttpSender channel.
   IMessageSink m_MsgSink;

   public CustomRealProxy( Type type, String uri,
                           IChannelSender channelSender) : base(type) {
      m_Uri = uri;
      string unused;
      // Get the first message sink of 'channelSender'.
      m_MsgSink = channelSender.CreateMessageSink(m_Uri, null, out unused);
   }

   // This method is called by the CLR before any client call.
   public override IMessage Invoke( IMessage msgIn ) {
      // The URI of the remote object must be stored in msgIn.
      IDictionary d = msgIn.Properties;
      d["__Uri"] = m_Uri;

      // Display 'in parameters' stored in msgIn.
      IMethodCallMessage msgCall = (IMethodCallMessage) msgIn;
      Console.Write( "CustomRealProxy: Before calling:{0}(",
                     msgCall.MethodName );
      for ( int i = 0; i < msgCall.InArgCount; i++ )
        Console.Write(" {0}={1} ",msgCall.GetArgName(i),msgCall.GetArg(i));
```

```
      Console.WriteLine(“)”);

      // Perform the remote call !
      IMethodReturnMessage msgOut =
         (IMethodReturnMessage) m_MsgSink.SyncProcessMessage( msgIn );

      // Display the value returned in msgOut.
      Console.WriteLine( “CustomRealProxy: After calling:{0}() RetVal={1}”,
                         msgCall.MethodName, msgOut.ReturnValue );
      return msgOut;
   }
}


namespace RemotingClient {
   class Program {
      static void Main() {
         HttpChannel channel = new HttpChannel( 0 );
         ChannelServices.RegisterChannel( channel, false );

         // Build a custom real proxy.
         // We initialize it with the object type, the URI of
         // the WKO service and the channel it should use.
         CustomRealProxy proxy = new CustomRealProxy(
            typeof(Adder),
            “http://localhost:65100/AddService”,
            (IChannelSender) channel );
         IAdder obj = (IAdder) proxy.GetTransparentProxy();

         // Here, we haven’t contacted yet the server.

         // This line triggers the first call to the server.
         double d = obj.Add(3.0, 4.0);
      }
   }
}
```

下面是当我们在合适的服务器上使用以上代码时客户端的输出。

```
CustomRealProxy: Before calling:Add( d1=3  d2=4 )
CustomRealProxy: After calling:Add() RetVal=7
```

22.13.8　在类的所有实例上使用自定义真实代理

在先前的程序中，我们显式创建了真实代理并显式地请求其透明代理。可以让Adder类的所有实例各自都拥有一个自定义真实代理。这样，即使一个实例是通过new操作符来创建的，甚至我们是以非远程的方式来使用它的，该实例仍然会有它自己的自定义真实代理。事实上，我们在此介绍的这个技术通常是为了一个以非远程方式使用的对象（即普通对象）能够获得这样一个对象的真实代理所带来的好处。

为此，Adder类的实例必须继承自System.ContextBoundObject类。我们将在本章稍后解释ContextBoundObject类。接着，必须定义一个继承自System.Runtime.Remoting.Proxies.ProxyAttribute的.NET attribute类。必须重写这个类的MarshalByRefObject ProxyAttribute.CreateInstance(Type t)虚方法以便它能在所创建的对象及其引用之间嵌入真实代理。然后，必须为Adder类标记上该attribute 。下面的程序演示了上述内容。

例22-29

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Services;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Proxies;
using System.Runtime.Remoting.Activation;
public class CustomRealProxy : RealProxy {
   readonly bool m_bDisplay;
   readonly MarshalByRefObject m_TargetObj;

   public CustomRealProxy( MarshalByRefObject targetObj,
                           Type type, bool bDisplay ) : base(type) {
      m_bDisplay = bDisplay;
      m_TargetObj = targetObj;
   }

   public override IMessage Invoke( IMessage msgIn ) {
      IMessage msgOut;
      if ( msgIn is IConstructionCallMessage ) {
         IConstructionCallMessage ctorCallMsg =
             (IConstructionCallMessage) msgIn;

         // Get the default real proxy.
         RealProxy defaultRealProxy =
            RemotingServices.GetRealProxy( m_TargetObj );

         // Invoke the ctor on this real proxy
         defaultRealProxy.InitializeServerObject( ctorCallMsg );

         // Get the custom real proxy on the new object.
         msgOut=EnterpriseServicesHelper.CreateConstructionReturnMessage(
            ctorCallMsg, (MarshalByRefObject)GetTransparentProxy() );

         if ( m_bDisplay )
            Console.WriteLine( "CustomRealProxy: ctor call" );
      }
      else {
         IMethodCallMessage callMsg = (IMethodCallMessage) msgIn;

         if ( m_bDisplay )
            Console.WriteLine( "CustomRealProxy: Before calling:{0}",
                               callMsg.MethodName);

         msgOut = RemotingServices.ExecuteMessage( m_TargetObj, callMsg );

         if ( m_bDisplay )
            Console.WriteLine( "CustomRealProxy: After calling:{0}",
                               callMsg.MethodName);
      }
      return msgOut;
   }
}

[AttributeUsage(AttributeTargets.Class)]
public class CustomProxyAttribute : ProxyAttribute {
   bool m_bDisplay;
```

```
   public CustomProxyAttribute( bool bDisplay ) {
      m_bDisplay = bDisplay;
   }
   public override MarshalByRefObject CreateInstance( Type T ) {
      MarshalByRefObject targetObj = base.CreateInstance( T );
      RealProxy realProxy = new CustomRealProxy(targetObj, T, m_bDisplay);
      return (MarshalByRefObject) realProxy.GetTransparentProxy();
   }
}

// The true params indicates that we wish that the custom real
// proxy displays info on the console.
[CustomProxyAttribute(true)]
public class Adder : ContextBoundObject {
   public int Add( int a, int b ) { return a + b; }
}

public class Program {
   static void Main() {
      Adder obj = new Adder();
      obj.Add(5, 6);
   }
}
```

该程序在控制台的输出如下：

```
CustomRealProxy: ctor call
CustomRealProxy: Before calling:Add
CustomRealProxy: After calling:Add
```

22.13.9　读写方法调用的参数

我们已经看过如何从表示调用的消息里读取方法的参数。我们还可以使用自定义真实代理或者自定义消息接收器来修改调用的参数。例如，可以使用这个功能把作为参数传递的字符串从一种语言翻译成另一种语言。对包含在消息里的参数进行写访问是可以的，但没有像如下所示的读访问那么直接。

```
...
public override IMessage Invoke( IMessage msgIn ) {
  IMethodCallMessage callMsg = (IMethodCallMessage) msgIn;
  Console.Write("CustomRealProxy: Before calling:{0}(",callMsg.MethodName);
  for(int i=0; i< callMsg.InArgCount ; i++)
     Console.Write( " {0}={1} ",callMsg.GetArgName(i),callMsg.GetArg(i) );
  Console.WriteLine( ")" );
...
```

