一、简介

有太多的原因让我忽略了 Remoting，不过现在用它来开始 SOA 和 WCF 的旅途还是不错的选择。.NET Remoting 封装了分布式开发的消息编码和通讯方式，让我们用非常简单的方式既可完成不同模式的分布系统开发，同时其可配置、可扩展的特性也让我们拥有极大的灵活性。当然，我更看好其升级版本 —— WCF。

要了解 Remoting 的基本信息和介绍，还是看 MSDN 比较好。先写一个简单的 Example 来体验一下，为了方便，我直接在一个工程里面创建不同的应用程序域来模拟分布模式。

```
using System;
using System.Reflection;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;

namespace Learn.Library.Remoting
{
public class RemotingTest
{
/// <summary>
///  远程类型
/// </summary>
public class Data : MarshalByRefObject
{
private int i;

public int I
{
get { return i; }
```

```
}

public void Where()
{
Console.WriteLine("{0} in {1}", this.GetType().Name,
AppDomain.CurrentDomain.FriendlyName);
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
// 创建新的应用程序域，以便模拟分布系统。
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
// 创建并注册信道
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

// 注册远程对象激活模式
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.Singleton);
});
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
// 创建并注册信道
TcpClientChannel channel = new TcpClientChannel();
```

```
ChannelServices.RegisterChannel(channel, false);

// 创建远程对象并调用其方法
Data data = (Data)Activator.GetObject(typeof(Data), "tcp://localhost:801/data");
data.Where();

// 判断是否是代理
Console.WriteLine(RemotingServices.IsTransparentProxy(data));
}

static void Main()
{
Server();
Client();
}
}
}
```

在 Remoting 中，核心内容包括 "远程对象" 和 "信道"，前者是我们要使用的内容，后者则提供了分布环境的封装。使用 Remoting 一般包括如下步骤：

1. 创建可远程处理的类型。
2. 注册信道。
3. 注册远程类型(以及其激活方式)。
4. 创建远程对象代理，完成调用。

后面的章节将就这些内容去做点研究。

在分布系统中，远程对象需要跨越应用程序域进行传递，因此其表示方式会有所不同。基于性能和数据共享等原因考虑，Remoting 中远程对象可以是 "值封送对象(MBV)" 或 "引用封送对象(MBR)"。

MBV 机制类似于 Web 无状态请求，服务器创建对象实例传递给信道发送到客户端，而后服务器端不再继续维护其状态和生存期。而 MBR 则在其生存期内一直存活在服务器程序域中，客户端只是通过代理对象来完成调用消息传递，客户端可以通过相关接口来延长远程

对象的生存期。

实现 MBV 一般通过 SerializableAttribute 特性，或者实现 ISerializable 接口。运行下面的例子，我们会发现远程对象在客户端程序域内，并且不是代理对象。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
/// <summary>
///  远程类型
/// </summary>
[Serializable]
public class Data
{
private int i;

public int I
{
get { return i; }
set { i = value; }
}
```

```
public void Where()
{
Console.WriteLine("{0} in {1}", this.GetType().Name,
AppDomain.CurrentDomain.FriendlyName);
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
// 创建新的应用程序域，以便模拟分布结构。
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
// 创建并注册信道
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

// 注册远程类型
RemotingConfiguration.ApplicationName = "test";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Data));
});
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
// 创建并注册信道
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);
```

```
// 注册远程类型
RemotingConfiguration.RegisterActivatedClientType(typeof(Data),
"tcp://localhost:801/test");

// 创建远程对象并调用其方法
Data data = Activator.CreateInstance(typeof(Data)) as Data;
data.Where();

// 判断是否是代理
Console.WriteLine(RemotingServices.IsTransparentProxy(data));
}

static void Main()
{
Server();
Client();
}
}
}
```

输出：

```
Data in Learn.CUI.vshost.exe
False
```

MBR 则要求继承自 MarshalByRefObject 或 ContextBoundObject。继承自 ContextBoundObject 的远程对象，可以包含其执行上下文，比如事务等等，其性能不如 MarshalByRefObject。下面的例子中，远程对象依旧在服务器程序域内执行，客户端只是一个代理对象在转发调用。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
```

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
/// <summary>
/// 远程类型
/// </summary>
public class Data : MarshalByRefObject
{
private int i;

public int I
{
get { return i; }
set { i = value; }
}

public void Where()
{
Console.WriteLine("{0} in {1}", this.GetType().Name,
AppDomain.CurrentDomain.FriendlyName);
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
```

```
{
// 创建新的应用程序域，以便模拟分布结构。
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
// 创建并注册信道
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

// 注册远程类型
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.Singleton);
});
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
// 创建并注册信道
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);

// 创建远程对象并调用其方法
Data data = (Data)Activator.GetObject(typeof(Data), "tcp://localhost:801/data");
data.Where();

// 判断是否是代理
Console.WriteLine(RemotingServices.IsTransparentProxy(data));
}

static void Main()
{
Server();
```

```
Client();
}
}
}
```

输出：

```
Data in server
True
```

MBV 传递完整的副本到客户端，而后的所有调用都是在客户端程序域内进行，不再需要跨域的往返过程。但完整复制和序列化需要更多的资源和往返时间，因此不适合较大的对象，同时无法在多个客户端间共享信息。而 MBR 的所有调用都是通过代理进行的，因此往返过程比较多，但因执行过程都在服务器端进行，其整体性能还是非常高的。在系统设计时可以针对不同的应用采取不同的选择。

三：激活模式

对于 MBR，我们可以指定不同的激活模式。

- **服务器激活(Server-Activated Objects / SAO)**：只有在客户端调用代理对象第一个方法时才创建，区分为 Singleton 和 SingleCall 两种模式。Singleton 一如设计模式中的名称，无论有多少客户端都只有一个实例存在；而 SingleCall 则为每次调用创建一个新对象，因此它是无状态的。SingleCall 在方法调用完成后立即失效，不会参与生存期租约系统。
- **客户端激活(Client-Activated Objects / CAO)**：在客户端调用 new 或 Activator.CreateInstance 时立即创建。

通常情况下服务器激活模式只能使用默认构造函数，客户端激活模式则没有此限制。当然，可以通过其他方法来绕开这个限制，比如动态发布，我们会在后面的章节解决这个问题。

在 Remoting 中，我们使用 RemotingConfiguration 类型来完成远程类型的注册。RegisterWellKnownServiceType()、RegisterWellKnownClientType() 用于注册服务器激活对象，通过 WellKnownObjectMode 枚举参数可以指定 Singleton 或 SingleCall 模式。RegisterActivatedServiceType()、RegisterActivatedClientType() 用于注册客户端激活对象。

注册方式演示

1. SAO / Singleton

```
// Server
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.Singleton);

...

