第54章 .NET Remoting

本章内容：

- .NET Remoting 概述
- 把具有类似执行要求的对象组合在一起的上下文
- 实现一个简单的远程对象、客户端和服务器
- .NET Remoting 体系结构
- .NET Remoting 配置文件
- 在 ASP.NET 中驻留.NET Remoting 对象
- 使用 Soapsuds 访问远程对象的元数据
- 异步调用.NET Remoting 方法
- 利用事件调用客户端中的方法
- 使用 CallContext 自动把数据传递给服务器

本章将讨论.NET Remoting，它可以用来访问另一个应用程序域(如另一个服务器)中的对象。.NET Remoting 为客户端和服务器端的.NET 应用程序之间的通信提供了一种更为快速的格式。

本章将使用 HTTP、TCP 和 IPC 信道开发.NET Remoting 对象、客户端和服务器。首先以编程方式配置客户端和服务器，之后修改应用程序，以使用配置文件，其中只需要几个.NET Remoting 方法。本章还编写一些小程序，异步使用.NET Remoting，在客户端应用程序中调用事件处理程序。

.NET Remoting 类位于 System.Runtime.Remoting 名称空间及其子名称空间中，其中许多类在核心程序集 mscorlib 中，一些只用于跨网络通信的类可用于 System.Runtime.Remoting 程序集中。

54.1 使用.NET Remoting 的原因

.NET Remoting 是在不同应用程序域之间通信的技术。使用.NET Remoting 在不同应用程序域之间通信可以在同一个进程中、一个系统的进程之间或不同系统的进程之间进行。

对于客户端和服务器应用程序之间的通信，可以使用几种不同的技术。可以使用套接字编写应用程序，或使用 System.Net 名称空间中的一些辅助类，便于处理协议、IP 地址和端口号(详见第 24 章)。使用这种技术总是必须通过网络发送数据。所发送的数据可以是自己的自定义协议，其中由服务器解释数据包，这样服务器就知道应调用什么方法。我们不仅需要处理发送的数据，还需要自己创建线程。

使用 ASP.NET Web 服务，可以跨网络传递消息。通过 ASP.NET Web 服务，可以获得平台独立性。ASP.NET Web 服务不仅具有平台独立性，在客户端和服务器之间的耦合也比较松散，于是更容易处理版本问题。ASP.NET Web 服务详见第 55 章。

.NET Remoting 总是在客户端和服务器之间提供较紧密的耦合，因为它们共享相同的对象类型。.NET Remoting 给 CLR 对象提供了跨不同应用程序域调用方法的功能。

.NET Remoting 的功能可以用应用程序类型和所支持的协议描述，还可以通过 CLR Object Remoting 来描述。

CLR Object Remoting 是.NET Remoting 的一个重要方面。所有的语言结构(如构造函数、委托、接口、方法、属性和字段等)都可以与远程对象一起使用。.NET Remoting 跨网络扩展 CLR 对象的功能，CLR Object Remoting 可以处理激活、分布式标识、生命周期和调用上下文等方面的工作。它与 XML Web 服务大不相同。在 XML Web 服务中，对象是抽象的，客户端不需要知道服务器的对象类型。

目前，网络通信的最佳选择是第 43 章介绍的 WCF。WCF 提供 ASP.NET Web 服务的功能，如平台无关性，以及.NET Remoting 为.NET 与.NET 通信提供的性能和灵活性。.NET Remoting 仍具备优势的一个地方是进程内部的应用程序域之间的通信。第 50 章讨论的 MAF 技术(System.AddIn)在后台使用.NET Remoting。当然还有许多基于.NET Remoting 的现有.NET 解决方案，所以不能把.NET Remoting 重写为一门新技术。

尽管 SOAP 由.NET Remoting 提供，但不能假定它可用于不同平台之间的交互操作。SOAP Document 样式对于.NET Remoting 不可用。.NET Remoting 是为客户端和服务器端的.NET 应用程序而设计的。如果要进行交互操作，就应使用 Web 服务。

.NET Remoting 是一个极为灵活的体系结构，它可以用于通过任意方式传输的任意应用程序中，方法是使用任意的有效负载编码(payload endcoding)。

组合使用 SOAP 和 HTTP 只是调用远程对象的一种方式。传输信道是“可插入的”，也可以替换。在.NET 4 中，可以获取 HttpChannel 类、TcpChannel 类和 IpcChannel 类分别表示的 HTTP 信道、TCP 信道和 IPC 信道，还可以构建传输信道，以使用 UDP、IPX、SMTP、共享的内存机制或消息队列，至于选择使用哪一个，自己完全有权决定。

“可插入(pluggable)”这个术语通常与.NET Remoting 一起使用。“可插入”的意思是设计一个特定的部分，这样用自定义实现方式可以替代它。

有效负载可以用于传输方法调用的参数，这个有效负载编码也可以替换。Microsoft 发布了 SOAP 和二进制编码机制。可以通过 HTTP 信道来使用 SOAP 格式化程序，也可以通过二进制格式化程序使用 HTTP。当然，这两种格式化程序都可以与 TCP 信道一起使用。

尽管 SOAP 可以和.NET Remoting 一起使用，但应该知道.NET Remoting 仅支持 SAOP RPC 样式，而 ASP.NET Web 服务支持 DOC 样式(默认)和 RPC 样式。RPC 样式在新的 SOAP 版本中已废弃。

通过.NET Remoting，不但可以在每一个.NET 应用程序中使用服务器功能，还可以在任何地方使用.NET Remoting，包括控制台应用程序、Windows 应用程序、Windows 服务或 COM+组件。.NET Remoting 还是用于对等通信的一种好技术。

54.2　.NET Remoting 术语详解

.NET Remoting 可以用于访问另一个应用程序域中的对象。不论两个对象是处于一个进程中，还是处于不同的进程中，甚至处于不同的系统中，都可以使用.NET Remoting。

远程程序集可以配置为在应用程序域本地工作，或者配置为远程应用程序的一部分。如果程序集是远程应用程序的一部分，则客户端收到一个代理而不是真实的对象进行会话。代理表示客户端进程中的远程对象，由客户端应用程序用于调用方法。当客户端在代理中调用方法时，代理把一条消息发送到信道中，该消息再传递给远程对象。

.NET 应用程序通常在应用程序域中工作。应用程序域可以看作进程中的子进程。传统上，进程通常用作隔离的边界。在一个进程中运行的应用程序不能访问和销毁另一个进程中的内存。对于相互通信的应用程序，需要跨进程的通信。利用.NET，应用程序域就成为进程中新的安全边界，原因是 MSIL 代码是类型安全和可验证的。如第 18 章所述，不同应用程序可以在同一进程内的不同应用程序域中运行。在同一应用程序域中的对象可以直接进行交互，但是在访问不同应用程序域中的对象时，必须使用代理。

下面列出了.NET Remoting 体系结构的主要元素：

- **远程对象**——远程对象是运行在服务器上的对象。客户端不能直接调用远程对象上的方法，而要使用代理。使用.NET，很容易把远程对象和本地对象区分开：即任何派生自 MarshalByRefObject 的类从来都不会离开它的应用程序域。客户端可以通过代理调用远程对象的方法。
- **信道**——信道用于客户端和服务器之间的通信。信道包括客户端的信道部分和服务器的信道部分。.NET Framework 4 提供了 3 种信道类型，它们分别通过 TCP、HTTP 和 IPC 进行通信。此外，还可以创建自定义信道，这些信道使用其他协议通信。
- **消息**——消息被发送到信道中。消息是为客户端和服务器之间的通信而创建的。消息包含远程对象的信息、被调用方法的名称以及所有的参数。
- **格式化程序**——格式化程序用于定义消息如何传输到信道中。.NET 4 有 SOAP 格式化程序和二进制格式化程序。使用 SOAP 格式化程序可以与不是基于.NET Framework 的 Web 服务通信。二进制格式化程序速度更快，可以有效地用在内部网环境中。当然，也可以创建自定义格式化程序。
- **格式化程序提供程序**——格式化程序提供程序用于把格式化程序与信道关联起来。通过创建信道，可以指定要使用的格式化程序提供程序，格式化程序提供程序则定义把数据传输到信道中时所使用的格式化程序。
- **代理**——客户端调用代理的方法，而不是远程对象的方法。代理分为两种：透明的代理和真实的代理。对于客户端，透明代理看起来与远程对象类似。在透明代理上，客户端可以

调用远程对象实现的方法。然后，透明代理调用真实代理上的 Invoke()方法。Invoke()方法使用消息接收器把消息传递给信道。

- **消息接收器**——消息接收器是一个侦听器(interceptor)对象，简称接收器。在客户端和服务器上都有侦听器。接收器与信道相关联。真实的代理使用消息接收器把消息传递到信道中，因此，在消息进入信道之前，接收器可以进行截获工作。根据接收器所处的位置，可以把接收器称为特使接收器(envoy sink)、服务器上下文接收器、对象上下文接收器等。
- **激活器**——客户端可以使用激活器在服务器上创建远程对象，或者获取一个被服务器激活的对象的代理。
- RemotingConfiguration 类——该类是用于配置远程服务器和客户端的一个实用程序类。它可以用于读取配置文件或动态地配置远程对象。
- ChannelServices 类——该类是一个实用程序类，可用于注册信道并把消息分配到信道中。

图 54-1 展示了如何把各个元素联系在一起。

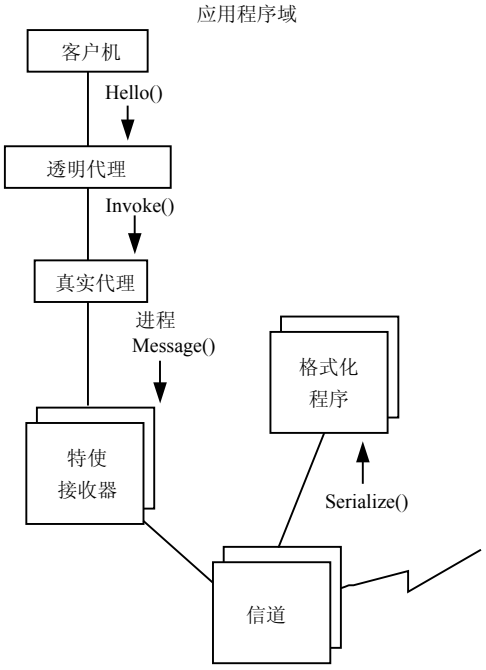

图 54-1

54.2.1 客户端通信

在客户端调用远程对象上的方法时，它实际上调用的是透明代理(而不是实际对象)上的方法。透明代理看起来与真实对象一样，它可以实现真实对象的公有方法。透明代理使用反射机制从程序集中读取元数据，获得真实对象的公共方法的信息。

然后，透明对象调用真实代理。真实代理负责把消息发送到信道中。真实代理是“可插入的”，可以使用自定义实现方式取代它。自定义实现方式可以用于写日志，或使用另一种方法查找信道等。真实代理对象的默认实现方式是查找特使接收器的集合(或链)，并把消息传递给第一个特使接收器。特使接收器可以截获和更改消息。这种接收器的示例有调试接收器、安全接收器以及同步接收器等。

最后一个特使接收器把消息发送到信道中。如何在线上发送消息取决于格式化程序。如前所述，.NET Framework 4 提供了 SOAP 格式化程序和二进制格式化程序。格式化程序也是“可插入的”。信道负责连接到服务器的监听网络接口上，或者发送已格式化的数据。定制信道的功能可以与上述功能不同，仅需执行代码，就可以完成把数据传输到另一端的必要工作。

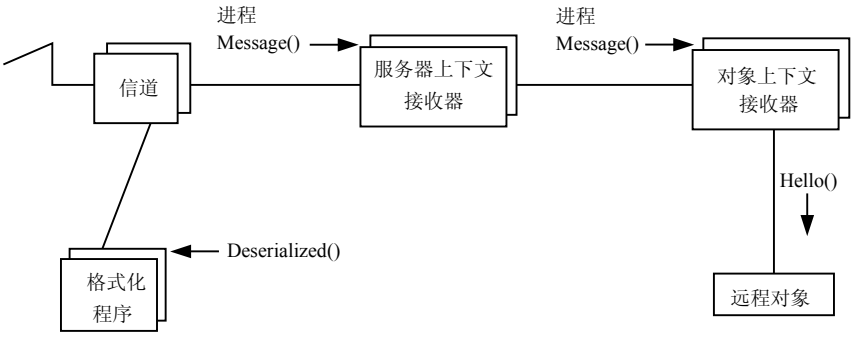

图　54-2

54.2.2　服务器端通信

图 54-2 展示了服务器端的消息传递：

- 信道接收来自客户端的已格式化消息，并且使用格式化程序打乱消息中的 SOAP 或二进制数据。然后，信道调用服务器上下文接收器。
- 服务器上下文接收器是一个接收器链，链中的最后一个接收器继续调用对象上下文接收器链。
- 最后一个对象上下文接收器调用远程对象上的方法。