事实上，无论我们使用哪个接口操作消息，我们都无法对包含在消息里的参数进行写访问。换句话说，这些接口在参数表上只提供了get访问器而没有提供set访问器。要实现我们的目标，我们建议使用System.Runtime.Remoting.Messaging.MethodCallMessageWrapper类，它支持在参数数组上使用set访问器。下面的代码片段示范了这个操作。

```
...
public override IMessage Invoke( IMessage msgIn ) {
   IMethodCallMessage callMsg = (IMethodCallMessage) msgIn;
   MethodCallMessageWrapper callMsgWrapper =
      new MethodCallMessageWrapper( callMsg );
   object[] tmpArgs = callMsgWrapper.Args;
   tmpArgs[0] = 1; // Modifiy the first arg.
```

```
    tmpArgs[1] = 2; // Modifiy the second arg.
    callMsgWrapper.Args = methodArgs;
    callMsg = callMsgWrapper;

    msgOut = RemotingServices.ExecuteMessage( m_ObjTarget , callMsg );
...
```

22.14　信道

22.14.1　简介

信道是传送表示跨AppDomain方法调用的消息的实体。因此，在客户端和服务器端各自的AppDomain里至少有一个信道。不过，一个应用程序可以包含多个信道，而一个信道的实现可用于客户端或服务器端。

信道是实现了System.Runtime.Remoting.Channels.IChannel接口的类的实例。仅能为客户端所用的信道叫做*发送方信道*。根据定义，发送方信道要实现System.Runtime.Remoting.Channels.IChannelSender接口。仅能为服务器所用的信道叫做*接收方信道*。根据定义，接收方信道要实现System.Runtime.Remoting.Channels.IChannelReceiver接口。

.NET Framework通过System.Runtime.Remoting.Channels.Http.HttpChannel、System.Runtime.Remoting.Channels.Tcp.TcpChannel和System.Runtime.Remoting.Channels.Ipc.IpcChannel类提供了三个信道实现。这些类中的每个都可以用作发送方信道和接收方信道。

很明显，HTTP和TCP协议分别由HttpChannel和TcpChannel类支持。IpcChannel类则支持*命名管道*的概念。命名管道是一个Windows对象，同一台机器上的两个windows进程可以利用它实现通信。当然，也可以使用HTTP或TCP等网络协议来实现同一台机器上的两个Windows进程之间的通信。命名管道的优势在于它们是在操作系统层面实现的，因而没有使用任何的网络API。IPC是Inter Process Communication的首字母缩略词，有时它也表示*同箱通讯*。

为了在AppDomain中注册信道，可以使用ChannelServices.RegisterChannel()静态方法或者通过RemotingConfiguration.Configure()静态方法装载.NET Remoting的配置文件。这两项技术已在前几页介绍过了。

每个信道都有名字，并且在同一个AppDomain中两个信道不能同名。默认情况下，HttpChannel的信道实例的名字是“http”，TcpChannel的信道实例的名字是“tcp”，而IpcChannel的信道实例的名字是“ipc”。下面代码示范了如何为信道指定名字。

```
...
static void Main() {
   IDictionary prop = new Dictionary<string,string>();
   prop[“name”] = “tcp2”;
   prop[“port”] = “65101”;
   ChannelServices.RegisterChannel( new TcpChannel( prop, null, null ) );
...
```

每个HTTP或者TCP类型的信道都需要一个端口号。同一台机器上的多个信道不能使用相同的端口号。为了避免端口选择的冲突，可以在创建信道时将端口指定为0。这样CLR会寻找一个空闲的端口号并分配给这个信道。你可能已经猜到，微软所实现的HTTP和TCP信道在内部使用了套接字。

当两个AppDomain处在同一个进程中时，不应该使用重量级的机制用于进程内部的通讯。因此，mscorlib.dll程序集包含了专门用于客户端的AppDomain和服务器端的AppDomain处在同一个进程内的情况的CrossAppDomain内部类。这个类由CLR自动使用。不必做任何事就可以获得这方面的优化。

22.14.2　发送方信道和代理

处在同一个AppDomain内的发送方信道和代理之间的关系经常遭到误解。这会导致理解上的偏差，例如为每个远程对象创建发送方信道。

在AppDomain中的发送方信道是供代理使用的消息接收器链的工厂。让我们回顾之前示范创建自定义真实代理的程序（例22-28）。CustomRealProxy类的构造函数需要知晓用于联系远程对象的发送方信道。

```
...
   public CustomRealProxy(Type type, String uri,
                          IChannelSender channelSender): base(type){
      m_Uri = uri;
      string unused;
      // Obtain a message sink.
      m_MsgSink = channelSender.CreateMessageSink(m_Uri, null, out unused);
   }
...
```

IChannelSender.CreateMessageSink()方法请求底层信道创建消息接收器链。这条链主要通过远程对象激活服务和信道端口来进行配置。

22.14.3　接收方信道和服务器对象

处于同一个AppDomain内的接收方信道和服务器对象之间的关系经常遭到误解。这可能导致理解上的偏差，例如为每个服务器对象创建接收方信道。

之所以对此产生误解部分是由于用于暴露WKO对象激活服务的RemotingConfiguration.RegisterWellKnownServiceType()方法和用于在服务器上暴露一个类的RemotingConfiguration.RegisterActivatedServiceType()方法都没有接受一个接收方信道作为参数。原因很简单：在AppDomain中的每个接收方信道都能接收对这个AppDomain中每个能以远程方式使用的对象的调用。

调用RemotingConfiguration.RegisterWellKnownServiceType()方法会在WKO服务的URI与WKO服务本身之间创建一个内部关联。服务根据调用模式和类来进行配置。调用RemotingConfiguration.RegisterActivatedServiceType()方法会在内部记录某个MarshalByRefObject的继承类，该类可以由远程客户端实例化。记住，每次创建ObjRef的实例来引用AppDomain内的对象时，都会为该对象创建一个基于GUID的唯一URI。随后，会在这个URI与真实对象之间建立一个内部关联。

当一个接收方信道接收到表示调用的信息时，可能出现以下三种情况。

- 这个调用在一个WKO服务上完成。在这种情况下，如果调用模式是single-call模式，将会激活一个新对象。如果调用模式是singleton，要么URI所指定的对象已存在，要么一个新的对象被创建出来。
- 这个调用在这个AppDomain中与所提供的URI相关联的对象上完成。如果与这个URI相关联的对象还存在，这个调用将在这个对象上完成。否则将向客户端发送信息告知对象已不存在了。
- 这个调用是对一个CAO类的构造调用。在这种情况下，将会创建出一个新对象。该对象会关联到一个新的基于GUID生成的URI。这个新的URI将通过ObjRef类的实例返回给客户端。

22.14.4　消息接收器、格式化程序和信道

在服务器端，表示调用的消息也是由相互链接在一起的消息接收器处理的。然而，接收方信道和发送方信道之间在概念上存在着一个根本的差异。发送方信道为每个远程对象的真实代理创建一个消息接收器链。接收方信道在其创建之时创建了消息接收器链。这条链将为所有通过这个接收方信道进