// Client1
RemotingConfiguration.RegisterWellKnownClientType(typeof(Data),
"tcp://localhost:801/data");
Data data = new Data(); //  必须使用默认构造方法。

...

// Client2
Data data = (Data)Activator.GetObject(typeof(Data), "tcp://localhost:801/data");
```

2. SAO / SingleCall

```
// Server
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.SingleCall);

...

// Client1
RemotingConfiguration.RegisterWellKnownClientType(typeof(Data),
"tcp://localhost:801/data");
Data data = new Data(); //  必须使用默认构造方法。

...
```

```
// Client2
Data data = (Data)Activator.GetObject(typeof(Data), "tcp://localhost:801/data");
```

3. CAO

由于 RegisterActivatedServiceType 中不能为远程对象指定 URI，因此我们需要使用 ApplicationName 属性。

```
// Server
RemotingConfiguration.ApplicationName = "test"; //
RemotingConfiguration.RegisterActivatedServiceType(typeof(Data));

...

// Client1
RemotingConfiguration.RegisterActivatedClientType(typeof(Data),
"tcp://localhost:801/test");
Data data = new Data(123); //  可以使用非默认构造方法

...

// Client2
RemotingConfiguration.RegisterActivatedClientType(typeof(Data),
"tcp://localhost:801/test");
Data data = (Data)Activator.CreateInstance(typeof(Data), 123); //  可以使用非默认构造方法
```

创建时间的区别

我们在远程对象创建和方法调用代码之间进行一些延时来观察创建时间的异同。

1. 服务器激活

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;
```

```
using System.Reflection;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
/// <summary>
///  远程类型
/// </summary>
public class Data : MarshalByRefObject
{
public Data()
{
Console.WriteLine(DateTime.Now);
}

public void Test()
{
Console.WriteLine(DateTime.Now);
}
}

/// <summary>
///  服务器端代码
/// </summary>
static void Server()
{
```

```
// 创建新的应用程序域，以便模拟分布结构。
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
// 创建并注册信道
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

// 注册远程类型
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.SingleCall);
});
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
// 创建并注册信道
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);

// 创建远程对象并调用其方法
Data data = (Data)Activator.GetObject(typeof(Data), "tcp://localhost:801/data");
Thread.Sleep(3000);
data.Test();
}

static void Main()
{
Server();
Client();
}
```

```
}
}
```

输出：

2007-2-23 14:22:06

2007-2-23 14:22:06

输出结果表明，对象在方法调用时才被创建。

2. 客户端激活

```
using System;
using System.Threading;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
/// <summary>
///  远程类型
/// </summary>
public class Data : MarshalByRefObject
{
public Data()
{
```

```
Console.WriteLine(DateTime.Now);
}

public void Test()
{
Console.WriteLine(DateTime.Now);
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
// 创建新的应用程序域，以便模拟分布结构。
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
// 创建并注册信道
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

// 注册远程类型
RemotingConfiguration.ApplicationName = "test";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Data));
});
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
// 创建并注册信道
TcpClientChannel channel = new TcpClientChannel();
```

```
ChannelServices.RegisterChannel(channel, false);

// 创建远程对象并调用其方法
RemotingConfiguration.RegisterActivatedClientType(typeof(Data),
"tcp://localhost:801/test");
Data data = new Data();
Thread.Sleep(3000);
data.Test();
}

static void Main()
{
Server();
Client();
}
}
}
```

输出：

```
2007-2-23 14:25:07
2007-2-23 14:25:10
```

输出结果证实远程对象在执行 new 语句时被创建。

Singleton 和 SingleCall 的区别

Singleton

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
```

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

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
/// <summary>
/// 远程类型
/// </summary>
public class Data : MarshalByRefObject
{
private Guid id = Guid.NewGuid();

public void Test()
{
Console.WriteLine(id.ToString());
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
// 创建新的应用程序域，以便模拟分布结构。
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
// 创建并注册信道
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);
```

```
// 注册远程类型
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.Singleton);
});
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
// 创建并注册信道
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);

// 注册远程对象
RemotingConfiguration.RegisterWellKnownClientType(typeof(Data),
"tcp://localhost:801/data");

// 创建远程对象并调用其方法
Data data1 = new Data();
Data data2 = new Data();

data1.Test();
data2.Test();
}

static void Main()
{
Server();
Client();
}
}
}
```

输出：

94cc586b-a24f-4633-9c88-c98a93246453

94cc586b-a24f-4633-9c88-c98a93246453

两个不同的代理对象引用了同一个远程对象，符合 Singleton 的定义。

SingleCall

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
/// <summary>
///  远程类型
/// </summary>
public class Data : MarshalByRefObject
{
private Guid id = Guid.NewGuid();

public void Test()
{
Console.WriteLine(id.ToString());
```

```
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
// 创建新的应用程序域，以便模拟分布结构。
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
// 创建并注册信道
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

// 注册远程类型
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.SingleCall);
});
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
// 创建并注册信道
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);

// 创建远程对象并调用其方法
Data data = (Data)Activator.GetObject(typeof(Data), "tcp://localhost:801/data");

data.Test();
```

```
data.Test();
}

static void Main()
{
Server();
Client();
}
}
}
```

输出：
6141870b-43db-4d2f-a991-73ae4097ffd3
8e28bf26-5aab-4975-a36a-da61a4bcb93e

四：生存期租约

Remoting 采取了一种称之为 "租约" 的机制来管理远程对象(Singleton、CAO)的生存期策略。每个应用程序域中都有一个租约管理器(LifetimeServices)，它负责管理所有参与生存期的远程对象租约。租约管理器定期检查所有租约以确定过期的租约时间，如果租约已过期，将向该对象发起人(Sponsor)的发送请求，查询是否有谁要续订租约，若没有任何发起人续订该租约，租约将被移除，该远程对象也会被删除等待垃圾回收器回收。如果远程对象被发起人多次续订租约或被客户端持续调用，其生存期可以比其生存期租约长得多。

所谓发起人 (Sponsor，MSDN 翻译为"主办方"，真别扭!) 就是一个或多个与远程对象关联，用于定义租约延长时间的对象。租约管理器通过回调发起人方法(ISponsor.Renewal)来查询是否续订租约。发起人需要继承自 MarshalByRefObject，且必须实现 ISponsor 接口。在 System.Runtime.Remoting.Lifetime 名字空间中，Framework 为我们提供了一个缺省实现 —— ClientSponsor。

租约参数

- 当参与生存期管理的远程对象被创建后，其租约被设置为 LifetimeServices.LeaseTime 或 ILease.InitialLeaseTime。
- 我们可以通过 ILease.CurrentLeaseTime 来检查对象租约过期的剩余时间。

- 客户端调用远程对象方法时，会发生隐式续订租约行为。当 CurrentLeaseTime 小于 RenewOnCallTime，则租约被设置为 RenewOnCallTime。
- 租约管理器每隔一定时间(LeaseManagerPollTime)检查一次租约列表，如果某租约过期则通知其发起人，询问是否进行续订。如发起人未能在 SponsorshipTimeout 时间内响应，则移除该主办方并调用另一主办方。如果没有其他主办方，则租约过期，且垃圾回收器将处置该远程对象。

租约过期试验

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Threading;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Lifetime;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
/// <summary>
///  远程类型
/// </summary>
public class Data : MarshalByRefObject
{
private Guid guid = Guid.NewGuid();

public void Test()
{
```

```
Console.WriteLine("{0} in {1}...", guid, AppDomain.CurrentDomain.FriendlyName);
Console.WriteLine((InitializeLifetimeService() as ILease).CurrentLeaseTime);
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
// 创建新的应用程序域，以便模拟分布结构。
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
// 设置缺省生存期
LifetimeServices.LeaseTime = TimeSpan.FromSeconds(1);
LifetimeServices.RenewOnCallTime = TimeSpan.FromSeconds(1);
LifetimeServices.LeaseManagerPollTime = TimeSpan.FromSeconds(1);
LifetimeServices.SponsorshipTimeout = TimeSpan.FromSeconds(1);

TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.Singleton);
});
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterWellKnownClientType(typeof(Data),
```

```
"tcp://localhost:801/data");

// 创建远程对象并调用其方法
for (int i = 0; i < 3; i++)
{
Data data = new Data();
data.Test();
Thread.Sleep(5000);
}
}

static void Main()
{
Server();
Client();
}
}
}
```