注意对象上下文接收器被限制为对象上下文；服务器上下文接收器被限制为服务器上下文。单个的服务器上下文接收器可以用于访问许多对象接收器。

日志接收器就是一个接收器的例子。利用日志接收器，可以记录发送和接收的消息。利用客户系统上的接收器，可以检查服务器是否可用，并动态修改另一个服务器。

.NET Remoting 具有极大的可定制性：可以替代真实代理、添加接收器对象、替代格式化程序和信道等。当然，也可以使用所有已有的对象。

或许我们想知道通过这些层时的系统开销，如果什么工作也没有做，就不会有太多的系统开销；如果添加了一些功能，系统开销就取决于所添加的功能。

54.3　上下文

.NET Remoting 可以用于建立通过网络进行通信的服务器和客户端，在讨论这个方面的内容之前，首先看看在应用程序域中何时需要信道：通过上下文调用对象。

如果您编写过 COM+组件，则一定知道 COM+上下文。.NET 中的上下文与它非常相似。上下

文是包含一个对象的边界。同样，在 COM+上下文中，这种集合中的对象需要上下文特性定义的相同用法规则。

我们知道，一个进程可以有多个应用程序域。应用程序域类似于带有安全边界的子进程。第 18 章讨论过应用程序域。

应用程序域可以有不同的上下文。上下文用于把具有相似执行需求的对象组合在一起。上下文由一组属性组合而成，它用于截获工作：即从不同的上下文中访问上下文绑定对象(context-bound object)时，侦听器可以在调用到达对象之前进行截获工作，如线程同步、事务和安全管理。

派生自 MarshalByRefObjects 的类被绑定在应用程序域中。在应用程序域外，访问对象时需要代理。派生自 ContextBoundObject(它又派生自 MarshalByRefObjects)的类被限制在上下文中。在上下文之外，访问对象时需要代理。

上下文绑定对象可以拥有上下文特性。没有上下文特性的上下文绑定对象在创建者的上下文中创建。带有上下文特性的上下文绑定对象在新的上下文中创建。如果属性兼容，那么也可以在创建者的上下文中创建带有上下文特性的上下文绑定对象。

为了更好地理解上下文，必须知道下面的术语：

- 默认上下文——在创建应用程序域的同时，会在该应用程序域中创建默认上下文。如果实例化的新对象需要不同的上下文特性，则系统会为它创建新的上下文。
- 上下文特性——上下文特性可以赋予派生自 ContextBoundObject 的类。实现 IContextAttribute 接口，可以创建自定义特性类。.NET Framework 在 System.Runtime.Remoting.Contexts 名称空间中有一个上下文特性类 SynchronizationAttribute。
- 上下文属性——上下文特性定义对象需要的上下文属性。上下文属性类实现 IContextProperty 接口。活动属性为主调链提供消息接收器。ContextAttribute 类实现 IContextProperty 和 IcontextAttribute 接口，可以用作自定义属性的基类。
- 消息接收器——消息接收器是方法调用的侦听器。使用消息接收器，可以截获方法调用。特性可以提供消息接收器。

54.3.1 激活

如果创建的类实例需要不同于主调上下文的上下文，系统就会创建新的上下文。如果当前上下文的所有属性都可以接受，就会请求与目标类相关联的特性类。如果这些属性中的任何一个不能接受，则运行库就请求所有与该特性类相关联的属性类，并且创建一个新的上下文。然后，运行库为要安装的接收器请求属性类。属性类可以实现一个 IContributeXXXSink 接口，以提供接收器对象。其中的几个接口可以用于不同的接收器。

54.3.2 特性和属性

通过上下文特性可以定义上下文的属性。上下文特性类主要是特性。第 14 章有特性的详细介绍。上下文特性类必须实现 IContextAttribute 接口。因为 ContextAttribute 类总是有 IContextAttribute 接口的一个默认实现方式，所以自定义上下文特性类可以派生自 ContextAttribute 类。

.NET Framework 有两个上下文特性类：System.Runtime.Remoting.Contexts.SynchronizationAttribute 和 System.Runtime.Remoting.Activation.UrlAttribute。Synchronization 特性用于定义同步的要求，它指定对象需要的同步属性。可以指定多个线程不能同时访问该对象，但是访问对象的线程可以更改。

使用这个特性的构造函数，可以设置以下 4 个值中的 1 个值：

- NOT_SUPPORTED，定义不应该在设置了同步的上下文中对类进行实例化。
- REQUIRED，指定需要有同步上下文。
- REQUIRED_NEW，总是创建一个新的上下文。
- SUPPORTED，表示得到什么样的上下文并不重要，只要上下文中有对象即可。

54.3.3　上下文之间的通信

上下文之间是怎样进行通信的呢？客户端使用代理代替真实对象，代理创建的消息要传输到信道中，接收器可以截获。相同的机制也可以用在跨不同应用程序域或不同系统的通信中。跨上下文的通信不需要 TCP 或 HTTP 信道，但这里也使用了信道。CrossContextChannel 可以在信道的客户端和服务器端使用同一虚拟内存。此外，跨上下文的通信也不需要格式化程序。

54.4　远程对象、客户端和服务器

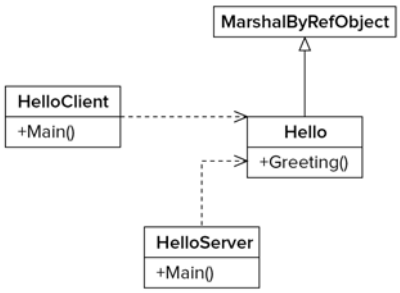

图　54-3

在详细讨论.NET Remoting 体系结构之前，本节先简要地讨论远程对象和一个非常简单的客户端/服务器应用程序，该应用程序使用远程对象。之后，详细讨论所有需要的步骤和选项。

图 54-3 显示了客户端和服务器应用程序中的主要.NET Remoting 类。实现的远程对象是 Hello。HelloServer 是服务器上应用程序的主类，HelloClient 是客户端上应用程序的主类。

54.4.1　远程对象

分布式计算需要远程对象。从不同系统中远程调用的对象必须派生自 System.MarshalByRefObject 对象。MarshalByRefObject 对象被限制在创建它们的应用程序域中，这说明不能跨应用程序域传递它们；而应使用代理对象访问另一个应用程序域中的远程对象。其他应用程序域可以存在于同一进程、另一个进程或另一个系统中。

远程对象具有分布式标识。因此，对象的引用可以传递给其他客户端，而其他客户端仍访问同一对象。代理知道远程对象的标识。

除了具有从 Object 类继承的方法之外，MarshalByRefObject 类还有用于初始化和获取生命周期服务的方法。生命周期服务定义远程对象的生命周期有多长。本章后面的内容将讨论生命周期服务和租约功能。

为了查看起作用的.NET Remoting，下面给远程对象创建一个简单的类库。Hello 类派生自 System.MarshalByRefObject。在构造函数中，把消息写入控制台中，提供对象的生命周期信息。此外，添加一个从客户端调用的 Greeting()方法。

为了易于区分后面几节中的程序集和类，我们在所使用的方法调用的参数中给它们指定不同的名称。程序集的名称是 RemoteHello，类的名称是 Hello。


可从
wrox.com
下载源代码

```
using System;

namespace Wrox.ProCSharp.Remoting
{
   public class Hello : System.MarshalByRefObject
   {
      public Hello()
      {
         Console.WriteLine("Constructor called");
      }

      public string Greeting(string name)
      {
         Console.WriteLine("Greeting called");
         return "Hello, " + name;
      }
   }
}
```

代码段 RemoteHello/RemoteHello/Hello.cs

54.4.2 简单的服务器应用程序

对于服务器，创建一个新的 C#控制台应用程序 HelloServer。为了使用 TcpServerChannel 类，必须引用 System.Runtime.Remoting 程序集。此外，还需要引用上一节创建的 RemoteHello 程序集。

在 Main()方法中，用端口号 8086 创建一个 System.Runtime.Remoting.Channels.Tcp.TcpServerChannel 类型的对象。该信道使用 System.Runtime.Remoting.Channels.ChannelServices 类注册，使之可用于远程对象。远程对象类型通过调用 RemotingConfiguration. RegisterWellKnownServiceType()方法注册。

在这个例子中，指定客户端所使用的远程对象的 URI 类型和一种模式。WellKnownObject. SingleCall 模式说明为每个方法调用创建一个新的实例，示例程序不保存远程对象中的状态。

.NET Remoting 允许创建无状态和有状态的远程对象。在第一个例子中，我们使用了已知的单调用(single-call)对象，它不保存状态。另一个对象类型称为客户端激活的对象，这种对象保存状态。本章后面在介绍对象的激活顺序时，将详细论述它们的区别，以及如何使用这些对象类型。

在远程对象注册之后，有必要使服务器一直处于运行状态，直到按任意键为止：


可从
wrox.com
下载源代码

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

namespace Wrox.ProCSharp.Remoting
{
   class Program
   {
      static void Main()
      {
         var channel = new TcpServerChannel(8086);
```

```
            ChannelServices.RegisterChannel(channel, true);
            RemotingConfiguration.RegisterWellKnownServiceType(
                 typeof(Hello), "Hi", WellKnownObjectMode.SingleCall);
            Console.WriteLine("Press return to exit");
            Console.ReadLine();
        }
    }
}
```

代码段 RemoteHello /HelloServer/Program.cs

54.4.3　简单的客户端应用程序

这个客户端应用程序也是一个 C#控制台应用程序 HelloClient。在该项目中，也引用 System.Runtime. Remoting 程序集，以便可以使用 TcpClientChannel 类。此外，也必须引用 RemoteHello 程序集。尽管将在远程服务器上创建对象，但是为了代理能在运行期间读取类型信息，还需要在客户端上引用程序集。

在客户端程序中，要创建一个 TcpClientChannel 对象，这个对象在 ChannelServices 中注册。对于 TcpChannel，可以使用默认的构造函数，因此可以选择任意一个端口。接下来，使用 Activator 类把代理返回远程对象。代码是 System.Runtime. Remoting.Proxies.__TransparentProxy 类型。这个对象看起来像是真实对象，原因是它提供相同的方法。透明代理使用真实代理把消息发送给信道：


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

namespace Wrox.ProCSharp.Remoting
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Press return after the server is started");
            Console.ReadLine();

            ChannelServices.RegisterChannel(new TcpClientChannel(), true);
            Hello obj = (Hello)Activator.GetObject(
                                        typeof(Hello), "tcp://localhost:8086/Hi");
            if (obj == null)
            {
                Console.WriteLine("could not locate server");
                return;
            }
            for (int i=0; i < 5; i++)
            {
                Console.WriteLine(obj.Greeting("Stephanie"));
            }
        }
    }
}
```

代码段 RemoteHello/HelloClient/Program.cs

代理是客户端应用程序用于替代远程对象的对象。第 55 章使用的代理与本章的代理有类似的功能。Web 服务的代理和.NET Remoting 的代码在实现方式上有很大的区别。

现在可以启动服务器和客户端。在客户端控制台中，Hello Stephanie 文本会出现 5 次。在服务器控制台窗口中，会看到如下所示的输出结果。因为选择的是 WellKnownObjectMode.SingleCall 激活模式，所以为每一个方法调用创建一个新的实例。

```
Press return to exit
Constructor called
Greeting called
Constructor called
Greeting called
Constructor called
Greeting called
Constructor called
Greeting called
Constructor called
Greeting called
```

54.5 .NET Remoting 体系结构

既然讨论了简单的客户端/服务器后，本节就介绍.NET 的体系结构并详细讨论它。以前面创建的程序为基础，下面详细介绍.NET 体系结构以及它的扩展机制。

本节将论述下面的主题：

- 信道的功能和配置
- 格式化程序及其用法
- 实用程序类 ChannelServices 和 RemotingConfiguration
- 激活远程对象的不同方式，以及如何在.NET Remoting 中使用无状态和有状态的对象
- 消息接收器的功能
- 如何按值和按引用传递对象
- 用.NET Remoting 租约机制管理有状态的对象的生命周期

54.5.1 信道

信道用于.NET 客户端和服务器之间的通信。.NET Framework 4 发布的信道类使用 TCP 、HTTP 或 IPC 进行通信。我们可以为其他的协议创建自定义信道。

HTTP 信道使用 HTTP 协议进行通信。因为防火墙通常让端口 80 处于打开的状态，所以客户端能够访问 Web 服务器，因为.NET Remoting Web 服务可以侦听端口 80，所以客户端更容易使用它们。

虽然在 Internet 上也可以使用 TCP 信道，但是必须配置防火墙，这样客户端能够访问 TCP 信道所使用的指定端口。与 HTTP 信道相比，在内部网环境中使用 TCP 信道能够进行更加高效的通信。

IPC 信道最适合于在单个系统上进行跨进程的通信。因为它使用 Windows 进程间通信机制，所以它比其他信道快。

当执行远程对象上的方法调用时，导致客户信道对象就把消息发送到远程信道对象中。

服务器应用程序和客户端应用程序都必须创建信道。下面的代码说明了如何在服务器端创建 TcpServerChannel：

```
using System.Runtime.Remoting.Channels.Tcp;
  ...
TcpServerChannel channel = new TcpServerChannel(8086);
```