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。图22-6展示了这个差异。

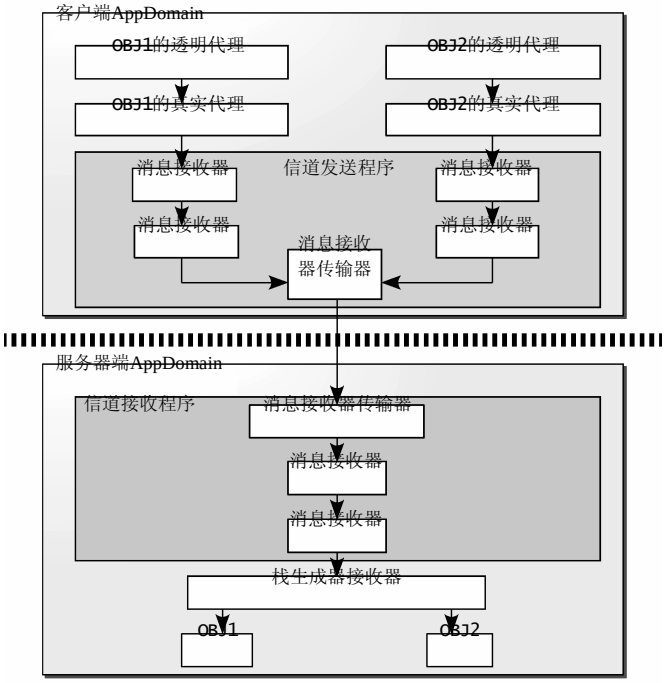

图22-6　消息接收器和信道

22.14.5　信道接收器提供程序

为了创建消息接收器链，你需要使用*信道接收器提供程序*（channel sink provider）。发送方信道的信道接收器提供程序是实现了`System.Runtime.Remoting.Channels.IClientChannelSinkProvider`接口的类的实例。相应地，接收方信道的信道接收器提供程序则实现了`IServerChannelSinkProvider`接口。需要提醒的是，在发送方信道中，每个真实代理都需要一条消息接收器链，而在接收方信道中则只需一条消息接收器链。

这两个接口都带有一个名为`CreateSink()`的方法，在信道的实现中会调用该方法以便从提供程序那里获取一个新的消息接收器。

```
IServerChannelSink IServerChannelSinkProvider.CreateSink(
      IChannelReceiver  channel);
IClientChannelSink IClientChannelSinkProvider.CreateSink(
      IChannelSender    channel,
      String            url,
      Object            remoteChannelData);
```

为了创建一条提供程序链，这两个接口都提供了`Next { get; set; }`属性。可以在代码中方便地创建这样一条链，并通过信道的实现类中某个适当的构造函数把它提供给信道。不过，我们通常选择通过使用配置文件来配置提供程序链。接下来的一节中将演示该方法。

22.14.6　示例：显示网络消息的大小

我们将会创建我们自己的消息接收器，并为这些消息接收器创建相应的接收器提供程序。这个消息接收器的功能是在客户端的控制台上显示发送和接收的字节数。

1．消息接收器和消息接收器提供程序

我们需要为这个项目开发四个类，如下所示。

- 一个名为CustomClientSink的类，它的实例是消息接收器。这个类的实例必须放置在发送方信道中的格式化程序之后。
- 一个名为CustomServerSink的类，它的实例是消息接收器。这个类的实例必须放置在接收方信道中的格式化程序之前。
- 一个名为CustomClientSinkProvider的类，它的实例是CustomClientSink的实例的提供程序。这个类的实例必须链接在发送方信道中格式化程序的提供程序之后。
- 一个名为CustomServerSinkProvider的类，它的实例是CustomServerSink的实例的提供程序。这个类的实例必须链接在接收方信道中格式化程序的提供程序之前。

我们还要开发一个Helper静态类，它所提供的GetStreamLength()方法将返回数据流的长度。无论数据流是否支持随机访问（查找），该方法都能够返回正确的结果。下面是所有这些类的代码。

例22-30　CustomChannelSink.cs

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.Collections;
using System.IO;

namespace CustomChannelSink {
```

```
internal static class Helper {
  public static Stream GetStreamLength(Stream inStream,out long length){
     // Does 'inStream' support seek access?
     if (inStream.CanSeek) {
        length = inStream.Length;
        return inStream;
     }
     // Here 'seek acces' is not supported. We must copy 'inStream'
     // to 'outStream' to obtain the stream length.
     Stream outStream = new MemoryStream();
     byte[] buffer = new Byte[1024];
     int tmp, nBytesRead = 0;
     while ( ( tmp = inStream.Read( buffer, 0, 1024 ) ) > 0 ) {
        outStream.Write( buffer, nBytesRead, tmp );
        nBytesRead += tmp;
     }
     outStream.Seek( 0, SeekOrigin.Begin );
     length = nBytesRead;
     return outStream;
  }
}

//
// Custom Client Sink.
//
public class CustomClientSink : BaseChannelSinkWithProperties,
                                IClientChannelSink {
   private IClientChannelSink m_NextSink;
   public CustomClientSink( IClientChannelSink nextSink ) {
      m_NextSink = nextSink;
   }
   public IClientChannelSink NextChannelSink {
      get { return m_NextSink; }
   }
   public void AsyncProcessRequest( IClientChannelSinkStack sinkStack,
                                    IMessage msgIn,
                                    ITransportHeaders headers,
                                    Stream msgStream ) {
      long length;
      msgStream = Helper.GetStreamLength( msgStream, out length );
      Console.WriteLine(
         "CustomClientSink:Async, length of the request stream {0}",
         length );
      // Chaining message sink for async return processing.
      sinkStack.Push( this, null );
      m_NextSink.AsyncProcessRequest(sinkStack,msgIn,headers,msgStream);
   }
   public void AsyncProcessResponse(
                        IClientResponseChannelSinkStack sinkStack,
                        Object state,
                        ITransportHeaders headers,
                        Stream msgStream) {
      long length;
      msgStream = Helper.GetStreamLength( msgStream, out length );
      Console.WriteLine(
```

```
            "CustomClientSink:Async, length of the response stream {0}",
            length);
         m_NextSink.AsyncProcessResponse(
                                    sinkStack, state, headers, msgStream);
      }
      public Stream GetRequestStream(IMessage msg,
                                    ITransportHeaders headers) {
         return m_NextSink.GetRequestStream( msg, headers );
      }

      public void ProcessMessage( IMessage msg,
                                 ITransportHeaders headersIn,
                                 Stream msgInStream,
                                 out ITransportHeaders headersOut,
                                 out Stream msgOutStream) {
         long length;
         msgInStream = Helper.GetStreamLength( msgInStream, out length );
         Console.WriteLine(
            "CustomClientSink:Sync, length of the request stream {0}",
            length);
         m_NextSink.ProcessMessage(msg, headersIn, msgInStream,
                                   out headersOut, out msgOutStream);
         msgOutStream = Helper.GetStreamLength( msgOutStream, out length );
         Console.WriteLine(
            "CustomClientSink:Sync, length of the response stream {0}",
            length);
      }
   }