输出：

```
6ac6fcb9-6a5d-427b-92bb-cfdcb2265911 in server...
00:00:00.9899856
86a1716f-1ebd-47d2-af80-e501d20813b9 in server...
00:00:01
9a1039e0-fac7-4bbb-80c4-0bc7d5c9c2ff in server...
00:00:01
```

通过输出结果，我们发现 Singleton MBR 对象过期后重新创建。

初始化租约

1. 默认租约：用 LifetimeServices 设置所有对象的默认租约。

```
LifetimeServices.LeaseTime = TimeSpan.FromSeconds(1);
LifetimeServices.RenewOnCallTime = TimeSpan.FromSeconds(1);
```

```
LifetimeServices.LeaseManagerPollTime = TimeSpan.FromSeconds(1);
LifetimeServices.SponsorshipTimeout = TimeSpan.FromSeconds(1);
```

2. 自定义租约：重写 MarshalByRefObject.InitializeLifetimeService() 自定义对象租约。当 ILease.CurrentState 属性为 LeaseState.Initial 时，可以设置租约的属性。一旦设置，就不能直接更改它们。

```
public class Data : MarshalByRefObject
{
public override object InitializeLifetimeService()
{
ILease lease = (ILease)base.InitializeLifetimeService();
if (lease.CurrentState == LeaseState.Initial)
{
lease.InitialLeaseTime = TimeSpan.FromSeconds(1);
lease.SponsorshipTimeout = TimeSpan.FromSeconds(1);
lease.RenewOnCallTime = TimeSpan.FromSeconds(2);
}

return lease;
}
}
```

续订租约

续订租约的方式有：

- 隐式续订：在调用远程对象方法时发生。
- 显示续订：调用 ILease.Renew()。
- 发起租约：通过创建发起人(Sponsor)来延长租约。

在客户端调用 ILease.Renew() 显示续订租约，如果 CurrentLeaseTime 小于续订时间，则租约时间被设置为续订时间。

```
static void Client()
{
// 创建并注册信道
TcpClientChannel channel = new TcpClientChannel();
```

```
ChannelServices.RegisterChannel(channel, false);

// 注册远程对象
RemotingConfiguration.RegisterWellKnownClientType(typeof(Data),
"tcp://localhost:801/data");

// 创建远程对象并调用其方法
Data data = new Data();
(data.GetLifetimeService() as ILease).Renew(TimeSpan.FromSeconds(100));
}
```

五：信道

信道(Channel)是 Remoting 体系的承载平台，负责处理客户端和服务器之间的通讯，其内容包括跨域通讯、消息传递、对象编码等等。信道必须实现 IChannel 接口，根据通讯方向又分别提供了继承版本 IChannelReceiver 和 IChannelSender。Remoting 框架为我们提供了 IPC、TCP 以及 HTTP 的实现版本，当然我们还可以在网络上找到其他协议的实现版本。

```
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);
```

我们可以使用实用类 ChannelServices 来管理程序域内的信道，诸如注册、注销等等。程序域内可以同时使用多个信道，每个信道需提供唯一的名称，以便 ChannelServices 进行管理，同时信道会随程序域的退出自动销毁。

```
TcpServerChannel channel = new TcpServerChannel("tcp801", 801);
ChannelServices.RegisterChannel(channel, false);

IChannel c2 = ChannelServices.GetChannel("tcp801");
Console.WriteLine(Object.ReferenceEquals(channel, c2));

channel.StopListening(null);
channel.StartListening(null);

foreach (IChannel c in ChannelServices.RegisteredChannels)
```

```
{
Console.WriteLine(c.ChannelName);
}

ChannelServices.UnregisterChannel(channel);
```

信道内部包含由多个接收器(Sink)组成的接收链(Sink Chain)。接收器用于处理客户端和服务器之间的来往消息，诸如格式化程序接收器(FormatterSink)、传输接收器(TransportSink)或堆栈生成器接收器(StackBuilderSink)等等。每个接收器或实现 IClientChannelSink，或实现 IServerChannelSink。

Remoting 采用信道接收提供程序(Channel Sink Provider，实现 IClientChannelSinkProvider、IClientFormatterSinkProvider 或 IServerChannelSinkProvider 接口的对象) 来创建接收器，已有的提供程序包括 BinaryClientFormatterSinkProvider / BinaryServerFormatterSinkProvider、SoapClientFormatterSinkProvider / SoapServerFormatterSinkProvider。

在客户端接收链中第一个接收器通常是格式化程序接收器(IClientFormatterSink)，且必须实现 IMessageSink。代理通过信道接收提供程序找个该接收器，并通过接口方法将消息传递给链中所有的接收器，最后由传输接收器发送到服务器。同样，在服务器排在最后的接收器是格式化程序接收器和堆栈生成器接收器，分别执行反序列化和将将消息转换成相应的调用堆栈。

```
TcpServerChannel channel = new TcpServerChannel("tcp801", 801, new
BinaryServerFormatterSinkProvider());
ChannelServices.RegisterChannel(channel, false);

...

TcpClientChannel channel = new TcpClientChannel("tcp801", new
BinaryClientFormatterSinkProvider());
ChannelServices.RegisterChannel(channel, false);
```

只要看看 BinaryClientFormatterSinkProvider 和 BinaryClientFormatterSink 的代码就很容易理解如何使用提供者模型构造一个接收链了。