构造函数的参数指定 TCP 套接字侦听哪个端口。服务器信道必须指定一个众所周知的端口，在访问服务器时，客户端必须使用该端口。但是，在客户端上创建 TcpClientChannel 时，不必指定一个众所周知的端口，TcpClientChannel 的默认构造函数会选择一个可用端口，在客户端与服务器连接时，该端口被传递给服务器，以便服务器能够把数据返回给客户端。

创建新的信道实例，会使套接字立即转换到侦听状态，在命令行中输入 netstat –a，可以验证套接字是否处于侦听状态。

测试跨网络发送了哪些数据的一种有效工具是 tcpTrace。它可以从 http://www.pocketsoap.com/tcptrace 上下载。

HTTP 信道的使用方式类似于 TCP 信道。可以指定服务器能在哪个端口上创建侦听套接字。

服务器可以侦听多个信道。下面的代码创建并注册 HTTP、TCP 和 IPC 信道：

```
var tcpChannel = new TcpServerChannel(8086);
var httpChannel = new HttpServerChannel(8085);
var ipcChannel = new IpcServerChannel("myIPCPort");

// register the channels
ChannelServices.RegisterChannel(tcpChannel, true);
ChannelServices.RegisterChannel(httpChannel, false);
ChannelServices.RegisterChannel(ipcChannel, true);
```

信道类必须实现 IChannel 接口。IChannel 接口有以下两个属性：

- ChannelName 属性是只读的，它返回信道的名称。信道的名称取决于协议的类型，例如，HTTP 信道的名称为 HTTP。
- ChannelPriority 属性也是只读的。在客户端和服务器之间可以使用多个信道进行通信，优先级定义信道的次序。在客户端上，具有较高优先级的信道首先连接到服务器上。优先级值越高，优先级就越高，其默认值是 1，但允许使用负值创建较低的优先级。

实现的其他接口要根据信道的类型决定，服务器信道实现 IChannelReceiver 接口，而客户端信道实现 IChannelSender 接口。

HTTPChannel、TcpChannel 和 IPCChannel 类都可以用于服务器和客户端。它们实现 IChannelSender 和 IChannelReceiver 接口。这些接口都派生自 IChannel 接口。

客户端的 IchannelSender 接口除了有 Ichannel 接口之外，还有一个 CreateMessageSink()方法，这个方法返回一个实现 IMessageSink 接口的对象。IMessageSink 接口可以把同步和异步消息放到信道中。在使用服务器端的接口 IChannelReceiver 时，通过 StartListening()方法可以把信道设置为侦听状态，而通过 StopListening()方法则可以停止对信道的侦听。ChannelData 属性用于访问所获取的数据。

使用信道类的属性，可以获取信道的配置信息。HTTP 和 TCP 信道都有 ChannelName、ChannelPriority 和 ChannelData 属性。使用 ChannelData 属性可以获取存储在 ChannelDataStore 类中的 URI 信息。此外，HttpServerChannel 类还有一个 Scheme 属性。下面的代码显示一个辅助方法 ShowChannelProperties()，该方法在文件中显示对应的信息：

```
static void ShowChannelProperties(IChannelReceiver channel)
{
   Console.WriteLine("Name: {0}", channel.ChannelName);
   Console.WriteLine("Priority: {0}", channel.ChannelPriority);
   if (channel is HttpServerChannel)
   {
      HttpServerChannel httpChannel = channel as HttpServerChannel;
      Console.WriteLine("Scheme: {0}", httpChannel.ChannelScheme);
   }
   if (channel is TcpServerChannel)
   {
      TcpServerChannel tcpChannel = channel as TcpServerChannel;
      Console.WriteLine("Is secured: {0}", tcpChannel.IsSecured);
   }

   ChannelDataStore data = (ChannelDataStore)channel.ChannelData;
   if (data != null)
   {
      foreach (string uri in data.ChannelUris)
      {
         Console.WriteLine("URI: {0}", uri);
      }
   }
   Console.WriteLine();
}
```

1. 设置信道属性

使用构造函数 TCPServerChannel(IDictionary, IServerChannelSinkProvider)，可以在一个列表中设置信道的所有属性。Dictionary 泛型类实现 IDictionary 接口，因此，使用这个类可以设置 Name、Priority 和 Port 属性。为了使用 Dictionary 类，必须导入 System.Collections.Generic 名称空间。

在 TcpServerChannel 类的构造函数中，除了参数 IDictionary 之外，还可以传递实现 IServerChannelSinkProvider 接口的对象。在本例中，设置 BinaryServerFormatterSinkProvider 而不设置 SoapServerFormatterSinkProvider，前者是 HttpServerChannel 的默认值。BinaryServerFormatterSinkProvider 类的默认实现代码把 BinaryServerFormatterSink 类与使用 BinaryFormatter 对象的信道联系起来，以转换数据，便于传输：

```
var properties = new Dictionary<string, string>();
properties["name"] = "HTTP Channel with a Binary Formatter";
properties["priority"] = "15";
properties["port"] = "8085";
var sinkProvider = new BinaryServerFormatterSinkProvider();
var httpChannel = new HttpServerChannel(properties, sinkProvider);
```

根据信道的类型，可以指定不同的属性。TCP 和 HTTP 信道都支持本例中使用的 name 和 priority 信道属性。这些信道也支持其他属性，如 bindTo，如果计算机配置了多个 IP 地址，bindTo 指定绑

定可以使用的 IP 地址。TCP 服务器信道支持 rejectRemoteRequests，只允许从本地计算机上连接客户端。

2. 信道的“可插入性”

创建的自定义信道可以使用 HTTP、TCP 和 IPC 之外的其他传输协议发送消息。此外，也可以对现有的信道进行扩展，从而提供更多功能：

- 发送部分必须实现 IChannelSender 接口。最重要的部分是 CreateMessageSink()方法，在该方法中，客户端要发送 URL，此外，使用这个方法可以实例化与服务器的连接。在这里必须创建消息接收器，代理使用该消息接收器把消息发送到信道中。
- 接收部分必须实现 IChannelReceiver 接口。必须在 ChannelData 的 get 属性中启动侦听功能。然后，可以等待另一个线程接收来自客户端的数据。在打乱消息之后，使用 ChannelServices.SyncDispatchMessage()方法把消息分配给对象。

54.5.2　格式化程序

.NET Framework 提供了两个格式化程序类，即：

- System.Runtime.Serialization.Formatters.Binary.BinaryFormatter
- System.Runtime.Serialization.Formatters.Soap.SoapFormatter

通过格式化程序接收器对象和格式化程序接收器提供程序，可以把格式化程序和信道联系起来。

这两个格式化程序类都实现 System.Runtime.Remoting.Messaging.IRemotingFormatter 接口，该接口定义 Serialize()方法和 Deserialize()方法，以便在信道之间来回传输数据。

格式化程序也是“可插入的”。在编写自定义格式化程序类时，必须把实例与要使用的信道联系起来，这项工作使用格式化程序接收器和格式化程序接收器提供程序就可以完成。例如，在创建前面介绍的信道时，格式化程序接收器提供程序 SoapServerFormatterSinkProvider 可以作为参数传递。格式化程序接收器提供程序为服务器实现 IServerChannelSinkProvider 接口，为客户端实现 IClientChannelSinkProvider 接口。这两个接口都定义 CreateSink()方法，这个方法必须返回格式化程序接收器。SoapServerFormatterSinkProvider 返回 SoapServerFormatterSink 类的一个实例。

在客户端，SoapClientFormatterSink 类使用 SoapFormatter 类的 SyncProcessMessage()方法和 AsyncProcessMessage()方法，序列化消息。而 SoapServerFormatterSink 类使用 SoapFormatter 类反序列化消息。

所有这些接收器和提供程序类都可以扩展，并可以被自定义实现方式替代。

54.5.3　ChannelServices 和 RemotingConfiguration

ChannelServices 实用程序类用于把信道注册到.NET Remoting 运行库中。此外，也可以使用这个类访问所有已注册的信道。因为在这里信道是隐式创建的(详见后面的内容)，所以在使用配置文件配置信道时，ChannelServices 类极其有用。

使用静态方法 ChannelServices.RegisterChannel()可以注册信道。这个方法的第一个参数需要该信道，把第二个参数设置为 true，可以验证安全性对于信道是否可用。因为 HttpChannel 类没有安全支持，所以把对应的检查设置为 false。

下面是注册 HTTP、TCP 和 IPC 信道的服务器代码：

```
var tcpChannel = new TcpChannel(8086);
var httpChannel = new HttpChannel(8085);
var ipcChannel = new IpcChannel("myIPCPort");
ChannelServices.RegisterChannel(tcpChannel, true);
ChannelServices.RegisterChannel(httpChannel, false);
ChannelServices.RegisterChannel(ipcChannel, true);
```

ChannelServices 实用程序类可以用于分配同步消息和异步消息，以及注销指定信道。RegisteredChannels 属性返回一个 IChannel 数组，数组中的元素是已注册的所有信道。此外，还可以使用 GetChannel()方法根据名称获取指定的信道。使用 ChannelServices 类，可以编写自定义管理实用程序，用于管理信道。下面的小示例阐明了如何阻止服务器信道侦听所传入的请求：

```
HttpServerChannel channel =
(HttpServerChannel)ChannelServices.GetChannel("http");
channel.StopListening(null);
```

RemotingConfiguration 类是另一个.NET Remoting 实用程序类。在服务器端，这个类用于为服务器激活的对象注册远程对象类型，把远程对象编组到已编组的对象引用类 ObjRef 中。ObjRef 是在网络上发送的对象的可序列化表示。在客户端，为了从对象引用中创建代理，需要使用 RemotingServices 类打乱远程对象。

1. 知名对象的服务器

下面的服务器端代码把知名的远程对象类型注册为 RemotingServices：

```
RemotingConfiguration.RegisterWellKnownServiceType(
                              typeof(Hello),                    // Type
                              "Hi",                             // URI
                              WellKnownObjectMode.SingleCall); // Mode
```

RegisterWellKnownServiceType()方法的第一个参数是 typeof(Hello)，它指定远程对象的类型。第二个参数 Hi 是远程对象的 URI，客户端访问远程对象时要使用这个 URI。最后一个参数是远程对象的模式。模式可以是 WellKnownObjectMode 枚举的一个值：SingleCall 或 Singleton。

- SingleCall 意味着对象不保存状态。每一次调用远程对象时，都会创建一个新的实例。使用 RemotingConfiguration.RegisterWellKnownServiceType()方法和 WellKnownObjectMode.SingleCall 参数，可以从服务器中创建 SingleCall 对象。因为不需要为数千个客户端保存资源，因此，这种模式在服务器上非常高效。
- 使用 singleton，服务器的所有客户端都可以共享对象。一般情况下，如果要在所有的客户端之间共享一些数据，则可以使用这种对象类型。对于只读数据这是毫无问题的，但是，对于可读写的数据，就必须考虑数据锁定问题和可伸缩性。使用 RemotingConfiguration. RegisterWellKnownServiceType()方法和 WellKnownObjectMode. Singleton 参数，服务器可以创建单一对象。必须考虑单一对象所使用资源的锁定问题。在客户端并发地访问单一对象时，必须确保数据不能被损坏，还必须检查锁定是否足够有效，以便实现必要的可伸缩性。

2. 客户端激活的对象的服务器

如果远程对象应该保存某一特定客户端的状态，则可以使用客户端激活的对象。下一节将讨论

在客户端如何调用服务器激活的对象或客户端激活的对象。在服务器端，客户端激活的对象的注册方式必须与服务器激活的对象不同。

不调用 RemotingConfiguration.RegisterWellKnownType()方法，而必须调用 Remoting Configuration. RegisterActivatedServiceType()方法。使用这个方法时，只指定类型，不指定 URI。原因是，对于客户端激活的对象，客户端可以使用同一个 URI 对不同的对象类型进行实例化。所有客户激活的对象的 URI 必须使用 RemotingConfiguration.ApplicationName 定义：

```
RemotingConfiguration.ApplicationName = "HelloServer";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Hello));
```

54.5.4　对象的激活

客户端可以使用和创建远程 Activator 类。使用 GetObject()方法，可以得到服务器激活的远程对象或知名的远程对象的代理。CreateInstance()方法返回客户端激活的远程对象的代理。

new 运算符可以代替 Activator 类激活远程对象。为此，还必须使用 RemotingConfiguration 类在客户端中配置远程对象。

1. 应用程序的 URL

在激活对象时，必须指定远程对象的 URL。这个 URL 与使用 Web 浏览器进行浏览时所使用的 URL 相同。第一部分指定协议、服务器名或 IP 地址、端口号和 URI，其中 URI 在服务器中以下面的格式注册远程对象时指定：

```
protocol://server:port/URI
```

下面的代码示例连续使用 3 个 URL 示例。在 URL 中，用 http、tcp 和 ipc 指定协议，对于 HTTP 和 TCP 信道，服务器名是 localhost，端口号是 8085 和 8086。对于 IPC 信道，不需要定义主机名，因为 IPC 只能在单个系统上使用。对于所有协议，URI 是 Hi，如下所示：

```
http://localhost:8085/Hi
tcp://localhost:8086/Hi
ipc://myIPCPort/Hi
```