   //
   // Custom Server Sink.
   //
   public class CustomServerSink : BaseChannelSinkWithProperties,
                                   IServerChannelSink {
      private IServerChannelSink m_NextSink;
      public CustomServerSink( IServerChannelSink nextSink ) {
         m_NextSink = nextSink;
      }
      public IServerChannelSink NextChannelSink {
         get { return m_NextSink; }
      }
      public void AsyncProcessResponse(
                        IServerResponseChannelSinkStack sinkStack,
                        object state,
                        IMessage msg,
                        ITransportHeaders headers,
                        Stream msgStream) {
         long length;
         msgStream = Helper.GetStreamLength( msgStream, out length );
         Console.WriteLine(
            "CustomServerSink:Async, length of the response stream {0}",
            length);
         m_NextSink.AsyncProcessResponse(
                           sinkStack, state, msg, headers, msgStream);
      }
```

```
    public Stream GetResponseStream(
                      IServerResponseChannelSinkStack sinkStack,
                      object state,
                      IMessage msg,
                      ITransportHeaders headers) {
       return null;
    }

    public ServerProcessing ProcessMessage(
                      IServerChannelSinkStack sinkStack,
                      IMessage msgIn,
                      ITransportHeaders headersIn,
                      Stream msgInStream,
                      out IMessage msgOut,
                      out ITransportHeaders headersOut,
                      out Stream msgOutStream) {
       long length;
       msgInStream = Helper.GetStreamLength(msgInStream, out length);
       Console.WriteLine(
          "CustomServerSink:Sync, length of the request stream {0}",
          length);
       // Chaining message sink for return processing.
       sinkStack.Push( this, null );
       ServerProcessing svrProc = m_NextSink.ProcessMessage(
                      sinkStack, msgIn, headersIn, msgInStream,
                      out msgOut, out headersOut, out msgOutStream);
       msgOutStream = Helper.GetStreamLength(msgOutStream,out length);
       Console.WriteLine(
          "CustomServerSink:Sync, length of the response stream {0}",
          length);
       return svrProc;
    }
}

//
// Custom Client Sink Provider.
//
public class CustomClientSinkProvider : IClientChannelSinkProvider {
   private IClientChannelSinkProvider m_NextProvider;
   public CustomClientSinkProvider(IDictionary prop,
      ICollection providerData) { }
   public IClientChannelSinkProvider Next {
      get { return m_NextProvider; }  set { m_NextProvider = value; }
   }
   public IClientChannelSink CreateSink(
                        IChannelSender channel,
                        string url,
                        object remoteChannelData ) {
      IClientChannelSink next =
         m_NextProvider.CreateSink( channel, url, remoteChannelData );
      Console.WriteLine(
         "CustomClientSinkProvider:Creating a message sink.");
      return new CustomClientSink( next );
   }
```

```
   }

   //
   // Custom Server Sink Provider.
   //
   public class CustomServerSinkProvider : IServerChannelSinkProvider {
      private IServerChannelSinkProvider m_NextProvider;
      public CustomServerSinkProvider( IDictionary prop,
                                       ICollection providerData) { }
      public IServerChannelSinkProvider Next {
         get { return m_NextProvider; } set { m_NextProvider = value; }
      }
      public IServerChannelSink CreateSink(IChannelReceiver canal) {
         IServerChannelSink next = m_NextProvider.CreateSink(canal);
         Console.WriteLine(
            "CustomServerSinkProvider:Creating a message sink.");
         return new CustomServerSink( next );
      }
      public void GetChannelData( IChannelDataStore channelData ) { }
   }
}
```

需要提及以下几点。

- 在提供程序的构造函数中，作为参数传递的字典对应了你希望赋予提供程序的属性。这里，我们没有用到这个功能。如果需要的话，我们只需在配置文件中加入下面的代码。

```
<provider
  type = "CustomChannelSink.CustomClientSinkProvider,ChannelSink"
  Prop1="hello"/>
```

  可以这样获取属性的值：

```
public CustomClientSinkProvider( IDictionary prop,
                                 ICollection providerData) {
   string s = (string) Prop["Prop1"];
   ...
```

  这样，就可以配置你的提供程序了。例如，可以为*数据流*指定压缩或者加密算法的名称。

- 在服务器端的消息接收器中使用的`IServerChannelSinkStack`类型与异步调用有关。事实上，这个栈中将保存那些在服务器端发送返回消息时所用到的消息接收器。
- 在客户端和服务器端的消息接收器上使用的`ITransportHeaders`类型可用于传递与消息相关的信息。例如，如果消息接收器的任务是加密/解密一个*流*，那么可以在这个头中指定消息实际上是否已加密。

```
class CustomClientSink{ ...
   public void ProcessMessage(
           IMessage              msg,
           ITransportHeaders     headersIn, ...){
      headersIn["Crypted"] ="Yes";
      ...
```

  这使得接收方信道中的消息接收器在试图解密消息时可以判断它是否真的经过加密。

```
class CustomServerSink{ ...
   public ServerProcessing ProcessMessage(
            IServerChannelSinkStack   sinkStack,
            IMessage                  msgIn,
            ITransportHeaders         headersIn,...){
      string sCrypted = (string) HeadersIn["Crypted"] ;
      if( sCrypted != null && sCrypted == "Yes") { ...
```

  接收方信道也因此变得更加灵活，因为无论消息是否经过加密它都能够对它们进行处理。

2．服务器端

在服务器端，所有与信道相关的信息都可以在配置文件中找到。

Server.cs

```
...
static void Main() {
   RemotingConfiguration.Configure("Server.config");
   Console.WriteLine("Press a key to stop the server...");
   Console.Read();
}
...
```

接着我们在HTTP接收方信道中指定二进制格式化程序。

例22-31　Server.config

```
<configuration>
   <system.runtime.remoting>
      <application name = "Server">
         <service>
            <wellknown type="RemotingInterfaces.Adder,Interface"
             mode ="Singleton" objectUri="Service1.rem" />
         </service>
         <channels>
            <channel port="65100" ref ="http">
               <serverProviders>
                  <provider type=
                 "CustomChannelSink.CustomServerSinkProvider,ChannelSink" />
                  <formatter ref="binary"/>
               </serverProviders>
            </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

3．客户端

在客户端，所有相关的信息也存储在配置文件中。

Client.cs

```
...
static void Main() {
   RemotingConfiguration.Configure("Client.config",false);

   Adder objA = new Adder();
   double dA = objA.Add( 3.0 , 4.0 );
   Console.WriteLine("Returned value:" + dA);
   Adder objB = new Adder ();
   double dB = objB.Add( 3.0 , 4.0 );
   Console.WriteLine("Returned value:" + dB);
}
...
```