BinaryClientFormatterSinkProvider

```
public IClientChannelSink CreateSink(IChannelSender channel, string url, object
remoteChannelData)
{
IClientChannelSink sink1 = null;
if (this._next != null)
{
sink1 = this._next.CreateSink(channel, url, remoteChannelData);
if (sink1 == null)
{
return null;
}
}
SinkChannelProtocol protocol1 = CoreChannel.DetermineChannelProtocol(channel);
BinaryClientFormatterSink sink2 = new BinaryClientFormatterSink(sink1);
sink2.IncludeVersioning = this._includeVersioning;
sink2.StrictBinding = this._strictBinding;
sink2.ChannelProtocol = protocol1;
return sink2;
}
```

BinaryClientFormatterSink

```
public BinaryClientFormatterSink(IClientChannelSink nextSink)
{
this._includeVersioning = true;
this._channelProtocol = SinkChannelProtocol.Other;
this._nextSink = nextSink;
}

public IClientChannelSink NextChannelSink
{
get
{
return this._nextSink;
```

```
}
}
```

信道内的接收器是可 "插入的"，这意味着我们可以实现自己的接收器，并将其装配到信道接收链中。比如对消息进行加密，或者对数据流进行压缩等等。

六：异步调用

Remoting 的异步调用和单个应用程序域异步编程基本相同。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Threading;
using System.Security.Permissions;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Services;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
public delegate int AddHandler(int a, int b);

/// <summary>
/// 远程类型
/// </summary>
```

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

/// <summary>
///  服务器端代码
/// </summary>
static void Server()
{
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

RemotingConfiguration.ApplicationName = "test";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Data));
});
}

/// <summary>
///  客户端代码
/// </summary>
static void Client()
{
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterActivatedClientType(typeof(Data),
"tcp://localhost:801/test");

Data data = new Data();
```

```
AddHandler add = new AddHandler(data.Add);
IAsyncResult ar = add.BeginInvoke(1, 2, null, null);
ar.AsyncWaitHandle.WaitOne();
Console.WriteLine(add.EndInvoke(ar));
}

static void Main()
{
Server();
Client();
}
}
}
```

我们还可以为方法添加 OneWayAttribute 特性使其成为单向方法来完成类似的异步方法调用。

不过 [OneWary] 有一些条件限制：

1. 该方法无返回值和 out 或 ref 参数。
2. 该方法不能引发任何异常

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Threading;
using System.Security.Permissions;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;
```

```
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Services;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
public delegate void TestHandler();

/// <summary>
/// 远程类型
/// </summary>
public class Data : MarshalByRefObject
{
[OneWay]
public void Test()
{
Thread.Sleep(5000);
Console.WriteLine("Server:{0}", DateTime.Now);
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

RemotingConfiguration.ApplicationName = "test";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Data));
```

```
});
}

/// <summary>
///  客户端代码
/// </summary>
static void Client()
{
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterActivatedClientType(typeof(Data),
"tcp://localhost:801/test");

Data data = new Data();
data.Test();
Console.WriteLine("Client:{0}", DateTime.Now);
}

static void Main()
{
Server();
Client();
}
}
}
```

七：调用上下文

调用上下文(CallContext)提供了用于存储属性集的数据槽，可以让我们在调用服务器方法时将一些额外数据一并传送过去。当然，这些额外数据有点限制，就是必须要实现 ILogicalThreadAffinative 接口。调用上下文在应用程序域边界被克隆，其数据槽不在其他逻辑线程上的调用上下文之间共享。

我们利用这个特性写一个简单的身份验证例子。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Threading;
using System.Security.Permissions;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Services;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
/// <summary>
///  身份验证类型
/// </summary>
[Serializable]
public class Identity : ILogicalThreadAffinative
{
private string username;
private string password;

public Identity(string username, string password)
{
this.username = username;
this.password = password;
}
```

```
public string Username
{
get { return username; }
set { username = value; }
}

public string Password
{
get { return password; }
set { password = value; }
}
}

/// <summary>
/// 远程类型
/// </summary>
public class Data : MarshalByRefObject
{
public void Test()
{
// 执行身份验证
Identity identity = CallContext.GetData("identity") as Identity;
if (identity != null && identity.Username == "user1" && identity.Password == "pass")
{
Console.WriteLine("Test AppDomain:{0}", AppDomain.CurrentDomain.FriendlyName);
}
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
```

```
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

RemotingConfiguration.ApplicationName = "test";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Data));
});
}

/// <summary>
///  客户端代码
/// </summary>
static void Client()
{
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterActivatedClientType(typeof(Data),
"tcp://localhost:801/test");

//  传送身份验证数据
CallContext.SetData("identity", new Identity("user1", "pass"));
Data data = new Data();
data.Test();
}

static void Main()
{
Server();
Client();
}
}
}
```

数据槽中的数据可以双向传输，也就是说我们可以从服务器返回更多的信息给客户端。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Threading;
using System.Security.Permissions;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Services;

namespace Learn.Library.Remoting
{
public class RemotingTest2
{
[Serializable]
public class ServerTime : ILogicalThreadAffinative
{
private DateTime time;

public ServerTime(DateTime time)
{
this.time = time;
}

public DateTime Time
{
```

```
get { return time; }
}
}

/// <summary>
/// 远程类型
/// </summary>
public class Data : MarshalByRefObject
{
public void Test()
{
CallContext.SetData("ExInfo", new ServerTime(DateTime.Now));
Console.WriteLine("Test AppDomain:{0}", AppDomain.CurrentDomain.FriendlyName);
}
}

/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

RemotingConfiguration.ApplicationName = "test";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Data));
});
}

/// <summary>
/// 客户端代码
/// </summary>
```

```
static void Client()
{
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterActivatedClientType(typeof(Data),
"tcp://localhost:801/test");

Data data = new Data();
data.Test();
Console.WriteLine((CallContext.GetData("ExInfo") as ServerTime).Time);
}

static void Main()
{
Server();
Client();
}
}
}
```

八：元数据

.NET Remoting 基础结构需要正确的元数据，以便将一个应用程序域中的对象连接到另一个域中的对象。通常我们将包含远程类型的程序集同时发布到服务器和客户端，但这并不是一个好主意。有太多的原因阻止我们这么做：

1. 我们并不想客户端开发人员知道远程对象的内部细节，诸如私有成员内容等。
2. 我们不希望每次升级都更新客户端文件。

Soapsuds

Remoting 为我们提供了一个工具 "Soapsuds"。不要被它的名字所迷惑，它同样适用于二进制序列化的远程对象，因为我们只是用它来创建一个远程代理而已。假设远程类型存放在 RemoteLibrary 中，类型全名是 RemoteLibrary.Data。

RemoteLibrary.csproj