2. 激活知名对象

在 54.4.3 节的简单客户端例子中，激活了知名对象。下面详细讨论一下激活顺序：

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

Hello obj = (Hello)Activator.GetObject(typeof(Hello),
                                       "tcp://localhost:8086/Hi");
```

GetObject()是 System.Activator 类的一个静态方法，它调用 Remoting Services.Connect()方法以返回远程对象的代理对象。GetObject()方法的第一个参数指定远程对象的类型。代理实现所有公有的和受保护的方法和属性，以便客户端可以像在真实对象上那样调用这些方法。第二个参数是远程对象的 URL。这里使用 tcp://localhost:8086/Hi 字符串，其中 tcp 定义使用的协议，localhost:8086 是主机名和端口号，最后的 Hi 是对象的 URI，其中对象在服务器上使用 method.RemotingConfiguration.RegisterWellKnownServiceType()方法

指定。

也可以直接使用 RemotingServices.Connect()方法，而不使用 Activator.GetObject()方法：

```
Hello obj = (Hello)RemotingServices.Connect(typeof(Hello),
                                            "tcp://localhost:8086/Hi");
```

如果偏爱使用 new 运算符激活知名的远程对象，则可以在客户端上使用 RemotingConfiguration.RegisterWellKnownClientType()方法注册远程对象。这里需要的参数与上面相似：即远程对象的类型和 URI。new 运算符实际上并没有创建新的远程对象，它返回一个与 Activator.GetObject()方法相似的代理。如果使用 WellKnownObjectMode.SingleCall 标记注册远程对象，那么规则总是一样：使用每一个方法调用创建远程对象。

```
RemotingConfiguration.RegisterWellKnownClientType(typeof(Hello),
"tcp://localhost:8086/Hi");
Hello obj = new Hello();
```

3. 激活客户端激活的对象

远程对象可以保存客户端的状态。Activator.CreateInstance()方法用于创建客户端激活的远程对象。使用 Activator.GetObject()方法，可以在调用方法时创建远程对象，而当方法执行完时，可以销毁远程对象。对象不在服务器上保存状态，这一点与 Activator.CreateInstance()方法不同。使用静态的 CreateInstance()方法，按次序开始激活，进而创建远程对象。在租约时间到期并且进行垃圾收集之前，对象将一直处于激活状态。本章后面的内容将讨论租约机制。

一些重载的 Activator.CreateInstance()方法只能用于创建本地对象。为了创建远程的对象，就需要一个能够传递激活属性的方法。下面的示例就使用其中一个重载的方法，该方法接收两个字符串参数和一个对象数组，第一个参数是程序集的名称，第二个参数是远程对象的类型。利用第三个参数，可以把参数传递给远程对象类的构造函数。通过 UrlAttribute 在对象数组中指定信道和对象名。为了使用 UrlAttribute 类，必须导入 System.Runtime.Remoting.Activation 名称空间：

```
object[] attrs = {new UrlAttribute("tcp://localhost:8086/HelloServer") };
ObjectHandle handle = Activator.CreateInstance(
             "RemoteHello", "Wrox.ProCSharp.Remoting.Hello", attrs);
if (handle == null)
{
   Console.WriteLine("could not locate server");
   return;
}

Hello obj = (Hello)handle.Unwrap();
Console.WriteLine(obj.Greeting("Christian"));
```

当然，对于客户端激活的对象，虽然也可以使用运算符 new 替代 Activator 类，但是使用 new 运算符，就必须使用 RemotingConfiguration.RegisterActivatedClientType()方法注册客户端激活的对象。在客户端激活的对象体系结构中，new 运算符不但返回代理，也创建远程对象：

```
RemotingConfiguration.RegisterActivatedClientType(typeof(Hello),
                                        "tcp://localhost:8086/HelloServer");
Hello obj = new Hello();
```

4. 代理对象

Activator.GetObject()方法和 Activator.CreateInstance()方法都给客户端返回一个代理。实际上使用两个代理：即透明代理和真实代理。透明代理看起来像远程对象，它实现远程对象的所有公共方法。这些方法调用 RealProxy 的 Invoke()方法，传递包含待调用方法的消息。在消息接收器的帮助下，RealProxy 把消息发送到信道中。

使用 RemotingServices.IsTransparentProxy()方法，可以检查对象是否真是透明代理。还可以通过 RemotingServices.GetRealProxy()方法获取真实代理。使用 Vasula Studio 调试器，很容易得到真实代理的所有属性：

```
ChannelServices.RegisterChannel(new TCPChannel());
Hello obj = (Hello)Activator.GetObject(typeof(Hello),
                                       "tcp://localhost:8086/Hi");
if (obj == null)
{
   Console.WriteLine("could not locate server");
   return;
}
if (RemotingServices.IsTransparentProxy(obj))
{
   Console.WriteLine("Using a transparent proxy");
   RealProxy proxy = RemotingServices.GetRealProxy(obj);

   // proxy.Invoke(message);
}
```

5. 代理的“可插入性”

真实代理可以用自定义代理替代。自定义代理可以扩展基类 System.Runtime. Remoting.RealProxy。在自定义代理的构造函数中接收远程对象的类型。调用 RealProxy 的构造函数，可以创建真实代理和透明代理。在构造函数中，使用 ChannelServices 类创建消息接收器 IChannelSender.CreateMessageSink()，可以访问已注册信道。除了实现构造函数之外，自定义信道还必须重写 Invoke()方法。在 Invoke()方法中，可以接收到可分析的消息，然后把它们发送到消息接收器中。

6. 消息

代理可以把消息发送到信道中。在服务器端，分析消息之后，就可以进行方法调用。因此，下面讨论消息。

.NET Framework 有一些消息类可以用于方法调用、响应，以及返回消息等。所有消息类都可以实现 IMessage 接口，该接口只有一个 Properties 属性。Properties 属性表示一个带有 IDictionary 接口的字典，字典中包括对象的 URI、MethodName、MethodSignature、TypeName、Args 和 CallContext 等。

图 54-4 显示了消息类和接口的层次结构。发送给真实代理的消息是 MethodCall 类型的对象。通过 IMethodCallMessage 和 IMethodMessage 接口比通过 IMessage 接口更容易实现对消息属性的访问。不必使用 IDictionary 接口，仍可以直接访问方法名、URI 和参数等内容。真实代理把 ReturnMessage 返回给透明代理。

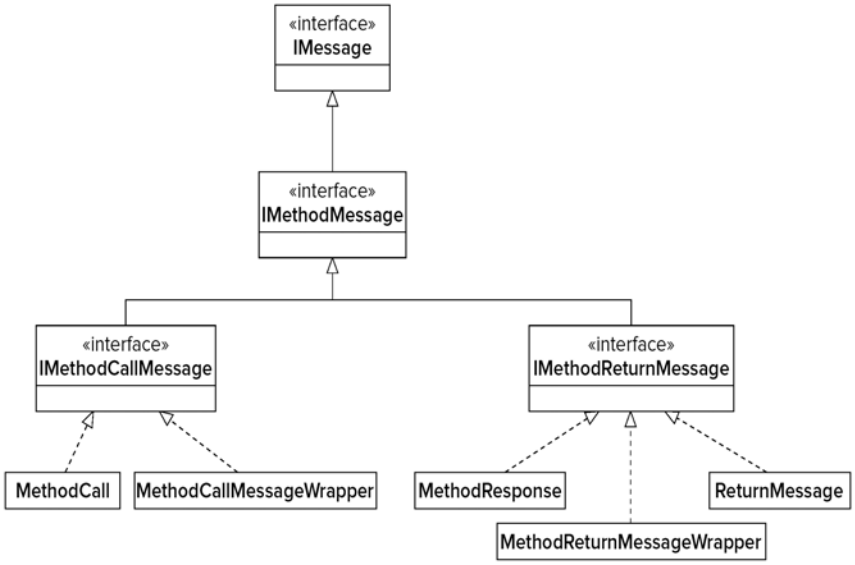

图 54-4

54.5.5 消息接收器

Activator.GetObject()方法调用 RemotingServices.Connect()方法连接已知对象。在 Connect()方法中，Unmarshal()方法不但在创建代理时发生，也在创建特使接收器时发生。代理使用一个特使接收器链把消息传递到信道中。所有接收器都是侦听器，它们可以更改消息，执行一些额外的操作，如创建锁、写事件以及执行安全检查等。

所有消息接收器都实现 IMessageSink 接口，这个接口定义一个属性和两个方法：

- NextSink 属性——接收器使用这个属性到达下一个接收器，并向前传递消息。
- SyncProcessMessage()方法——对于同步消息，前面的接收器或远程基础结构调用这个方法，它的 IMessage 参数用于发送消息和返回消息。
- AsyncProcessMessage()方法——对于异步消息，接收器链中前面的接收器或远程基础结构调用这个方法。该方法有两个参数：消息和接收回应的消息接收器。

下面几节讨论可以使用的 3 个不同的消息接收器。

1. 特使接收器

通过 IEnvoyInfo 接口，可以到达特使接收器链。编组对象引用 ObjRef 有一个 EnvoyInfo 属性，该属性返回 IEnvoyInfo 接口。特使列表从服务器上下文中创建，因此，服务器可以把一些功能注入客户端。特使可以收集客户端的身份信息，并把这些信息传递给服务器。

2. 服务器上下文接收器

在信道的服务器端接收消息时，消息就传递给服务器上下文接收器。服务器上下文接收器链中的最后一个接收器把消息传递到对象接收器链中。

3. 对象接收器

对象接收器与某个具体的对象关联。如果对象类定义特定上下文特性，就为该对象创建上下文接收器。

54.5.6　在远程方法中传递对象

远程方法调用中的参数类型不仅可以是基本的数据类型，还可以是我们自己定义的类。为了进行远程处理，必须区分下面 3 种类型的类：

- **按值编组的类**——这种类通过信道进行序列化。要编组的类必须用 Serializable 特性标记。这些类的对象没有远程标识，因为完整的对象通过信道编组，而且与客户端序列化的对象独立于服务器对象(或相反)。按值编组的类也称作未绑定的类，原因是它们没有依赖于应用程序域的数据。序列化详见第 29 章。
- **按引用编组的类**——这种类有远程标识。对象不是在网络上传递的，而是返回一个代理。按引用编组的类必须派生自 MarshalByRefObject。MarshalByRefObjects 称为应用程序域绑定对象。MarshalByRefObject 的一个专业化版本是 ContextBoundObject：抽象类 ContextBoundObject 派生自 MarshalByRefObject。如果类派生自 ContextBoundObject，则当上下文边界交叉时，甚至在同一应用程序域中也需要代理。这样的对象称为上下文绑定对象，它们只在创建上下文中有效。
- **不能用于远程通信的类**——这种类不能序列化，也不派生自 MarshalByRefObject 的。这些类型的类不能在远程对象的公共方法中用作参数。它们只能用于创建它们的应用程序域中。如果类的数据成员只在应用程序域中有效(如 Win32 文件句柄)则应该使用这种类。

为了阐明类的编组问题，我们将把远程对象改为向客户端发送一个对象：MySerialized 类将按值编组。在方法中，消息被写入控制台中，以便验证调用是在客户端上进行还是在服务器上进行。此外，把 Hello 类扩展为返回 MySerialized 实例：


```
using System;

namespace Wrox.ProCSharp.Remoting
{
   [Serializable]
   public class MySerialized
   {
      public MySerialized(int val)
      {
         a = val;
      }
      public void Foo()
      {
         Console.WriteLine("MySerialized.Foo called");
      }
      public int A
      {
      get
      {
         Console.WriteLine("MySerialized.A called");
         return a;
```

```
        }
        set
        {
           a = value;
        }
      }
      protected int a;
   }

   public class Hello : MarshalByRefObject
   {
      public Hello()
      {
         Console.WriteLine("Constructor called");
      }
      public string Greeting(string name)
      {
         Console.WriteLine("Greeting called");
         return "Hello, " + name;
      }
      public MySerialized GetMySerialized()
      {
         return new MySerialized(4711);
      }
    }
}
```

代码段 PassingObjects/RemoteHello/Hello.cs

此外，为了查看使用按值编组的对象和按引用编组的对象的效果，还需要修改客户端应用程序。调用 GetMySerialized()方法和 GetMyRemote()方法获取新的对象。再使用 RemotingServices.IsTransparentProxy()方法检查返回的对象是否是代理。

可从 wrox.com 下载源代码

```
ChannelServices.RegisterChannel(new TcpClientChannel(), false);
Hello obj = (Hello)Activator.GetObject(typeof(Hello),
      "tcp://localhost:8086/Hi");
if (obj == null)
{
   Console.WriteLine("could not locate server");
   return;
}
MySerialized ser = obj.GetMySerialized();
if (!RemotingServices.IsTransparentProxy(ser))
{
   Console.WriteLine("ser is not a transparent proxy");
}
ser.Foo();
```

代码段 PassingObjects /HelloClient/Program.cs

在客户端控制台窗口中，可以看到在客户端上调用了 ser 对象。因为 ser 对象序列化到客户端，所以它不是透明代理。