这里，我们也在HTTP发送方信道中指定使用二进制格式化程序。

例22-32　Client.config

```
<configuration>
   <system.runtime.remoting>
      <application name = "Client">
         <client>
            <wellknown type="RemotingInterfaces.Adder,Interface"
             url="http://localhost:65100/Service1.rem" />
         </client>
         <channels>
            <channel ref ="http">
               <clientProviders>
                  <formatter ref="binary"/>
                  <provider type =
                 "CustomChannelSink.CustomClientSinkProvider,ChannelSink" />
               </clientProviders>
            </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

4．执行这个示例

下面是服务器端在控制台上的输出。

```
CustomServerSinkProvider:Creating a message sink.
Press a key to stop the server...
CustomServerSink:Sync, length of the request stream 155
CAdditionneur ctor
CAdditionneur Add( 3 + 4 )
CustomServerSink:Sync, length of the response stream 32
CustomServerSink:Sync, length of the request stream 155
CAdditionneur Add( 3 + 4 )
CustomServerSink:Sync, length of the response stream 32
```

下面是客户端在控制台上的输出。

```
CustomClientSinkProvider:Creating a message sink.
CustomClientSink:Sync, length of the request stream 155
CustomClientSink:Sync, length of the response stream 32
Returned value:7
CustomClientSinkProvider:Creating a message sink.
CustomClientSink:Sync, length of the request stream 155
CustomClientSink:Sync, length of the response stream 32
Returned value:7
```

如果我们选择的是SOAP格式化程序而不是二进制格式化程序，调用的流的大小将会是574字节，而返回的流的大小将会是586字节。

如果我们打算使用压缩或加密库，这个示例可以方便地调整来压缩或加密通过网络传输的二进制数据。

22.15　.NET 上下文

22.15.1　简介

我们已经看到AppDomain能够在运行期间实现类型、安全和异常层面的隔离。然而还存在着比AppDomain更加细粒度的用于存储.NET对象的实体。一个.NET AppDomain能够包含多个这种叫做.NET上下文的实体。在本章，只要不会与COM+上下文概念发生混淆，我们就把它们称作上下文，这两个概念是完全不同的（COM+上下文在8.8节介绍）。某些作者使用托管上下文（managed context）

或者执行上下文（execution context）来指代.NET上下文，而非托管上下文则指代COM+上下文。所有.NET对象都存在于上下文中，每个AppDomain中至少存在一个上下文。这个上下文称为AppDomain的默认上下文，它在AppDomain创建的时候就创建了。图22-7总结了它们之间的关系。

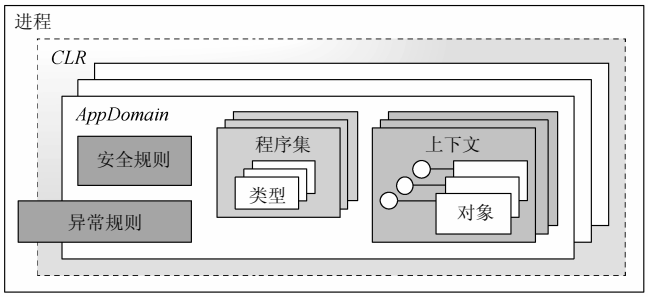

图22-7　进程、AppDomain和上下文

上下文的概念使得拦截对象的调用成为可能。拦截调用意味着我们可以对每个输入或输出参数做一个或多个转换。这些转换是通过消息接收器来完成的。这样做主要是为了让客户端在调用对象的方法时不会察觉到调用被拦截了以及在方法处理的前后对参数做了转换。

22.15.2　上下文绑定和上下文灵活对象

根据上一节的内容，上下文可以看作AppDomain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AppDomain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，即使是那些同一个AppDomain中的实体所执行的调用。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，甚至这个类的方法中所使用的`this`引用也是透明代理而不是对这个对象的直接引用。让`ContextBoundObject`类继承自`MarshalByRefObject`是合理的，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是*上下文绑定的*。没有继承自`ContextBoundObject`的类的实例则被视为是*上下文灵活的*。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。图22-8说明了以上内容。

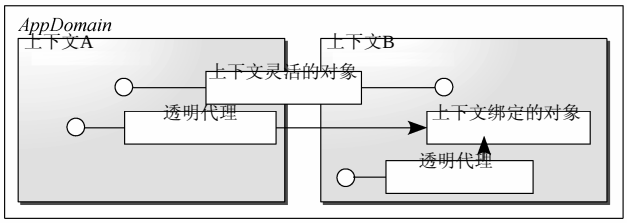

图22-8　上下文绑定的对象与上下文灵活的对象

22.15.3　上下文 attribute 和上下文属性

下面我们将介绍用于在上下文层面注入消息接收器的技术。首先介绍上下文attribute和上下文属性的概念。

1．上下文attribute

上下文attribute是指应用在上下文绑定类上的.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都要判断这个对象的创建者所在的上下文是否适用。下面这个方法执行了这个操作。

```
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都可以在这个新的上下文中注入一个或多个上下文属性。这些注入可以通过以下方法实现。

```
public void IContextAttribute.GetPropertiesForNewContext(
                                          IConstructionCallMessage ctorMsg)
```

2．上下文属性

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都注入了它的属性，那么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，如下所示：

```
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。

对消息接收器区域的介绍是下一小节的主题。在此之前，让我们先通过示例演示上下文attribute和上下文属性的概念。对于那些读过信道那节的读者，上下文attribute的角色相当于在信道中注入提供程序的配置文件。同样地，上下文属性的角色相当于消息接收器提供程序。

3．使用上下文attribute和属性的示例

下面的程序定义了LogContextAttribute类和LogContextProperty类。应用了LogContext-Attribute的类的所有实例都将承载在具有LogContextProperty类型的属性的上下文中。于是，该实例就可以访问这个属性提供的服务了。这个服务允许通过调用LogContextProperty.Log(string)方法向文件写入一个字符串。文件名是LogContextAttribute的参数。这样，我们就可以让每个类都拥有一个配置文件。当应用了LogContextAttribute的类的新实例创建好后，boolLogContextAttribute.IsContextOK(Context)方法将判断调用构造函数的实体所在的上下文是否包含带有相同文件名的LogContextAttribute的实例。如果不是，则会创建一个新的上下文。LogContextAttribute.GetPro-

pertiesForNewContext(IConstructionCallMessage ctor)方法创建一个LogContextProperty的实例。在这个方法返回时，CLR自动把新的属性注入新的上下文。下面是程序的代码。

例22-33

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Runtime.Remoting.Activation;
using System.Threading;

public class LogContextProperty : IContextProperty {
   public LogContextProperty(string sFileName) { m_sFileName = sFileName; }
   string m_sFileName;
   public string sFileName { get { return m_sFileName; } }
   public string Name { get { return "Log"; } }
   public bool IsNewContextOK( Context ctx ) { return true; }
   public void Freeze( Context ctx ) { }
   public void Log( string sLog ) {
      // We just write logs on the console.
      Console.WriteLine( "ContextID={0} To write '{1}' in the file '{2}'",
                         Thread.CurrentContext.ContextID,
                         sLog,
                         m_sFileName);
   }
}

[AttributeUsage(AttributeTargets.Class)]
public class LogContextAttribute : Attribute, IContextAttribute {
   string m_sFileName;
   public LogContextAttribute(string sFileName){ m_sFileName = sFileName; }
   // No need to create a new context if the current one already
   // contains the proper context property.
   public bool IsContextOK( Context currentCtx,
                            IConstructionCallMessage ctor) {
      LogContextProperty prop = currentCtx.GetProperty( "Log" )
                                                    as LogContextProperty;
      if ( prop == null ) return false;
      return ( prop.sFileName == m_sFileName );
   }
   public void GetPropertiesForNewContext(IConstructionCallMessage ctor) {
      IContextProperty prop = new LogContextProperty( m_sFileName );
      ctor.ContextProperties.Add( prop );
   }
```

```
}

[LogContextAttribute("LogFoo.txt")]
public class Foo : ContextBoundObject {
   public Foo CreateNewInst() { return new Foo(); }
   public int Add( int a, int b ) {
      string s = string.Format("Add {0}+{1}", a, b);
      Context ctx = Thread.CurrentContext;
      LogContextProperty logger = ctx.GetProperty("Log")
                                                    as LogContextProperty;
      logger.Log( s );
      return a + b;
   }
}

public class Program {
   static void Main() {
      Foo obj1 = new Foo();
      obj1.Add(4, 5);
      Foo obj2 = new Foo();
      obj2.Add(6, 7);
      Foo obj3 = obj1.CreateNewInst();
      obj3.Add(8, 9);
   }
}
```