```
namespace RemoteLibrary
{
public class Data : MarshalByRefObject
{
public void Test()
{
Console.WriteLine("Test AppDomain:{0}", AppDomain.CurrentDomain.FriendlyName);
}
}
}
```

在 "Visual Studio 2005 命令提示" 窗口中用 Soapsuds 创建客户端代理类型。

```
c:/> Soapsuds -types:RemoteLibrary.Data,RemoteLibrary -gc
```

你会看到生成的 RemoteLibrary.cs 文件，打开看一下。

```
using System;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Metadata;
using System.Runtime.Remoting.Metadata.W3cXsd2001;
using System.Runtime.InteropServices;

namespace RemoteLibrary
{
[SoapType(...)]
[ComVisible(true)]
public class Data : System.Runtime.Remoting.Services.RemotingClientProxy
{
// Constructor
public Data()
{
}

public Object RemotingReference
```

```
{
get{return(_tp);}
}

[SoapMethod(...)]
public void Test()
{
((Data) _tp).Test();
}
}
}
```

将这个文件拷贝到客户端目录，并添加到项目中。开始编写客户端代码。

```
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterActivatedClientType(typeof(RemoteLibrary.Data),
"tcp://localhost:801/test");

RemoteLibrary.Data data = new RemoteLibrary.Data();
data.Test();
```

其实还可以使用 System.Runtime.Remoting.MetadataServices 名字空间中的相关类用代码来创建这些代理源码或者程序集。

```
using System.IO;
using System.Runtime.Remoting.MetadataServices;

ArrayList list = new ArrayList();
FileStream fs = new FileStream("test.xml", FileMode.OpenOrCreate);

// 将远程类型转换为 XML 架构。
MetaData.ConvertTypesToSchemaToStream(new Type[] { typeof(Data) }, SdlType.Wsdl,
fs);

// 将 XML 架构流中转换为代理源码文件。
fs.Seek(0, SeekOrigin.Begin);
```

```
MetaData.ConvertSchemaStreamToCodeSourceStream(true,
AppDomain.CurrentDomain.BaseDirectory, fs, list);

// 显示所生成的代理源码文件名。
foreach (string s in list) Console.WriteLine(s);

// 将生成的源码转换为程序集文件。
MetaData.ConvertCodeSourceStreamToAssemblyFile(list,
Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "test.dll"), null);
```

接口隔离

还有一个方式就是使用接口进行隔离，这样客户端就不会看到远程类型。我们继续用上面的例子做演示。

1. 创建接口类型项目。项目中包含接口和一个工厂类型，由工厂类型负责创建目标类型。(注意为避免循环引用，我们通过配置文件读取目标类型信息。)

RemoteInterface.projc

```
namespace RemoteInterface
{
public interface IData
{
void Test();
}

public class Factory : MarshalByRefObject
{
public IData NewData()
{
return
(IData)Activator.CreateInstance(Type.GetType(ConfigurationManager.AppSettings["data"
]));
}
```

```
}
}
```

配置文件

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
<add key="data" value="RemoteLibrary.Data,RemoteLibrary"/>
</appSettings>
</configuration>
```

2. 修改原类型使其实现 IData 接口，注意添加 RemoteInterface.dll 引用。

RemoteLibrary.csproj

```
namespace RemoteLibrary
{
public class Data : MarshalByRefObject, RemoteInterface.IData
{
public void Test()
{
Console.WriteLine("Test AppDomain:{0}", AppDomain.CurrentDomain.FriendlyName);
}
}
}
```

3. 在服务器端，除了 RemoteLibrary.Data，还要注册 RemoteInterace.Factory。

Server.csproj

```
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

RemotingConfiguration.ApplicationName = "test";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Factory));
RemotingConfiguration.RegisterActivatedServiceType(typeof(Data));
```

4. 将 RemoteInterface.dll 提供给客户端，添加引用后开始编码。

Client.csproj

```
RemotingConfiguration.RegisterActivatedClientType(typeof(Facade),
"tcp://localhost:801/test");
RemotingConfiguration.RegisterActivatedClientType(typeof(IData),
"tcp://localhost:801/test");

Factory factory = (Facade)Activator.CreateInstance(typeof(Factory), null);
IData data = factory.NewData();
data.Test();
```

这种方式稍显复杂，但从架构模式上来说要更好一些。它隔绝了目标类型和客户端的联系，服务器可以更灵活地变化和升级。

九：动态发布

使用动态发布有什么好处？

1. 避开 SAO 只能使用默认构造方法的限制。
2. 自主管理 SAO 的载入、卸载，以及其 URI。

RemotingServices

通过使用类 RemotingServices 提供的方法，我们可以很轻松实现这些目标。

- **Marshal**： 用于将 MarshalByRefObject 转换为 ObjRef 类的实例。
- **Connect**：客户端可以用该方法创建远程代理对象的实例。
- **Disconnect**：断开服务器远程对象与信道的连接。客户端代理在断开后调用任何方法都会触发 RemotingException。
- **Unmarshal**：接受 ObjRef 并从它创建一个客户端代理对象。这个方法很少被使用，因为多数情况下我们并不会直接将 ObjRef 显示传递给客户端，而是交由 Remoting 基础结构来处理。

ObjRef

ObjRef 是远程对象的可序列化表示，用于跨应用程序域边界传输对象引用。为对象创建 ObjRef 称为封送。可以通过信道将 ObjRef 传输到另一个应用程序域(可能在另一个进程或计算机上)。达到其他应用程序域后，需立即分析 ObjRef，以便为该对象创建一个代理(通常连接到实际的对象)。此操作称为拆收处理 (Unmarshaling)。在拆收处理过程中，分析 ObjRef 以提取远程对象的方法信息，并创建透明代理和 RealProxy 对象。在透明代理注册到公共语言运行库之前，将已分析的 ObjRef 的内容添加到透明代理中。

ObjRef 包含：描述所封送对象的 Type 和类的信息，唯一标识特定对象实例的 URI，以及有关如何到达对象所在的远程处理分支的相关通信信息。

动态发布示例

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Threading;
using System.IO;
using System.Security.Permissions;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Services;

namespace Learn.Library.Remoting
{
/// <summary>
///  远程类型
```

```
/// </summary>
public class Data : MarshalByRefObject
{
public void Test()
{
Console.WriteLine("Test AppDomain:{0}", AppDomain.CurrentDomain.FriendlyName);
}

public void Disconnect()
{
// 断开连接
RemotingServices.Disconnect(this);
}
}

public class RemotingTest2
{
/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);

// 创建远程对象实例，当然也可以使用非默认构造方法。
// 此方式类似于 SAO.Singleton 。
Data data = new Data();

// 封送远程对象。
ObjRef objRef = RemotingServices.Marshal(data, "data");
});
```

```
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);