```
Press return after the server is started
```

```
ser is not a transparent proxy
MySerialized.Foo called
```

1. 安全性和序列化的对象

.NET Remoting 和 ASP.NET Web 服务的一个重要区别是对象编组的方式。在 ASP.NET Web 服务中，只有公共字段和属性通过网络传输。而.NET Remoting 使用另一种序列化机制来序列化所有数据，包括所有私有数据。恶意客户端可以在序列化和反序列化阶段中破坏应用程序。

为了解决这个问题，跨.NET Remoting 边界传递对象时，定义两个自动反序列化级别：低级反序列化和完整反序列化。

在默认情况下，使用低级反序列化。在低级反序列化中，不能传递 ObjRef 对象，也不能传递实现 ISponsor 接口的对象。为了传递这两类对象，可以把反序列化级别改为完整级别。这可以通过编程方式实现：创建一个格式化程序接收器提供程序，并给它赋予 TypeFilterLevel 属性。对于二进制格式化程序，提供程序类是 BinaryServerFormatterSinkProvider，对于 SOAP 格式化程序，提供程序类是 SoapServerFormatterSinkProvider。

下面的代码说明了如何利用完整级别的序列化支持创建一个 TCP 信道：


可从 wrox.com 下载源代码

```
var serverProvider = new BinaryServerFormatterSinkProvider();
serverProvider.TypeFilterLevel = TypeFilterLevel.Full;

var clientProvider = new BinaryClientFormatterSinkProvider();

var properties = new Dictionary<string, string>();
properties["port"] = 6789;

var channel = new TcpChannel(properties, clientProvider, serverProvider);
```

代码段 PassingObjects/HelloServer/Program.cs

首先，创建一个 BinaryServerFormatterSinkProvider，并把 TypeFilterLevel 属性设置为 TypeFilterLevel.Full。因为 TypeFilterLevel 枚举在 System.Runtime.Serialization.Formatters 名称空间中定义，所以必须声明这个名称空间。对于信道的客户端，创建一个 BinaryClientFormatterSinkProvider。客户端和服务器端的格式化程序接收器提供程序实例都传递给 TcpChannel 类的构造函数，定义信道特性的 IDictionary 属性也传递给该构造函数。

2. 方向特性

远程对象从来都不通过网络传输，而值类型和可序列化的类通过网络传输。有时只需要在一个方向上发送数据。这在数据通过网络传输时尤其重要。例如，如果要把集合中的数据发送给服务器，服务器再对这些数据执行一些计算操作，并给客户端返回一个简单的值，把集合发送回客户端就不是很有效。如果数据应发送给服务器、客户端或双向发送，则可以使用 COM 给参数声明方向特性[in]、[out]和 [in, out]。

在 C#中，有相似的特性：ref 和 out 方法参数。ref 和 out 方法参数可以用于可序列化的值类型和引用类型。使用 ref 参数时，数据可以双向编组；使用 out 时，数据从服务器发送到客户端；不使用参数 ref 和 out 时，数据从客户端发送到服务器。

有关 ref 和 out 关键字的内容详见第 3 章。

54.5.7 生命周期管理

客户端和服务器怎样检测到另一端是否可用？此时，我们遇到的问题是什么呢？

对于客户端，答案比较简单。只要客户端调用远程对象上的方法，就会产生一个 System.Runtime.Remoting.RemotingException 类型的异常。此时，只需处理这个异常，完成一些必要的工作，如重试、写日志以及通知用户等。

对于服务器，服务器应何时检测客户端是否还在？即服务器何时可以清理为该客户端保存的资源？可以一直等待来自客户端的下一个方法调用，但该客户端可能再没有方法调用了。在 COM 领域中，DCOM 协议使用 ping 机制解决这个问题。客户端把 ping 和引用对象的信息发送给服务器。因为客户端在服务器上可能有几百个引用的对象，所以 ping 中的信息非常多。为了使这个机制更加有效，DCOM 不发送所有对象的所有信息，而只发送与上一个 ping 不同的信息。

虽然这个 ping 机制在 LAN 上非常有效，但它并不适用于可伸缩的解决方案。考虑到有成千上万的客户端向服务器发送 ping 信息，.NET Remoting 为生命周期管理提供了一个伸缩性更强的解决方案：即租约分布式垃圾收集器(Leasing Distributed Garbage Collector，LDGC)。

这个生命周期管理只对客户端激活的对象和知名的单一对象有效。因为单一对象不保存状态，所以在每个方法调用之后就可以销毁它们。客户端激活的对象保存状态，我们应该知道它们使用的资源。如果在应用程序域外部引用客户端激活的对象，就需要创建租约。租约有一个租约时间。当租约时间为 0 时，租约就已经到期，此时远程对象就会断开连接，最后由垃圾收集器回收。

1. 租约的续约

当租约到期之后，如果客户端还调用对象上的方法，就会抛出异常。如果有一个客户端，其中需要租约远程对象的时间超过了 300 秒(默认的租约时间)时，那么有以下 3 种方法进行续约：

- **隐式续约**——当客户端调用远程对象上的方法时，租约的隐式续约会自动进行。如果当前租约时间小于 RenewOnCallTime 的值，租约时间就设置为 RenewOnCallTime。
- **显式续约**——通过显式续约，客户端可以指定新的租约时间，这项工作可以通过 ILease 接口的 Renew()方法完成。通过调用透明代理的 GetLifetimeService()方法，就可以使用 ILease 接口。
- **发起租约**——这是第三种续约的方法。客户端可以创建一个实现 ISponsor 接口的发起者，并使用 ILease 接口的 Register()方法在租约服务中注册这个发起者。发起者定义租约延长的时间。当租约到期时，发起者就要求延长租约时间。如果要长期租约服务器上的远程对象，就可以使用这个发起租约机制。

2. 租约的配置值

可以配置下面的一些值：

- LeaseTime——它定义租约到期之前的时间。
- RenewOnCallTime——这个时间是租约在方法调用上设置的时间，它指的是续约时间，如果当前租约时间的值低于这个时间，就要进行续约。

- SponsorshipTimeout——如果 SponsorshipTimeout 中没有租约发起者，远程基础结构就会寻找下一个发起者。如果没有更多发起者，租约就到期。
- LeaseManagerPollTime——租约管理器隔一段时间就检查一次，查看有没有对象到期，LeaseManagerPollTime 定义这个时间间隔。

表 54-1 中列出了它们的默认值。

表　54-1

租 约 配 置	默 认 值 (秒)
LeaseTime	300
RenewOnCallTime	120
SponsorshipTimeout	120
LeaseManagerPollTime	10

3. 管理生命周期所使用的类

ClientSponsor 类实现 ISponsor 接口。在客户端可以使用它延长租约时间。使用 ILease 接口，可以获取租约的所有信息、所有租约属性，以及当前租约的时间和状态。通过 LeaseState 枚举类型指定状态。通过 LifetimeServices 实用程序类，可以为应用程序域中所有远程对象的租约设置或获取属性。

4. 获取租约信息示例

在这个小示例代码中，调用透明代理的 GetLifetimeService()方法访问租约信息。对于 ILease 接口，必须声明 System.Runtime.Remoting.Lifetime 名称空间。对于 UrlAttribute 类，必须导入 System.Runtime.Remoting.Activation 名称空间。

租约机制只能用于有状态的(客户端激活的和单一)对象。由于每次调用方法时都实例化单一调用对象，因此租约机制不适用于单一调用对象。为了通过服务器提供客户端激活的对象，可以把远程处理配置更改为调用 RegisterActivatedServiceType()方法：

```
RemotingConfiguration.ApplicationName = "Hello";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Hello));
```

在客户端应用程序中，远程对象的实例化也必须更改。在此，并不使用 Activator.GetOnject()方法，而是使用 Activator.CreateInstance()方法调用客户端激活的对象：

```
ChannelServices.RegisterChannel(new TcpClientChannel(), true);

object[] attrs = {new UrlAttribute("tcp://localhost:8086/Hi") };
Hello obj = (Hello)Activator.CreateInstance(typeof(Hello), null, attrs);
```

为了显示租约时间，可以调用代理对象中的 GetLifetimeService()方法使用返回的 ILease 接口：

```
ILease lease = (ILease)obj.GetLifetimeService();
if (lease != null)
{
   Console.WriteLine("Lease Configuration:");
   Console.WriteLine("InitialLeaseTime: {0}", lease.InitialLeaseTime);
   Console.WriteLine("RenewOnCallTime: {0}", lease.RenewOnCallTime);
```

```
        Console.WriteLine("SponsorshipTimeout: {0}", lease.SponsorshipTimeout);
        Console.WriteLine(lease.CurrentLeaseTime);
    }
```

5. 更改默认的租约配置

使用 System.Runtime.Remoting.Lifetime.LifetimeServices 实用程序类，服务器自身就可以为所有远程对象更改默认的租约配置：

```
LifetimeServices.LeaseTime = TimeSpan.FromMinutes(10);
LifetimeServices.RenewOnCallTime = TimeSpan.FromMinutes(2);
```

如果希望不同的默认生命周期依赖于远程对象的类型，则可以重写 MarshalByRefObject 基类的 InitializeLifetimeService()方法，更改远程对象的租约配置：

```
public class Hello : System.MarshalByRefObject
{
    public override Object InitializeLifetimeService()
    {
        ILease lease = (ILease)base.InitializeLifetimeService();
        lease.InitialLeaseTime = TimeSpan.FromMinutes(10);
        lease.RenewOnCallTime = TimeSpan.FromSeconds(40);
        return lease;
    }
```

使用后面讨论的配置文件，也可以完成生命周期服务的配置。

54.6 配置文件

使用配置文件，可以不用把信道和对象配置写入到源代码中。如果使用配置文件，就可以在不更改源代码的情况下，重新配置信道，添加另外的信道等。与.NET 平台上所有其他的配置文件一样，这个配置文件也使用 XML，还使用了第 18 章和第 21 章中的应用程序和配置文件。对于.NET Remoting，可以使用一些 XML 元素和特性配置信道和远程对象。远程配置文件的不同之处是这个配置不需要放在应用程序配置文件中，该文件可以有任意名称。为了简化构建过程，本章将在应用程序配置文件中编写远程处理配置，应用程序配置文件的名称应该是可执行文件的名称后面跟上.config 文化扩展名。

在下载的代码(从 www.wrox.com 网站)中，可以在客户端和服务器示例的根目录中找到配置文件示例：clientactivated.config 和 wellknown.config。客户端示例中还有一个配置文件 wellknownhttp.config，它指定了一个通向知名远程对象的 HTTP 信道。为了使用这些配置文件，必须对它们重命名，RemotingConfiguration.Configure()方法的参数使用这些新文件名，并把它们放到包含可执行文件的目录中。

下面就是一个配置文件的示例：


```
<configuration>
    <system.runtime.remoting>
        <application name="Hello">
            <service>
                <wellknown mode="SingleCall"
                    type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
```

```
                    objectUri="Hi" />
            </service>
            <channels>
               <channel ref="tcp" port="6791" />
               <channel ref="http" port="6792" />
               <channel ref="ipc" portName="myIPCPort" />
            </channels>
         </application>
      </system.runtime.remoting>
   </configuration>
```

代码段 ConfigurationFiles/HelloServer/app.config

<configuration>是所有.NET 配置文件的 XML 根元素。所有远程配置都可以在子元素<system.runtime.remoting>中找到。<application>是<system.runtime.remoting>的一个子元素。

以下列表描述<system.runtime.remoting>中部件的主要元素和特性：

- 对于<application>元素，使用这个元素的 name 特性，可以指定应用程序的名称。在服务器端，name 特性的值是服务器的名称；而在客户端，name 特性的值是客户端应用程序的名称。上面的示例是服务器的配置文件：<application name="Hello"> 把远程应用程序名称定义为 Hello，客户端访问远程对象时可以把 Hello 用作 URL 的一部分。
- 在服务器端，<service>元素用于指定远程对象的集合。这个元素有两个子元素<wellknown>和<activated>，用于指定远程对象(知名对象或客户端激活的对象)的类型。
- <service>元素的客户端部分是<client>。与<service>元素类似，<client>元素也有两个子元素<wellknown>和<activated>，用于指定远程对象的类型。但是与<service>元素的两个子元素不同，<client>元素还有一个 url 特性，用于指定远程对象的 URL。
- <wellknown>元素用在服务器和客户端上，指定知名的远程对象。服务器部分如下所示：

```
<wellknown mode="SingleCall"
     type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          objectURI="Hi" />
```

- 当 mode 特性的值为 SingleCall 或 Singleton 时，type 特性的值就是远程对象的类型，其值包括 Wrox.ProCSharp.Hello 名称空间。程序集名 RemoteHello 位于该名称空间后面。objectURI 是在信道中注册的远程对象的名称。
- 客户端上的 type 特性和在服务器上的属性相同。但是在客户端不需要使用 mode 和 objectURL 特性，而需要使用 url 特性定义远程对象的路径：协议、主机名、端口号、应用程序名，以及对象的 URI。