该程序输出：

```
ContextID=1 To write 'Add 4+5' in the file 'LogFoo.txt'
ContextID=2 To write 'Add 6+7' in the file 'LogFoo.txt'
ContextID=1 To write 'Add 8+9' in the file 'LogFoo.txt'
```

obj1和obj3这两个对象位于同一个上下文中，因为obj3是在obj1所在的上下文中创建的。

22.15.4　消息接收器区域

消息接收器区域有以下四种：服务器区域、对象区域、信使区域和客户端区域。要理解区域概念，必须考虑上下文绑定的对象是否被位于另一个上下文中的实体调用。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，而把被调用对象所在的上下文称为目标上下文。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上下文。
- 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。于是，上下文中的每个对象都有一个信使区域。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。

图22-9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。因为上下文属性类是可以自定义的，所以我们可以选择哪个区域必须注入消息接收器。

你可能注意到，每个区域包含一个用于通知CLR退出区域的系统终结器接收器，不过不需要太在意它。

当调用方上下文和目标上下文处在同一个AppDomain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22-9展示了这些实例。

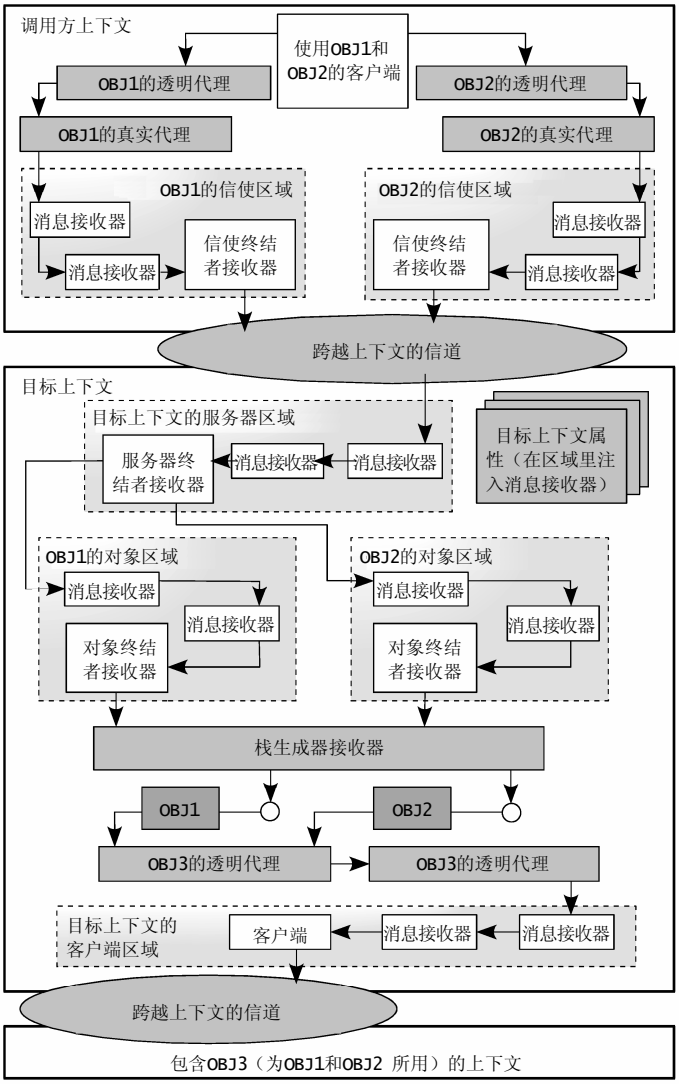

图22-9　上下文和区域

22.15.5 使用区域的示例

我们想通过例22-34的程序说明以下内容。

- 自定义上下文attribute（CustomDisplayContextAttribute类）的代码在上下文中注入了自定义上下文属性（CustomDisplayContextProperty类）。这个上下文属性在目标上下文的对象区域、服务器区域和客户端区域以及调用方上下文的信使区域中注入了自定义消息接收器（CustomDisplayMessageSink类）。
- 消息接收器的行为可以通过传递给上下文attribute的参数修改。这里，这个参数是一个布尔值，它指示消息接收器是否必须在控制台上输出某些东西。
- 消息接收器通过上下文属性注入这四个区域。CustomDisplayContextAttribute上下文attribute确保为Foo类的每个实例创建上下文。我们首先创建Foo的实例，然后我们在这个实例上调用一个方法并穿越信使区域、服务器区域和对象区域的消息接收器。要穿越客户端区域的消息接收器，我们创建了Foo的第二个实例，然后让第一个实例调用它。于是，这里有了三个上下文：执行Main()方法的上下文（ContextID = 0）、包含Foo的第一个实例的上下文（ContextID = 1）和包含Foo的第二个实例的上下文（ContextID = 2）。
- 上下文内部调用不会触发所有这些消息接收器的调用。这在Foo的第一个实例调用自身时可以看到。
- 在区域（这里是客户端区域）中注入了多个消息接收器。
- CLR在区域中注入消息接收器的时间。我们很清楚的看到这个时间依赖于区域的类型。