// 连接服务器，并创建代理实例。
Data data = (Data)RemotingServices.Connect(typeof(Data), "tcp://localhost:801/data");

// 调用远程对象方法。
data.Test();

// 调用远程对象方法，断开连接。
data.Disconnect();

// 再次调用远程对象方法时，因连接已断开，将抛出 RemotingException。
data.Test();
}

static void Main()
{
Server();
Client();
}
}
}
```

十：追踪服务

.NET Remoting 的追踪服务使我们可以获取由远程结构发出的有关对象与代理的行为通知。追踪服务是可插入的，我们可以将一个或多个自定义跟踪处理程序注册到追踪服务中，当发生封送、取消封送或断开当前 AppDomain 中的对象或代理时，注册到中的每个追踪处理程序都将被远程处理调用。

创建自定义追踪处理程序很简单，实现 ITrackingHandler 接口，然后调用 TrackingServices.RegisterTrackingHandler() 将其实例注册到跟踪服务即可。追踪服务一般用于日志记录和调试。TrackingServices 实用类还可以注销(TrackingServices.UnregisterTrackingHandler)追踪处理程序，或查询(TrackingServices.RegisteredHandlers)所有的已注册追踪处理程序。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Threading;
using System.IO;
using System.Security.Permissions;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.CompilerServices;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Lifetime;
using System.Runtime.Remoting.Services;

namespace Learn.Library.Remoting
{
/// <summary>
/// 追踪服务
/// </summary>
public class TrackingHandler : ITrackingHandler
{
```

```
public void MarshaledObject(Object obj, ObjRef or)
{
Console.WriteLine("Marshaled: {0}. HashCode: {1}", obj.GetType(), obj.GetHashCode());
}

private void DumpChannelInfo(IChannelInfo info)
{
}

public void UnmarshaledObject(Object obj, ObjRef or)
{
Console.WriteLine("Unmarshaled: {0}. HashCode: {1}", obj.GetType(),
obj.GetHashCode());
}

public void DisconnectedObject(Object obj)
{
Console.WriteLine("Disconnected: {0}. HashCode: {1}", obj.GetType(),
obj.GetHashCode());
}
}

/// <summary>
///  远程类型
/// </summary>
public class Data : MarshalByRefObject
{
public void Test()
{
Console.WriteLine("Test AppDomain:{0}", AppDomain.CurrentDomain.FriendlyName);
}
}

public class RemotingTest2
{
```

```
/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
LifetimeServices.LeaseTime = TimeSpan.FromSeconds(1);
LifetimeServices.RenewOnCallTime = TimeSpan.FromSeconds(1);
LifetimeServices.SponsorshipTimeout = TimeSpan.FromSeconds(1);
LifetimeServices.LeaseManagerPollTime = TimeSpan.FromSeconds(1);

// 注册追踪服务
TrackingServices.RegisterTrackingHandler(new TrackingHandler());

TcpServerChannel channel = new TcpServerChannel(801);
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.Singleton);
});
}

/// <summary>
/// 客户端代码
/// </summary>
static void Client()
{
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterWellKnownClientType(typeof(Data),
"tcp://localhost:801/data");

Data data = new Data();
data.Test();
```

```
}

public static void Execute()
{
Server();
Client();
}
}
}
```

输出：

```
Unmarshaled: System.AppDomain. HashCode: 12036987
Unmarshaled: System.AppDomain. HashCode: 12036987
Unmarshaled: System.AppDomain. HashCode: 12036987
Marshaled: Learn.Library.Remoting.Data. HashCode: 38583594
Test AppDomain:server
Disconnected: Learn.Library.Remoting.Data. HashCode: 38583594
```

十一：事件

在 Remoting 中使用 Event 主要是为了实现 CallBack 机制，让服务器在接收到某个 "消息" 时，主动调用某个或多个客户端的方法。

我们先看一个例子。

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

namespace Learn.Library.Remoting
{
```

```
/// <summary>
/// 委托类型
/// </summary>
public delegate void TestHandler();

/// <summary>
/// 远程类型
/// </summary>
public class Data : MarshalByRefObject
{
public TestHandler OnTest;

public void Test()
{
Console.WriteLine("Test...(AppDomain:{0})", AppDomain.CurrentDomain.FriendlyName);
if (OnTest != null) OnTest();
}
}

public class RemotingTest2
{
/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
BinaryServerFormatterSinkProvider bin = new BinaryServerFormatterSinkProvider();
bin.TypeFilterLevel = TypeFilterLevel.Full;

TcpServerChannel channel = new TcpServerChannel("server", 801, bin);
ChannelServices.RegisterChannel(channel, false);
```

```
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.Singleton);
});
}

/// <summary>
///  客户端代码
/// </summary>
static void Client()
{
TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterWellKnownClientType(typeof(Data),
"tcp://localhost:801/data");

Data data = new Data();
data.OnTest += delegate
{
Console.WriteLine("OnTest...(AppDomain:{0})",
AppDomain.CurrentDomain.FriendlyName);
};
data.Test();
}

static void Main()
{
Server();
Client();
}
}
}
```

输出：

```
Test...(AppDomain:server)
OnTest...(AppDomain:server)
```

运行结果表明客户端事件方法 OnTest 被顺利执行。只不过结果有点问题，OnTest 是在服务器程序域内执行，这显然和我们设想服务器去通知客户端有所出入。这种方式实质上是将客户端委托方法一起序列化为消息传递到服务器端，然后在服务器应用程序域被执行，因此客户端是无法接收到所谓 "回调消息" 的。

要实现我们所需要的 Remoting Event，需要做如下步骤：

1. 采取所谓 Duplex 方式。也就是说在客户端和服务器同时启用 ServerChannel 和 ClientChannel，因此我们需要使用 HttpChannel 或 TcpChannel。
2. 客户端事件方法应该是一个继承自 MarshalByRefObject 类型的实例方法。因为服务器是通过创建客户端的 MBR SAO 对象来实现回调的。
3. 缺省情况下，Delegate 无法被序列化，因此我们需要将服务器的 Formatter.TypeFilterLevel 设置为 Full。

修改后的代码。

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

namespace Learn.Library.Remoting
{
/// <summary>
///  委托类型
/// </summary>
public delegate void TestHandler();

/// <summary>
///  远程类型
```

```
/// </summary>
public class Data : MarshalByRefObject
{
public TestHandler OnTest;

public void Test()
{
Console.WriteLine("Test...(AppDomain:{0})", AppDomain.CurrentDomain.FriendlyName);
if (OnTest != null) OnTest();
}
}

/// <summary>
/// 客户端远程类型
/// </summary>
public class ClientData : MarshalByRefObject
{
public void OnTestMethod()
{
Console.WriteLine("Test...(AppDomain:{0})", AppDomain.CurrentDomain.FriendlyName);
}
}

public class RemotingTest2
{
/// <summary>
/// 服务器端代码
/// </summary>
static void Server()
{
AppDomain server = AppDomain.CreateDomain("server");
server.DoCallBack(delegate
{
BinaryClientFormatterSinkProvider cbin = new BinaryClientFormatterSinkProvider();
BinaryServerFormatterSinkProvider sbin = new BinaryServerFormatterSinkProvider();
```

```
sbin.TypeFilterLevel = TypeFilterLevel.Full;

Hashtable properties = new Hashtable();
properties["port"] = 801;

TcpChannel channel = new TcpChannel(properties, cbin, sbin);
ChannelServices.RegisterChannel(channel, false);

RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.Singleton);
});
}

/// <summary>
///  客户端代码
/// </summary>
static void Client()
{
TcpChannel channel = new TcpChannel(802);
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterWellKnownClientType(typeof(Data),
"tcp://localhost:801/data");

Data data = new Data();
data.OnTest += new ClientData().OnTestMethod;
data.Test();
}

static void Main()
{
Server();
Client();
}
}
}
```