```
<wellknown type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
                url="tcp://localhost:6791/Hello/Hi" />
```

- <activated>元素用于客户端激活的对象。必须使用 type 特性为客户端和服务器应用程序定义类型和程序集：

```
<activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
```

- 使用<channel>元素可以指定信道。<channel>是<channels>元素的子元素，目的是为单个应用程序配置信道集合。在客户端和服务器上，<channel>元素的使用方法相似。使用 XML

特性 ref 可以引用一个预先配置的信道名称。对于服务器信道，必须用 XML 特性 port 设置端口号。XML 特性 displayName 用于指定从.NET Framework Configuration 工具中使用的信道名称，如后面所述：

```
<channels>
   <channel ref="tcp" port="6791" displayName="TCP Channel" />
   <channel ref="http" port="6792" displayName="HTTP Channel" />
   <channel ref="ipc" portName="myIPCPort" displayName="IPC Channel"
        />
</channels>
```

预定义的信道名称是 tcp、http 和 ipc，它们定义 TcpChannel、HttpChannel 和 IpcChannel 类。

54.6.1 知名对象的服务器配置

在示例文件 wellknown_Server.config 中，name 属性的值为 Hello。在下面的配置文件中，把 TCP 信道设置为在端口 6791 处侦听，把 HTTP 信道设置为在端口 6792 处侦听。IPC 信道用端口名 myIPCPort 配置。远程对象类是 RemoteHello 程序集中的 Wrox.ProCSharp.Hello，对象在信道中称为 Hi，对象模式是 SingleCall：

可从 wrox.com 下载源代码

```
<configuration>
   <system.runtime.remoting>
      <application name="Hello">
         <service>
            <wellknown mode="SingleCall"
                       type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
                       objectUri="Hi" />
         </service>
         <channels>
            <channel ref="tcp" port="6791"
               displayName="TCP Channel (HelloServer)" />
            <channel ref="http" port="6792"
               displayName="HTTP Channel (HelloServer)" />
            <channel ref="ipc" portName="myIPCPort"
               displayName="IPC Channel (HelloServer)" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

代码段 ConfigurationFiles/HelloServer/Wellknown.config

54.6.2 知名对象的客户端配置

对于知名对象，必须在客户端配置文件 Wellknown_Client.config 中指定程序集和信道。远程对象的类型可以在 RemoteHello 程序集中找到，Hi 是信道中对象的名称，远程类型 Wrox.ProCSharp.Remoting.Hello 的 URI 是 ipc://myIPCPort/Hello/Hi。在客户端中，也使用 IPC 信道，但是不指定端口，因此可以任意挑选可用的端口。客户端选择的信道必须对应于 URL：


```
<configuration>
   <system.runtime.remoting>
      <application name="Client">
         <client displayName="Hello client for well-known objects">
            <wellknown type = "Wrox.ProCSharp.Remoting.Hello, RemoteHello"
                                 url="ipc://myIPCPort/Hello/Hi" />
         </client>
         <channels>
            <channel ref="ipc" displayName="IPC Channel (HelloClient)" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

代码段 ConfigurationFiles/HelloClient/Wellknown_Client.config

配置文件中的一个小改动是使用 HTTP 信道(如 WellknownHttp_Client.config 所示)：


```
<client>
   <wellknown type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
                      url="http://localhost:6792/Hello/Hi" />
</client>
<channels>
   <channel ref="http" displayName="HTTP Channel (HelloClient)" />
</channels>
```

代码段 ConfigurationFiles/HelloClient/WellknownHttp_Client.config

54.6.3　客户端激活的对象的服务器配置

只需更改配置文件(它可以在 clientactivated_Server.config 中找到)，就可以把服务器配置从服务器激活的对象改为客户端激活的对象。在这里指定<service>元素的<activated>子元素。使用<activated>元素对服务器进行配置时，必须指定 type 特性。application 元素的 name 特性定义 URI：


```
<configuration>
   <system.runtime.remoting>
      <application name="HelloServer">
         <service>
            <activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
         </service>
         <channels>
            <channel ref="http" port="6788"
                      displayName="HTTP Channel (HelloServer)" />
            <channel ref="tcp" port="6789"
                displayName="TCP Channel (HelloServer)" />
            <channel ref="ipc" portName="myIPCPort"
                      displayName="IPC Channel (HelloServer)" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

代码段 ConfigurationFiles/HelloServer/ClientActivated.config

54.6.4 客户端激活的对象的客户端配置

clientactivated_Client.config 配置文件使用<client>元素的 url 特性和<activated>元素的 type 特性定义客户端激活的远程对象：


```
<configuration>
   <system.runtime.remoting>
      <application>
         <client url="http://localhost:6788/HelloServer"
                 displayName="Hello client for client - activated objects">
            <activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
         </client>
         <channels>
            <channel ref="http" displayName="HTTP Channel (HelloClient)" />
            <channel ref="tcp" displayName="TCP Channel (HelloClient)" />
            <channel ref="ipc" displayName="IPC Channel (HelloClient)" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

代码段 ConfigurationFiles/HelloClient/ClientActivated_Client.config

54.6.5 使用配置文件的服务器代码

在服务器代码中，必须使用 RemotingConfiguration 类的静态方法 Configure()配置远程服务。这里在配置文件中定义的所有信道都已用.NET Remoting 运行库创建和配置。可能也希望从服务器应用程序中了解信道配置的信息，因此可以创建静态方法 ShowActivatedServiceTypes() 和 ShowWellKnownServiceTypes()，它们在加载和启动远程处理配置之后调用：


```
public static void Main()
{
   RemotingConfiguration.Configure("HelloServer.exe.config", false);
   Console.WriteLine("Application: {0}", RemotingConfiguration.ApplicationName);
   ShowActivatedServiceTypes();
   ShowWellKnownServiceTypes();
   System.Console.WriteLine("press return to exit");
   System.Console.ReadLine();
}
```

代码段 ConfigurationFiles/HelloServer/Program.cs

这两个函数显示了知名类型和客户端激活的类型的配置信息：

```
public static void ShowWellKnownServiceTypes()
{
   WellKnownServiceTypeEntry[] entries =
      RemotingConfiguration.GetRegisteredWellKnownServiceTypes();
   foreach (var entry in entries)
   {
      Console.WriteLine("Assembly: {0}", entry.AssemblyName);
      Console.WriteLine("Mode: {0}", entry.Mode);
      Console.WriteLine("URI: {0}", entry.ObjectUri);
      Console.WriteLine("Type: {0}", entry.TypeName);
   }
```

```
}
public static void ShowActivatedServiceTypes()
{
   ActivatedServiceTypeEntry[] entries =
   RemotingConfiguration.GetRegisteredActivatedServiceTypes();
   foreach (var entry in entries)
   {
      Console.WriteLine("Assembly: {0}", entry.AssemblyName);
      Console.WriteLine("Type: {0}", entry.TypeName);
   }
}
```

54.6.6　使用配置文件的客户端代码

在客户端代码中，只需使用配置文件 client.exe.config 配置远程处理服务。之后，不论处理的是服务器激活的远程对象还是客户端激活的远程对象，都可以使用 new 运算符创建远程类 Hello 的新实例。但是有一点不同：对于客户端激活的对象，非默认的构造函数和 new 运算符可以一起使用。但服务器激活的对象不行，原因是单一调用对象可以没有状态，因为每次调用之后都会销毁它们，并且单一对象只创建一次。调用非默认的构造函数只对客户端激活的对象有效，因为 new 运算符只能针对这种类型实例化远程对象。

在 HelloClient.cs 文件的 Main()方法中，可以更改远程处理代码，使用配置文件和 RemotingConfiguration.Configure()方法，用 new 运算符创建远程对象：


```
RemotingConfiguration.Configure("HelloClient.exe.config", false);
Hello obj = new Hello();
if (obj == null)
{
   Console.WriteLine("could not locate server");
   return;
}
for (int i=0; i < 5; i++)
{
   Console.WriteLine(obj.Greeting("Stephanie"));
}
```

代码段 ConfigurationFiles/HelloClient/Program.cs

54.6.7　客户端信道的延迟加载

对于.NET Remoting，如果客户端没有配置信道，就会默认配置能自动使用的 3 条信道：

```
<system.runtime.remoting>
   <application>
      <channels>
         <channel ref="http client" displayName="http client (delay loaded)"
                  delayLoadAsClientChannel="true" />
         <channel ref="tcp client" displayName="tcp client (delay loaded)"
                  delayLoadAsClientChannel="true" />
         <channel ref="ipc client" displayName="ipc client (delay loaded")"
                  delayLoadAsClientChannel="true" />
      </channels>
   </application>
</system.runtime.remoting>
```

当 XML 特性 delayLoadAsClientChannel 的值为 true 时，将指定应从没有配置信道的客户端上使用的信道。因为运行库会尝试使用延迟加载的信道连接到服务器，所以不需要在客户端配置文件中配置信道，前面使用的知名对象的客户端配置文件可以变得非常简单：

```
<system.runtime.remoting>
   <application>
      <channels>
         <channel ref="http client" displayName="http client (delay loaded)"
                  delayLoadAsClientChannel="true" />
         <channel ref="tcp client" displayName="tcp client (delay loaded)"
                  delayLoadAsClientChannel="true" />
         <channel ref="ipc client" displayName="ipc client (delay loaded")"
                  delayLoadAsClientChannel="true" />
      </channels>
   </application>
</system.runtime.remoting>
```

54.6.8 调试配置

如果有一个误配置的服务器配置文件(例如，指定远程程序集的错误名称)，在服务器启动时就检测不出这个错误。在客户端实例化远程对象并调用方法时，才能检测出该错误。客户端会得到一个异常，说明未找到该远程对象程序集。指定<debug loadTypes="true" />配置，会在服务器启动时加载远程对象，并在服务器上实例化该对象。这样，如果配置文件配置错误，就会在服务器上出现一个错误。

```
<configuration>
   <system.runtime.remoting>
      <application name="Hello">
         <service>
            <wellknown mode="SingleCall"
                       type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
                       objectUri="Hi" />
         </service>
         <channels>
            <channel ref="tcp" port="6791"
               displayName="TCP Channel (HelloServer)" />
         </channels>
      </application>
      <debug loadTypes="true" />
   </system.runtime.remoting>
</configuration>
```

54.6.9 配置文件中的生命周期服务

远程服务器的租约配置也可以使用应用程序配置文件来完成。<lifetime>元素有 leaseTime、sponsorshipTimeOut、renewOnCallTime 和 pollTime 特性，如下面的示例所示：

```
<configuration>
   <system.runtime.remoting>
      <application>
         <lifetime leaseTime = "15M" sponsorshipTimeOut = "4M"
                   renewOnCallTime = "3M" pollTime = "30s"/>
      </application>
```

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

使用配置文件时，不用更改源代码，通过编辑文件就可以完成对远程处理配置的更改。可以很容易把信道改为使用 TCP 而不使用 HTTP，并更改端口和信道的名称等内容。在配置文件中添加一行内容，就可以让服务器侦听两个信道，而不是一个。

54.6.10　格式化程序提供程序

本章前面已经讨论了更改格式化程序提供程序的属性，以支持跨网络编组所有对象。这里不像前面那样通过编程的方式实现，而是在配置文件中配置格式化程序提供程序的属性。

下面的服务器配置文件在<channel>元素进行更改，其中把<serverProviders>和<clientProviders>定义为子元素。在<serverProviders>元素中，引用了本地提供程序 wsdl、soap 和 binary，对于 soap 和 binary 提供程序，把 typeFilterLevel 属性设置为 Full。

```
<configuration>
   <system.runtime.remoting>
      <application name="HelloServer">
         <service>
            <activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
         </service>
         <channels>
           <channel ref="tcp" port="6789" displayName="TCP Channel (HelloServer)">
             <serverProviders>
                <provider ref="wsdl" />
                <provider ref="soap" typeFilterLevel="Full" />
                <provider ref="binary" typeFilterLevel="Full" />
             </serverProviders>
             <clientProviders>
                <provider ref="binary" />
             </clientProviders>
           </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

54.7　在 ASP.NET 中驻留远程服务器

迄今为止，所有服务器示例都是运行在自驻留(self-hosted)的.NET 服务器上。自驻留的服务器必须手动启动。.NET Remoting 服务器也可以在许多其他的应用程序类型中启动。在 Windows 服务中，服务器可以在系统启动时自动启动，此外，进程可以通过系统账户的证书运行。关于 Windows 服务的内容，可以参阅第 25 章。

ASP.NET 对.NET Remoting 服务器有一种特殊支持。ASP.NET 可用于自动启动远程服务器。与可执行的驻留应用程序相反，驻留在 ASP.NET 中的.NET Remoting 在配置时使用不同的文件，但语法相同。

为了使用 IIS(Internet Information Server，Internet 信息服务器)和 ASP.NET 中的基础结构，必须创建一个派生自 System.MarshalByRefObject 类的类，该类具有默认的构造函数。不再需要以前为服务器创建和注册信道所使用的代码；这些代码所做的工作可以由 ASP.NET 运行库完成。此外，也必须在 Web 服务器上创建一个虚拟目录，该目录映射到保存 Web.config 配置文件的目录上。远程类的程序集必须驻留在子目录 bin 中。