程序如下：

例22-34

```
using System;
using System.Runtime.Remoting.Contexts;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Activation;
using System.Threading;
using System.Collections;

//
// Instances of the context property class inject message sinks
// in all regions.
//
public class CustomDisplayContextProperty :
      IContextProperty,
      IContributeEnvoySink,
      IContributeObjectSink,
      IContributeServerContextSink,
      IContributeClientContextSink {
   public CustomDisplayContextProperty( bool bDisplay ) {
      m_bDisplay = bDisplay;
   }
   bool m_bDisplay;
   public bool bDisplay { get { return m_bDisplay; } }

   // IContextProperty
   public string Name { get { return "PropDisplay"; } }
   public bool IsNewContextOK(Context ctx) { return true; }
   public void Freeze( Context ctx ) {
      Console.WriteLine( "   Freeze ContextID={0}", ctx.ContextID );
```

```
    }

    // Inject two message sinks in the 'client' region.
    public IMessageSink GetClientContextSink( IMessageSink nextSink) {
       Console.WriteLine("   GetClientContextSink()");
       IMessageSink nextnextSink = new CustomDisplayMessageSink(
                                nextSink, "Client region1  ", m_bDisplay);
       return new CustomDisplayMessageSink(
                             nextnextSink, "Client region2  ", m_bDisplay);
    }

    // Inject a message sink in the 'server' region.
    public IMessageSink GetServerContextSink( IMessageSink nextSink) {
       Console.WriteLine("   GetServerContextSink()");
       return new CustomDisplayMessageSink(
                                nextSink, "Server region   ", m_bDisplay);
    }

    // Inject a message sink in the 'envoy' region.
    // NOTE: You can use 'mbro' to obtain a reference on the object.
    public IMessageSink GetEnvoySink( MarshalByRefObject mbro,
                                      IMessageSink nextSink) {
       Console.WriteLine("   GetEnvoySink()");
       return new CustomDisplayMessageSink(
                                nextSink, "Envoy  region   ", m_bDisplay);
    }

    // Inject a message sink in the 'object' region.
    public IMessageSink GetObjectSink( MarshalByRefObject mbro,
                                       IMessageSink nextSink) {
       Console.WriteLine("   GetObjectSink()");
       return new CustomDisplayMessageSink(
                                nextSink, "Object region   ", m_bDisplay);
    }
}

//---------------------------------------------------------------------
//
// Context attribute class. It forces the creation of one context per
// object. It injects a 'CustomDisplayContextProperty' in each created
// context.
//
[AttributeUsage(AttributeTargets.Class)]
public class CustomDisplayContextAttribute : Attribute, IContextAttribute {
   bool m_bDisplay;
   public CustomDisplayContextAttribute( bool bDisplay ) {
     m_bDisplay = bDisplay;
   }
   // Forces creating a context per object.
   public bool IsContextOK( Context currentCtx,
                            IConstructionCallMessage ctor) {
      return false;
   }

   // Injects a CustomDisplayContextProperty in each created context.
   public void GetPropertiesForNewContext( IConstructionCallMessage ctor ){
      IContextProperty prop = new CustomDisplayContextProperty(m_bDisplay);
      ctor.ContextProperties.Add( prop );
   }
```

```
}


//---------------------------------------------------------------------
//
// Instances of the 'CustomDisplayMessageSink' class are message sinks
// that display info on console.
//
[Serializable]
public class CustomDisplayMessageSink : IMessageSink {
   IMessageSink m_NextSink;
   // Message to display.
   string m_sDisplay;
   // Display only if 'm_bDisplay' is true.
   bool m_bDisplay;

   public IMessageSink NextSink { get { return m_NextSink; } }
   public CustomDisplayMessageSink( IMessageSink nextSink,
                                    string sDisplay,
                                    bool bDisplay) {
      m_NextSink = nextSink;
      m_sDisplay = sDisplay;
      m_bDisplay = bDisplay;
   }
   public IMessage SyncProcessMessage( IMessage msg ) {
      if ( m_bDisplay )
         Console.WriteLine( "   Begin MsgSink:{0} ContextID={1}",
                            m_sDisplay, Thread.CurrentContext.ContextID);
      // Contact next message sink in the chain...
      IMessage retMsg = m_NextSink.SyncProcessMessage( msg );
      if ( m_bDisplay )
         Console.WriteLine( "   End   MsgSink:{0} ContextID={1}",
                            m_sDisplay, Thread.CurrentContext.ContextID);
      return retMsg;
   }
   public IMessageCtrl AsyncProcessMessage( IMessage msg,
                                            IMessageSink replySink) {
      return m_NextSink.AsyncProcessMessage( msg, replySink );
   }
}


//---------------------------------------------------------------------
//
// The 'Foo' class is tagged with a 'CustomDisplayContextAttribute'.
// The parameter 'true' means that message sinks
// must display info on console.
//
[CustomDisplayContextAttribute( true )]
public class Foo : ContextBoundObject {
   public Foo() { Console.WriteLine("   Foo ctor"); }
   public int Add(int a, int b) {
      Console.WriteLine( "   Add {0}+{1}", a, b );
      return a + b;
   }
   public int AddCross( Foo tmp, int a, int b ) {
      Console.WriteLine( "   Cross Add {0}+{1}", a, b );
      return tmp.Add( a, b );
```

```
   }
}


public class Program {
   static void Main() {
      Console.WriteLine( “Before constructing obj1.” );
      Foo obj1 = new Foo();
      Console.WriteLine( “Before using obj1.” );
      obj1.Add( 4, 5 );
      Console.WriteLine( “Before constructing obj2.” );
      Foo obj2 = new Foo();
      Console.WriteLine( “Before obj1 calls obj2.” );
      obj1.AddCross( obj2, 6, 7 );
      Console.WriteLine( “Before obj1 calls obj1.” );
      obj1.AddCross( obj1, 8, 9 );
   }
}
```

执行程序后的输出如下所示。

```
Before constructing obj1.
   Freeze ContextID=1
   GetServerContextSink()
   Begin MsgSink:Server region    ContextID=1
   Foo ctor
   GetEnvoySink()
   End   MsgSink:Server region    ContextID=1
Before using obj1.
   Begin MsgSink:Envoy  region    ContextID=0
   Begin MsgSink:Server region    ContextID=1
   GetObjectSink()
   Begin MsgSink:Object region    ContextID=1
   Add 4+5
   End   MsgSink:Object region    ContextID=1
   End   MsgSink:Server region    ContextID=1
   End   MsgSink:Envoy  region    ContextID=0
Before constructing obj2.
   Freeze ContextID=2
   GetServerContextSink()
   Begin MsgSink:Server region    ContextID=2
   Foo ctor
   GetEnvoySink()
   End   MsgSink:Server region    ContextID=2
Before obj1 calls obj2.
   Begin MsgSink:Envoy  region    ContextID=0
   Begin MsgSink:Server region    ContextID=1
   Begin MsgSink:Object region    ContextID=1
   Cross Add 6+7
   Begin MsgSink:Envoy  region    ContextID=1
   GetClientContextSink()
   Begin MsgSink:Client region2   ContextID=1
   Begin MsgSink:Client region1   ContextID=1
   Begin MsgSink:Server region    ContextID=2
   GetObjectSink()
   Begin MsgSink:Object region    ContextID=2
   Add 6+7
   End   MsgSink:Object region    ContextID=2
```

```
   End   MsgSink:Server region    ContextID=2
   End   MsgSink:Client region1   ContextID=1
   End   MsgSink:Client region2   ContextID=1
   End   MsgSink:Envoy  region    ContextID=1
   End   MsgSink:Object region    ContextID=1
   End   MsgSink:Server region    ContextID=1
   End   MsgSink:Envoy  region    ContextID=0
Before obj1 calls obj1.
   Begin MsgSink:Envoy  region    ContextID=0
   Begin MsgSink:Server region    ContextID=1
   Begin MsgSink:Object region    ContextID=1
   Cross Add 8+9
   Add 8+9
   End   MsgSink:Object region    ContextID=1
   End   MsgSink:Server region    ContextID=1
   End   MsgSink:Envoy  region    ContextID=0
```