输出：
Test...(AppDomain:server)
Test...(AppDomain:Test.exe)

十二：配置文件

使用配置文件替代硬编码可使应用程序拥有更高的灵活性，尤其是对分布式系统而言，意味着我们可以非常方便地调整分布对象的配置。Remoting 的配置文件比较简单，详细信息可以参考 MSDN。

ms-help://MS.MSDNQTR.v80.chs/MS.MSDN.v80/MS.NETDEVFX.v20.chs/dv_fxgenref/html/52ebd450-de87-4a87-8bb9-6b13426fbc63.htm

下面是个简单的例子，包含了 SAO / CAO 的配置样例。

Server.cs

```
BinaryClientFormatterSinkProvider cbin = new BinaryClientFormatterSinkProvider();
BinaryServerFormatterSinkProvider sbin = new BinaryServerFormatterSinkProvider();
sbin.TypeFilterLevel = TypeFilterLevel.Full;

Hashtable properties = new Hashtable();
properties["port"] = 801;

TcpChannel channel = new TcpChannel(properties, cbin, sbin);
ChannelServices.RegisterChannel(channel, false);

RemotingConfiguration.RegisterWellKnownServiceType(typeof(Data), "data",
WellKnownObjectMode.Singleton);
RemotingConfiguration.ApplicationName = "test";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Data2));
```

Client.cs

```
TcpChannel channel = new TcpChannel();
ChannelServices.RegisterChannel(channel, false);
RemotingConfiguration.RegisterWellKnownClientType(typeof(Data),
```

```
"tcp://localhost:801/data");
RemotingConfiguration.RegisterActivatedClientType(typeof(Data2),
"tcp://localhost:801/test");

Data data = new Data();
data.Test();

Data2 data2 = new Data2();
data2.Test();
```

改成对应的配置文件，就是下面这个样子。

Server.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.runtime.remoting>
<application name="test">
<channels>
<channel ref="tcp" port="801">
<clientProviders>
<formatter ref="binary"/>
</clientProviders>
<serverProviders>
<formatter ref="binary" typeFilterLevel="Full" />
</serverProviders>
</channel>
</channels>
<service>
<wellknown mode="Singleton" type="Learn.Library.Remoting.Data, Learn.Library"
objectUri="data" />
<activated type="Learn.Library.Remoting.Data2, Learn.Library" />
</service>
</application>
</system.runtime.remoting>
</configuration>
```

Client.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.runtime.remoting>
<application>
<channels>
<channel ref="tcp">
<clientProviders>
<formatter ref="binary"/>
</clientProviders>
</channel>
</channels>
<client url="tcp://localhost:801/test">
<wellknown type="Learn.Library.Remoting.Data, Learn.Library"
url="tcp://localhost:801/data" />
<activated type="Learn.Library.Remoting.Data2, Learn.Library" />
</client>
</application>
</system.runtime.remoting>
</configuration>
```

Server.cs

```
RemotingConfiguration.Configure("server.config", false);
```

Client.cs

```
RemotingConfiguration.Configure("client.config", false);

Data data = new Data();
data.Test();

Data2 data2 = new Data2();
data2.Test();
```

如何定制 Sink 扩展.Net Remoting 功能

How to Build Custom Sinks to extend the Features of.Net Remoting Framework

关于 Channel, Sink (Channel Sink or Message Sink), Sink Chain and Channel Sink Provider 等.Net Remoting Framework 中一些基本概念，可以参考《信道、接收器、接收链和信道接受提供程序》。这里利用.Net Remoting Framework 内置的扩展特性，来定制 Remoting 流程，满足应用程序的特定需要。

.Net Remoting Framework 在 Client/Server 端同时实现了 Formatter Sink 和 Transport Sink，这是支持 Remote Object 调用的关键 Sink。对 Remote Object 的每一次调用都会产生消息，这些消息用于 Client 与 Server 之间的数据交换。远程调用的每一个消息都经过 Client/Server 的接受链（Sink Chain）。Sink 接受消息并处理消息，并传递消息到 Sink Chain 的下一个 Sink。

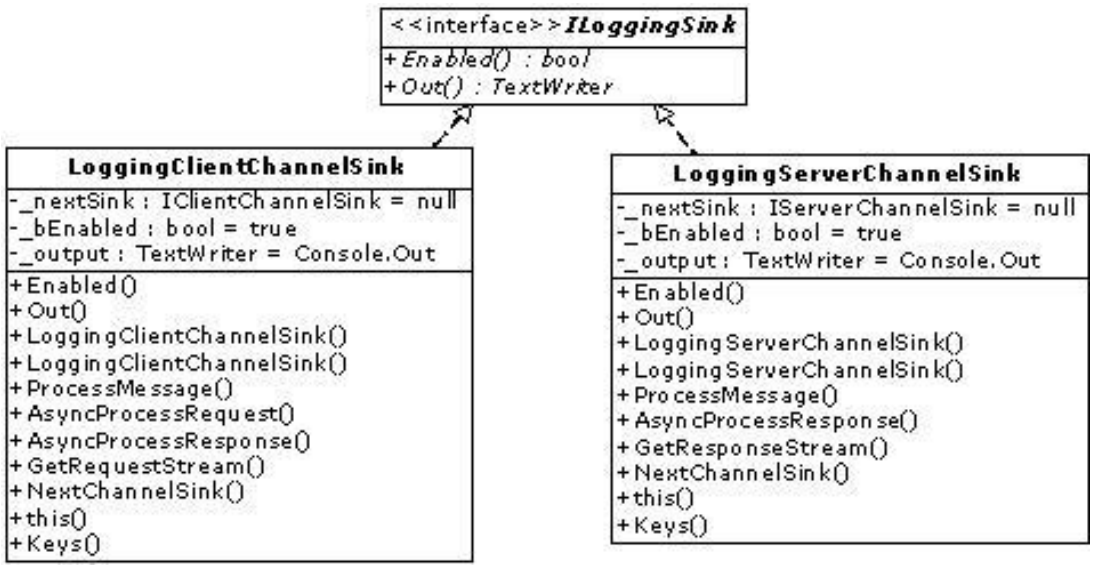
其中，Sink Chain 中的 Formatter Sink 用来序列化消息，以适于网络传输；或者反序列化消息，以适于 application 读取。.Net Remoting 缺省支持 SOAP Formatter Sink 和 Binary Formatter Sink，你也可以定制 Formatter 来扩展 Remoting 基础结构。