可以使用 IIS MMC 配置 Web 服务器上的虚拟目录。选择 Default Web Site 并打开 Action 菜单，就可以创建一个新的虚拟目录。

Web 服务器上的 Web.config 配置文件必须放在虚拟网站的主目录中。使用默认的 IIS 配置，将使用的信道会侦听端口 80：

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown mode="SingleCall"
               type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
               objectUri="HelloService.soap" />
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

如果远程对象驻留在 IIS 中，根据所使用的格式化程序类型(SOAP 或二进制)远程对象的名称就必须以.soap 或.bin 结尾。

现在，客户端使用下面的配置文件就可以连接到远程对象上。在这里必须指定远程对象的 URL，这个 URL 包括 Web 服务器 localhost、Web 应用程序的名称 RemoteHello(该名称在创建虚拟网站时指定)、远程对象 HelloService.soap(在文件 Web.config 中定义)的 URI。因为端口号 80 是 HTTP 协议的默认端口，所以不必指定它。不指定<channels>部分表示使用 machine.config 配置文件中延迟加载的 HTTP 信道：

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client url="http:/localhost/RemoteHello">
            <wellknown type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
                       url="http://localhost/RemoteHello/HelloService.soap" />
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

ASP.NET 中驻留的远程对象只支持知名对象，不支持客户端激活的对象。

54.8　类、接口和 Soapsuds

迄今为止，在所有.NET Remoting 示例中，总是把远程对象的程序集复制到服务器和客户端应用程序中。这样远程对象的 MSIL 代码就在客户端和服务器系统中(尽管在客户端应用程序中只需要元数据)。然而，复制远程对象的程序集表示客户端和服务器不能独立编程。为了只使用元数据，更好的方法是使用接口或 Soapsuds.exe 实用程序。

54.8.1　接口

使用接口可以把客户端代码和服务器代码完全分离开。接口只定义没有实现的方法。这样，协定(接口)和实现方式就完全分开了。而客户端系统只需要协定。使用接口时必须按照下面的步骤进行：

(1) 定义一个接口，把它放到一个独立的程序集中。
(2) 在远程对象类中实现这个接口。为此，必须引用接口的程序集。
(3) 在服务器端不需要进行更改，可以按照通常的方式配置服务器和对服务器进行编程。
(4) 在客户端，引用接口的程序集，而不引用远程类的程序集。
(5) 现在，客户端可以使用远程对象的接口，而不使用远程对象类。类似以前创建对象的方法，可以使用 Activator 类创建对象。由于接口本身不能实例化，因此不能使用 new 运算符。

由于接口定义了客户端和服务器之间的协定，因此客户端应用程序和服务器应用程序就可以独立地开发。此外，即使坚持旧的 COM 规则(即接口不应该更改)，也不会出现任何版本问题。

54.8.2　Soapsuds

如果使用 HTTP 信道和 SOAP 格式化程序，就可以使用 Soapsuds 实用程序从程序集中获取元数据。Soapsuds 可以把程序集转化为 XML 架构、把 XML 架构转化为包装类；也可以把包装类转化为 XML 架构，把 XML 架构转化为程序集。

下面的命令行把 Hello 类型从 RemoteHello 程序集转化为 HelloWrapper 程序集，在转化过程中将生成调用远程对象的透明代理：

```
soapsuds -types:Wrox.ProCSharp.Remoting.Hello,RemoteHello -oa:HelloWrapper.dll
```

如果使用 HTTP 信道和 SOAP 格式化程序，那么也可以直接使用 Soapsuds 从正在运行的服务器中获取类型信息：

```
soapsuds -url:http://localhost:6792/hello/hi?wsdl - oa:HelloWrapper.dll
```

现在，可以在客户端中引用 Soapsuds 生成的程序集，而不引用原始程序集。表 54-2 列出了 Soapsuds 命令的一些选项。

表　54-2

选　项	描　述
-url	从指定的 URL 中检索架构
-proxyurl	如果需要代理服务器访问服务器，就使用该选项指定代理
-types	指定类型和程序集，以便从中读取架构信息

(续表)

选　项	描　述
-is	输入架构文件
-ia	输入程序集文件
-os	输出架构文件
-oa	输出程序集文件

54.9　异步远程调用

如果服务器的方法要花费一段时间才能完成，同时客户端需要做一些不同的工作，就没有必要启动一个单独的线程进行远程调用。而进行异步调用，使方法启动后立即返回给客户端。在远程对象上可以进行异步调用，因为在委托的帮助下它们就像是在本地对象上调用一样。对于没有返回值的方法，也可以使用 OneWay 特性。

54.9.1　使用委托和.NET Remoting

为了实现方法的异步调用，创建一个委托 GreetingDelegate，这个委托的参数和返回值与远程对象的 Greeting()方法相同。泛型委托 Func <T>和 Action <T>非常适合于这种情况。Greeting()方法需要一个返回 string 类型的 string 参数 Func<string, string>。这是调用这个方法的委托定义。使用委托的 BeginInvoke()方法，启动 Greeting()调用。BeginInvoke()方法的第二个参数是一个 AsyncCallback 实例，该实例定义 HelloClient.Callback()方法。当远程方法完成时，会调用 HelloClient.Callback()方法。在 Callback()方法中，远程调用使用 EndInvoke()方法完成：

```
using System;
using System.Runtime.Remoting;
namespace Wrox.ProCSharp.Remoting
{
   public class HelloClient
   {
      private static string greeting;

      public static void Main()
      {
         RemotingConfiguration.Configure("HelloClient.exe.config");
         Hello obj = new Hello();
         if (obj == null)
         {
            Console.WriteLine("could not locate server");
            return;
         }
         Func<string, string> d = new Func<string, string>(obj.Greeting);
         IAsyncResult ar = d.BeginInvoke("Stephanie", null, null);

         // do some work and then wait
         ar.AsyncWaitHandle.WaitOne();
         string greeting = null;
         if (ar.IsCompleted)
```

```
            {
                greeting = d.EndInvoke(ar);
            }

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

第 8 章详细介绍了委托，第 20 章介绍了如何异步地使用委托。

54.9.2　OneWay 特性

返回值为 void、只有输入参数的方法可以由 OneWay 特性标记。不论客户如何调用方法，OneWay 特性(在 System.Runtime.Remoting.Messaging 名称空间中定义)都可以实现方法的自动异步。把 TakeAWhile()方法添加到远程对象类 RemoteHello 中，就可以创建一个“即发即弃(fire-and-forget)”的方法。如果客户端通过代理调用 TakeAWhile() 方法，则代理会立即返回给客户端。在服务器上，该方法会在一段时间后完成：

```
[OneWay]
public void TakeAWhile(int ms)
{
    Console.WriteLine("TakeAWhile started");
    System.Threading.Thread.Sleep(ms);
    Console.WriteLine("TakeAWhile finished");
}
```

54.10　.NET Remoting 的安全性

自从.NET 2.0 以来，.NET Remoting 就支持安全性。.NET Remoting 可以保护通过网络传输的数据，并可以验证用户的身份。

安全性可以使用每条管道配置。TCP、HTTP 和 IPC 信道支持安全性配置。在服务器配置中，必须定义通信的最低安全要求，而客户端配置定义与安全性相关的功能。如果客户端定义的安全性配置低于服务器指定的最低要求，通信就会失败。

首先介绍服务器配置。在下面摘录的配置文件中，显示了如何为服务器定义安全性。

使用 XML 属性 protectionLevel，可以指定服务器是否需要跨网络传递加密的数据。protectionLevel 属性可以设置的值有 None、Sign 和 EncryptAndSign。使用 Sign 值会创建一个签名，用于验证发送的数据是否跨网络传递过程中不更改。但是，使用嗅探器仍可以读取网络上的所有数据。对于 EncryptAndSign 值，发送的数据也是加密的，于是没有正式参与消息传输的系统不能读取数据。

impersonate 属性可以设置为 true 或 false。如果把 impersonate 属性设置为 true，服务器就可以模拟客户端的用户，以他的名义访问资源。

```
<channels>
    <channel ref="tcp" port="9001"
        secure="true"
        protectionLevel="EncryptAndSign"
```

```
        impersonate="false" />
</channels>
```

在客户端配置中，定义网络通信的功能。如果该功能不能满足服务器的要求，通信就会失败。在下面的示例配置中，配置 protectionLevel 和 TokenImpersonationLevel。

protectionLevel 可以设置为与服务器配置文件相同的选项。

TokenImpersonationLevel 可以设置为 Anonymous、Identification、Impersonation 和 Delegation。如果把 TokenImpersonationLevel 设置为 Anonymous，服务器就不能标识客户端的用户。如果把属性值设置为 Identification，服务器就可以找出服务器的用户标识。如果服务器把 impersonate 配置为 true，则客户端只有配置为 Anonymous 或 Identification 时，通信才失败。把 TokenImpersonationLevel 设置为 Impersonation，就允许服务器模拟客户端。把 TokenImpersonationLevel 设置为 Delegation，不但允许服务器模拟客户端访问服务器上的本地资源，还可以使用用户标识访问其他服务器上的资源。只有像使用 Active Directory 那样把 Kerberos 用于用户的登录身份验证，才能使用 Delegation。

```
<channels>
   <channel ref="tcp"
      secure="true"
      protectionLevel="EncryptAndSign"
      TokenImpersonationLevel="Impersonate" />
</channels>
```

如果以编程方式创建信道，而不使用配置文件，那么也可以以编程方式定义安全性设置。可以把包含所有安全性设置的集合传送给信道的构造函数，如下所示。集合类的要求是必须执行 IDictionary 接口，例如泛型类 Dictionary 或 Hashtable 类。

```
var dict = new Dictionary <string, string>();
dict.Add("secure", "true");

var clientChannel = new TcpClientChannel(dict, null);
```

安全性详见第 21 章。

54.11 远程处理和事件

使用.NET Remoting，不但客户端可以跨网络调用远程对象上的方法，而且服务器也可以调用客户端中的方法。关于这个方面，从基本的语言功能中已经知道，可以使用的机制是委托和事件。

总的来说，体系结构比较简单。服务器有客户端可以调用的远程对象，而客户端有服务器可以调用的远程对象：

- 服务器中的远程对象必须声明一个带有方法签名的外部函数(委托)，以便客户端在处理程序中实现这些外部函数。
- 由于与处理程序函数一起传递给客户端的参数必须是可编组的，因此发送给客户端的所有数据必须是序列化的。
- 远程对象也必须声明一个委托函数的实例，委托函数可以通过 event 关键字更改。客户端将使用该实例注册处理程序。

- 客户端必须创建一个带有处理程序方法的接收器对象，其中的处理程序方法必须与委托定义的接受器对象有一样的签名。此外，客户端必须注册接收器对象，这个接收器对象带有远程对象中的事件。

下面举一个例子。为了查看通过.NET Remoting 处理的事件的所有内容，需要创建 5 个类：Server、Client、RemoteObject、EventSink 和 StatusEventArgs。它们的依赖关系如图 54-5 所示。

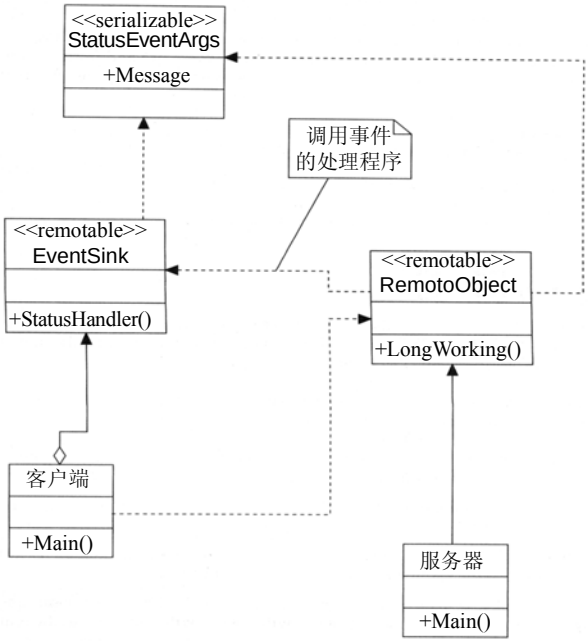

图　54-5

Server 类就像我们已经知道的远程处理服务器一样。Server 类将根据配置文件中的信息创建信道，并且注册远程对象，在远程处理运行库中，远程对象将在 RemoteObject 类中实现。远程对象声明委托的参数，并且触发已注册的处理程序函数中的事件。传递给处理程序函数的参数是 StatusEventArgs 类型。StatusEventArgs 类必须是可序列化的，以便它可以编组传递给客户端。

Client 类表示客户端应用程序。这个类创建 EventSink 类的一个实例，并且把 EventSink 类的 StatusHandler()方法注册为远程对象中委托的处理程序。因为 RemoteObject 类也跨网络调用，所以 EventSink 类必须像 RemoteObject 类一样是远程类。

54.11.1　远程对象

远程对象类在 RemoteObject.cs 文件中实现。如以前的示例所示，远程对象类必须派生自 MarshalByRefObject。为了使客户端能够注册从远程对象中调用的事件处理程序，必须使用 delegate 关键字声明一个外部函数。我们声明的 StatusEvent()委托带有两个参数：sender(使用这个参数，客户端可以确定激活事件的对象)和一个 StatusEventArgs 类型的变量。可以把要发送给客户端的所有附加信息都放到该参数类中。

要在客户端中执行的方法有一些严格的要求。它只能有输入参数，而不允许有返回类型、ref 和 out 参数；参数类型必须是[Serializable]或远程的(派生自 Marshal ByRefObject)，使用 EventHandler-

<StatusEventArgs>委托定义的参数可以满足这些要求：

在 RemoteObject 类中，声明 EventHandler<StatusEvent>类型的一个 Status 事件。客户端必须给 Status 事件添加事件处理程序，以便从远程对象中获取状态信息：