22.15.6　调用上下文

可以在运行在调用方上下文中的消息接收器与运行在目标上下文中的消息接收器之间传递信息。我们使用这个技术主要是为了把调用方上下文的信息传递给目标上下文（例如调用方上下文支持某些属性）。我们把这项功能称为调用上下文。尽管它的名字中含有“上下文”这三个字，但是这项功能可以用在任何消息接收器中，包括那些不在上下文中的。

为此，必须首先定义一个实现System.Runtime.Remoting.Messaging.ILogicalThreadAffinative接口的类来描述要传递的信息。在运行在调用方上下文（在信使区域或者客户端区域里）中的消息接收器的代码中，我们必须把这个类的实例附加到表示调用的消息上。在运行在目标上下文（在对象区域或者服务器区域里）中的消息接收器的代码中，必须把这个类的实例从表示调用的消息中分离出来。这些操作是通过IMethodMessage.LogicalCallContext属性完成的。

下面的代码演示如何运用该技术。

例22-35

```
...
public class DataContext : ILogicalThreadAffinative {
   public int Data;
   public DataContext( int i ){ Data=i; }
}

[Serializable]
public class CustomEnvoyMessageSink : IMessageSink{
   public IMessage SyncProcessMessage( IMessage msg ){
       DataContext dc = new DataContext( 691 );
      // Include the data in 'msg'.
      ((IMethodMessage)Msg).LogicalCallContext.SetData( "TheDataID" , dc );
      return m_NextSink.SyncProcessMessage( msg );
   }
...
}

[Serializable]
public class CustomServerMessageSink : IMessageSink{
   public IMessage SyncProcessMessage( IMessage msg ){
      // Get the data from 'msg'.
      DataContext dc = (DataContext)
         ((IMethodCallMessage)Msg).LogicalCallContext.GetData(
```

```
            "TheDataID" );
         if( dc != null )
            Console.WriteLine( "   DataContext:" + dc.Data );
         IMessage retMsg = m_NextSink.SyncProcessMessage( msg );
         return retMsg;
      }
   ...
   }
   ...
```

我们在前一节看到，在客户端区域和信使区域中注入消息接收器是发生在调用对象的构造函数之后。客户端区域和信使区域中的消息接收器无法把信息附加到表示构造函数调用的消息上，而服务器区域的消息接收器却可能依然试图从表示构造函数调用的消息中分离出这个信息。在这种情况下，我们可以在上下文attribute的GetPropertiesForNewContext()方法中附加这个信息。

例22-36

```
   ...
   public class CustomDisplayContextAttribute : Attribute,
                                                IContextAttribute {
      public void GetPropertiesForNewContext(IConstructionCallMessage ctor) {
         DataContext dc = new DataContext( 10 );
         ctor.LogicalCallContext.SetData( "TheDataID", dc );
         IContextProperty prop = new CustomDisplayContextProperty(m_bDisplay);
         ctor.ContextProperties.Add( prop );
      }
   ...
   }
```

22.16　小结

22.16.1　激活对象的四种方式

我们已经看到，激活对象的方式有四种：WKO single-call模式、WKO singleton模式、CAO和对象发布。下面这张表总结了这些模式之间的主要差异。

	WKO single call	WKO singleton	CAO	发布
对象何时激活	在每次调用时	在客户端的第一次调用时	在客户端调用构造函数时	在服务器调用构造函数时
客户端需要哪些信息才能访问对象	指示终结点的URI	指示终结点的URI	在调用构造函数时隐式获取ObjRef	显式获取ObjRef
对象在多个客户端之间共享吗	否	是	否	是
客户端是否持有对象的ObjRef实例	否	否	是	是
如果对象被租约管理器销毁了，调用是否返回异常	否	否	是，如果不能自动构建对象的话	是，如果不能自动构建对象的话
客户端需要知道类的元数据吗	否，客户端仅需接口的元数据	否，客户端仅需接口的元数据	是，除非你使用factory设计模式	是
这个模式能通过配置文件来配置吗	否	是	是	否
使用带参数的构造函数能激活这个对象吗	否	否	否，除非你使用factory设计模式	是

22.16.2　截获消息

我们用了本章的一半专门来介绍如何拦截表示调用的消息以及为何这样做。图22-10简要总结了各种可能的拦截层面。

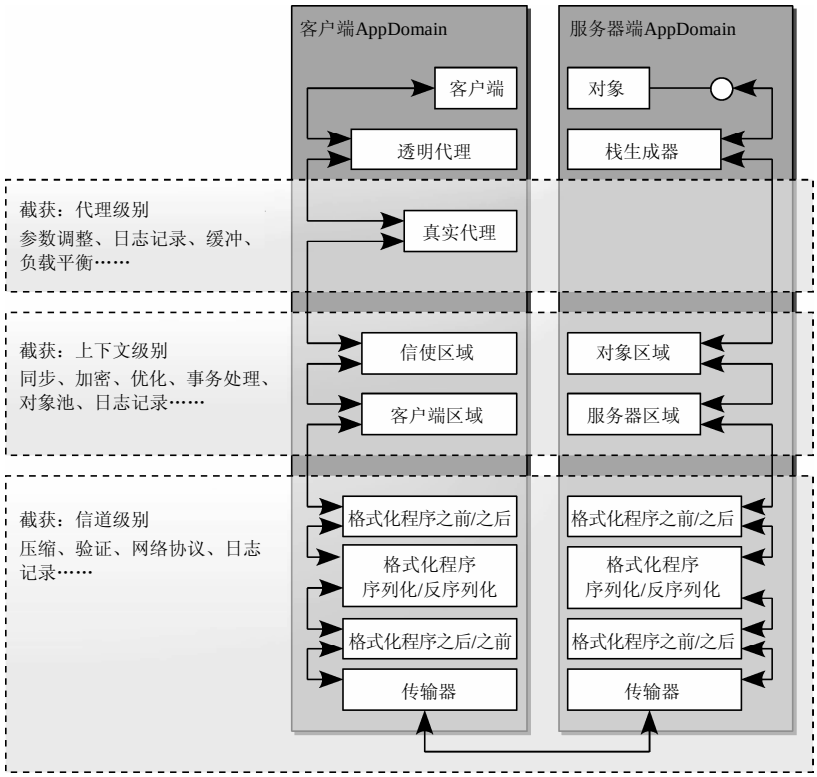

图22-10　拦截消息