```
public class RemoteObject : MarshalByRefObject
{
   public RemoteObject()
   {
      Console.WriteLine("RemoteObject constructor called");
   }
   public event EventHandler<StatusEventArgs>Status;
```

代码段 RemotingAndEvents/RemoteObject/RemoteObject.cs

在调用 Status(this, e)方法触发事件之前，LongWorking()方法检查事件处理程序是否已注册。为了验证事件是否异步地触发，就在执行 Thread.Sleep()之前在方法的开端触发一个事件，并在执行之后触发一个事件：

```
public void LongWorking(int ms)
{
   Console.WriteLine("RemoteObject: LongWorking() Started");
   StatusEventArgs e = new StatusEventArgs(
                          "Message for Client: LongWorking() Started");
   // fire event
   if (Status != null)
   {
      Console.WriteLine("RemoteObject: Firing Starting Event");
      Status(this, e);
   }
   System.Threading.Thread.Sleep(ms);
   e.Message = "Message for Client: LongWorking() Ending";
   // fire ending event
   if (Status != null)
   {
      Console.WriteLine("RemoteObject: Firing Ending Event");
      Status(this, e);
   }
   Console.WriteLine("RemoteObject: LongWorking() Ending");
}
```

54.11.2 事件参数

在 RemoteObject 类中可以看出，StatusEventArgs 类用作委托的参数。使用 Serializable 特性，可以把 StatusEventArgs 类的实例从服务器传输给客户端。我们使用 string 类型的一个简单属性把消息发送给客户端：


```
[Serializable]
public class StatusEventArgs : EventArgs
{
   public StatusEventArgs(string message)
   {
      this.Message = message;
   }
```

```
    public string Message { get; set; }
}
```

代码段 RemotingAndEvents/RemoteObject/RemoteObject.cs

54.11.3　服务器

在控制台应用程序中实现服务器。使用 RemotingConfiguration.Configure()方法，读取配置文件，从而设置信道及远程对象。服务器使用 Console.ReadLine()方法等待用户终止应用程序：


```
using System;
using System.Runtime.Remoting;
namespace Wrox.ProCSharp.Remoting
{
   class Server
   {
      static void Main()
      {
         RemotingConfiguration.Configure("Server.exe.config", true);
         Console.WriteLine("press return to exit");
         Console.ReadLine();
      }
   }
}
```

代码段 RemotingAndEvents/RemoteServer/Program.cs

54.11.4　服务器配置文件

创建服务器配置文件 Server.exe.config 的方法前面已经讨论过。其中有一个要点：因为客户端首先注册事件处理程序，之后才调用远程方法，所以远程对象必须为客户端保存状态。由于不能使用带有事件的单一调用对象，因此要把 RemoteObject 类配置为客户端激活的类型。另外，为了支持委托，必须用<provider>元素指定 typeFilterLevel 特性，启用完整的序列化：


```
<configuration>
   <system.runtime.remoting>
      <application name="CallbackSample">
         <service>
            <activated type="Wrox.ProCSharp.Remoting.RemoteObject,
                              RemoteObject" />
         </service>
         <channels>
            <channel ref="tcp" port="6791">
               <serverProviders>
                  <provider ref="binary" typeFilterLevel="Full" />
               </serverProviders>
            </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

代码段 RemotingAndEvents/RemoteServer/app.config

54.11.5 事件接收器

客户端需要使用事件接收器库，服务器也需要调用事件接收器库。事件接收器实现处理程序 StatusHandler()，StatusHandler()与委托一起定义。如前所述，方法只有输入参数，不返回任何 Void 值。因为将从服务器中远程调用 EventSink 类，所以它必须继承自 MarshalByRefObject 类，以便使它成为远程类：


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

namespace Wrox.ProCSharp.Remoting
{
   public class EventSink : MarshalByRefObject
   {
      public EventSink()
      {
      }
      public void StatusHandler(object sender, StatusEventArgs e)
      {
         Console.WriteLine("EventSink: Event occurred: " + e.Message);
      }
   }
}
```

代码段 RemotingAndEvents/EventSink/EventSink.cs

54.11.6 客户端

客户端通过 RemotingConfiguration 类读取客户端配置文件。这与以前客户端的行为没有区别。客户端在本地创建远程接收器类 EventSink 的一个实例。从服务器上的远程对象中调用的方法应传递给远程对象：


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

namespace Wrox.ProCSharp.Remoting
{
   class Client
   {
      static void Main()
      {
         RemotingConfiguration.Configure("Client.exe.config", true);
         Console.WriteLine("wait for server...");
         Console.ReadLine();
```

代码段 RemotingAndEvents/RemoteClient/Program.cs

不同的是，必须在本地创建远程接收器类 EventSink 的一个实例。由于不使用<client>元素配置 EventSink 类，因此它在本地实例化。接下来，实例化远程对象类 RemoteObject。由于在<client>元素中配置 RemoteObject 类，因此它在远程服务器上实例化：

```
var sink = new EventSink();
var obj = new RemoteObject();
if (!RemotingServices.IsTransparentProxy(obj))
```

```
        {
            Console.WriteLine("check your remoting configuration");
            return;
        }
```

现在，在远程对象中注册 EventSink 对象的处理程序方法。StatusEvent 是在服务器中定义的委托的名称。StatusHandler()方法的参数与在 StatusEvent 中定义的参数完全相同。

调用 LongWorking()方法时，服务器将在方法的开始和结尾处回调 StatusHandler()方法：

```
        // register client sink in server - subscribe to event
        obj.Status += sink.StatusHandle;
        obj.LongWorking(5000);
```

因为现在不用再考虑接收来自服务器的事件，所以取消订阅事件。下一次调用 LongWorking()方法时，不接收事件：

```
            // unsubscribe from event
            obj.Status -= sink.StatusHandler;
            obj.LongWorking(5000);
            Console.WriteLine("press return to exit");
            Console.ReadLine();
        }
      }
    }
```

54.11.7　客户端配置文件

客户端的配置文件 client.exe.config 与客户端激活的对象(这类对象前面介绍过)的配置文件几乎完全相同。唯一不同的是定义信道的端口号。由于服务器必须使用已知的端口才能到达客户端，因此必须把信道的端口号定义为<channel>元素的特性。因为客户端将使用 new 运算符在本地实例化 EventSink 类，所以不必为这个类定义<service>部分。服务器不是通过其名称访问该对象，而它接收对这个实例的编组引用：

可从 wrox.com 下载源代码

```
<configuration>
   <system.runtime.remoting>
      <application name="Client">
         <client url="tcp://localhost:6791/CallbackSample">
            <activated type="Wrox.ProCSharp.Remoting.RemoteObject,
                              RemoteObject" />
         </client>
         <channels>
            <channel ref="tcp" port="0">
               <serverProviders>
                  <provider ref="binary" typeFilterLevel="Full" />
               </serverProviders>
            </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

代码段 RemotingAndEvents/RemoteClient/app.config

54.11.8 运行程序

下面是服务器的最终输出。因为我们拥有客户端激活的对象，所以远程对象的构造函数被调用一次。接下来，开始调用 LongWorking()方法，触发客户端的事件。之后，由于客户端已经注销其关心的事件，所以下一次启动 LongWorking()方法不会触发事件。

```
press return to exit
RemoteObject constructor called
RemoteObject: LongWorking() Started
RemoteObject: Firing Starting Event
RemoteObject: Firing Ending Event
RemoteObject: LongWorking() Ending
RemoteObject: LongWorking() Started
RemoteObject: LongWorking() Ending
```

在客户端，可以看到跨网络触发的事件。

```
wait for server...

EventSink: Event occurred: Message for Client: LongWorking() Started
EventSink: Event occurred: Message for Client: LongWorking() Ending
```

54.12 调用上下文

客户端激活的对象能够保存某一具体客户端的状态。使用客户端激活的对象时，服务器需要为每个客户端分配资源。而使用服务器激活的 SingleCall 对象时，每个实例调用都会创建一个新的实例。由于单一调用对象不保存客户的状态，因此在服务器上不占用资源。在管理状态时，可以在客户端保存状态，对象的状态信息随每一次方法调用发送给服务器。要实现这样的状态管理，因为可以使用调用上下文自动完成，所以不必改变所有方法签名，使它包括附加参数，附加参数把状态传递给服务器。

调用上下文随逻辑线程一起流动，并和每一个方法调用一起传递。逻辑线程从主调线程开始，经过从主调线程中开始的所有方法调用，并且通过不同的上下文、不同的应用程序域和不同的进程。

使用 CallContext.SetData()方法可以把数据赋予调用上下文。该对象可以用作 SetData()方法的数据，但是这些对象所属的类必须实现 ILogicalThreadAffinative 接口。使用 CallContext.GetData()方法也可以在同一逻辑线程(但有可能是不同的物理线程)中得到同样的数据。

使用 CallContextData 类为调用上下文的数据创建一个新的 C#类库。CallContextData 类用于把一些数据随每一次方法调用从客户端传递给服务器。与调用上下文一起传递的类必须实现 System.Runtime.Remoting.Messaging.ILogicalThreadAffinative 接口。该接口没有方法，它只是运行库的一个标记，作用是指定 CallContextData 类的实例应该与逻辑线程一起流动。此外，CallContextData 类也必须使用 Serializable 特性标记，以便通过信道传输它：

```
using System;
using System.Runtime.Remoting.Messaging;
namespace Wrox.ProCSharp.Remoting
{
   [Serializable]
```

```
    public class CallContextData : ILogicalThreadAffinative
    {
        public CallContextData()
        {
        }
        public string Data { get; set; }
    }
}
```

在 Hello 远程对象类中，需要更改 Greeting()方法，以便访问调用上下文。为了使用 CallContextData 类，必须引用前面在 CallContextData.dll 文件中创建的 CallContextData 程序集。此外，为了使用 CallContext 类，必须打开 System.Runtime.Remoting.Messaging 名称空间。cookie 变量保存从客户传递给服务器的数据。因为该上下文的工作方式类似于基于浏览器的 cookie，客户端会自动把数据传递给 Web 服务器，所以把该变量命名为 cookie：

```
public string Greeting(string name)
{
    Console.WriteLine("Greeting started");
    CallContextData cookie = (CallContextData)CallContext.GetData("mycookie");
    if (cookie != null)
    {
        Console.WriteLine("Cookie: " + cookie.Data);
    }
    Console.WriteLine("Greeting finished");
    return "Hello, " + name;
}
```

在客户端代码中，调用 CallContext.SetData()方法，以设置调用上下文信息。在这个方法中，指定要传递给服务器的 CallContextData 类的实例。现在，每次在 for 循环中调用 Greeting()方法时，上下文数据都会自动传递给服务器：

```
CallContextData cookie = new CallContextData();
cookie.Data = "information for the server";
CallContext.SetData("mycookie", cookie);
for (int i=0; i < 5; i++)
{
    Console.WriteLine(obj.Greeting("Christian"));
}
```

调用上下文可以用于发送用户的信息、客户端系统的名称或者仅发送唯一标识符，在服务器端，使用唯一标识符可以从数据库中获取状态信息。

54.13　小结

在本章中我们看到，.NET Remoting 有助于跨网络调用方法。远程对象必须继承自 MarshalByRefObject。在服务器应用程序中，要加载配置文件，只需要一个方法，就可以设置和运行信道和远程对象。在客户端中，我们加载配置文件，并使用 new 运算符实例化远程对象。

即使不使用配置文件，也可以使用.NET Remoting。在服务器上，只需创建信道和注册远程对象；

而在客户端上，只需创建信道和使用远程对象。

此外，.NET Remoting 体系结构也非常灵活，并可以扩展。这项技术的所有部分，如信道、代理、格式化程序、消息接收器等都是可插入的，并可以用自定义实现方式替代。

使用 HTTP、TCP 和 IPC 信道可以跨网络通信，使用 SOAP 和二进制格式化程序可以在发送参数前对其进行格式化。

最后讨论了无状态和有状态的对象类型的用法，它们可以由知名对象和客户端激活的对象使用。而使用客户端激活的对象可以确定如何使用租约机制指定远程对象的生命周期。

我们还讨论了.NET Remoting 可以与.NET Framework 的其他部分有机集成，如调用异步方法、使用 delegate 和 event 关键字执行回调等。