为了测试/编写 Custom Sink，首先需要建立一个.Net Remoting Application，包括 Client 端 application 和 Server 端 Remote Objects，并且确认运行正确，没有 bug。建立 Remoting Application 的过程就省略了。

下面 demo 分别在 Client 或 Server 端建立定制的 Sink，用来记录 LOG 调用消息，其实就是 SOAP 消息。

1，定制 Sink

下面是 Client/Server Custom Sink 类图，都实现公共的 ILoggingSink 接口，其中 Enable 属性设定是否记录 LOG 消息，Out 属性用来控制消息输出，如 Console, File 等。



以 Client 端的 Sink 为例：

class LoggingClientChannelSink : BaseChannelObjectWithProperties, IClientChannelSink, ILoggingSink

这里 LoggingClientChannelSink 实现了 2 个接口：IClientChannelSink, ILoggingSink，同时还继承 Abstract Class BaseChannelObjectWithProperties。

注解：

（1）ILoggingSink – 是自定义接口，Client/Server 端的 Custom Sink 都实现该接口。

（2）**IClientChannelSink** - 是 Custom Sink 必须实现的接口，IClientChannelSink 接口提供了 ProcessMessage()方法和 NextChannelSink 属性等等，是 Custom Sink 必需的。

（3）**BaseChannelObjectWithProperties** - Provides a base implementation of a channel object that wants to provide a dictionary interface to its properties. （英文比 MSDN 的中文好理解）

The derived class only needs to implement the Keys property and this[].

We don't have to write implementation of the interface from scratch. We can derive our custom channel sinks from an abstract class named BaseChannelObjectWithPropertis. We can use this abstract class to help implementation our custom channel sinks.

下面的 ProcessMessage()方法和 NextChannelSink 属性来自 IClientChannelSink 接口。通过实现 IClientChannelSink 接口的 ProcessMessage()方法来序列化消息到 Console 或者 File，其中 LoggingHelper Class 中 PrintRequest 和 PrintResponse 分别负责打印 Request/Response 消息。

```
public void ProcessMessage(IMessage msg,
    ITransportHeaders requestHeaders, Stream requestStream,
    out ITransportHeaders responseHeaders, out Stream responseStream)
{
    if (_bEnabled)
        LoggingHelper.PrintRequest(_output, requestHeaders, ref
requestStream);

    _nextSink.ProcessMessage(msg, requestHeaders, requestStream,
        out responseHeaders, out responseStream);

    if (_bEnabled)
        LoggingHelper.PrintResponse(_output, responseHeaders, ref
responseStream);
} // ProcessMessage
```

获取客户端接收器链中的下一个客户端信道接收器。

```
public IClientChannelSink NextChannelSink
{
    get { return _nextSink; }
}
```

2，编写 **Sink Provider**

Client 或 Server 端的 Sink Provider 分别需要实现 IclientChannelSinkProvider 和 IServerChannelSinkProvider 接口。

.Net Remoting Framework 实现 IClientChannelSinkProvider 的类有：BinaryClientFormatterSinkProvider（为二进制客户端格式化程序接收器提供程序提供实现）和 SoapClientFormatterSinkProvider（为客户端格式化程序接收器提供程序提供实现）。

以 Client 端的 Sink Provider 实现为例：

Client Sink Provider 实现 IclientChannelSinkProvider 接口，负责创建 Client Sink。

```
public class LoggingClientChannelSinkProvider : IClientChannelSinkProvider
{
    private IClientChannelSinkProvider _next = null;
    public LoggingClientChannelSinkProvider()
    {
    }

    public LoggingClientChannelSinkProvider(IDictionary properties, ICollection
providerData)
    {
    }

    public IClientChannelSink CreateSink(IChannelSender channel, String url,
        Object remoteChannelData)
    {
        IClientChannelSink nextSink = null;
        if (_next != null)
        {
            nextSink = _next.CreateSink(channel, url, remoteChannelData);
            if (nextSink == null)
                return null;
        }
        // Create a new LoggingClientChannelSink class instance, passing the next sink
in the sink chain as a parameter to the constructor. That enables the
LoggingClientChannelSink class to perform its processing and the pass the message data
onto the client's transport sink.
        return new LoggingClientChannelSink(nextSink);
    }

    public IClientChannelSinkProvider Next
    {
        get { return _next; }
        set { _next = value; }
    }
```

```
} // class LoggingClientChannelSinkProvider
```

其中 Next 属性用来获取或设置信道接收器提供程序（Channel Sink Provider）链中的下一个接收器提供程序。

CreateSink()方法用来创建接收器链（Sink Chain），当调用 CreateSink 方法时，它创建自己的信道接收器，将 CreateSink 调用转发给链中的下一个接收器提供程序（如果有），并确保下一个接收器和当前的接收器链接在一起。

3，设置 Client/Server 端配置文件

为了使用 Client/Server Sink Provider，必须在 configuration file 中进行设置。<clientProviders>和<serverProviders>元素向.Net Remoting Framework 提供了加载 Channel Sink Provider 所必需的类型信息。

这样，当 Client/Server application 在注册 Channel 时，读取 Configuration file 中 provider 配置信息，并创建 Custom Sink，这里为 Logging Sink。

（1）Client 端配置文件

```
<channels>
<channel ref="http">
<clientProviders>
<formatter ref="soap" />
<provider type="Logging.LoggingClientChannelSinkProvider, LoggingSink" />
</clientProviders>
</channel>
</channels>
```

Client 端通过 RemotingConfiguration.Configure("client.exe.config")装载 configuration 文件后，Client 端的 Sink Chain 为：

SoapClientFormatterSink -Next→
LoggingSink -Next→
HttpClientTransportSink

最后的 HttpClientTransportSink 缺省由 HTTP Channel 来实例化。

（2）Server 端配置文件

```
<serverProviders>
<provider ref="wsdl" />
<formatter ref="soap" />
<provider type="Logging.LoggingServerChannelSinkProvider, LoggingSink" />
</serverProviders>
```

如果在 Server 端的 Configruation File 中指定了<serverProviders>元素，而没有显式在<serverProviders>之中 SdlChannelSinkProvider，也是没有**<provider ref="wsdl" />**这一行，wsdl 在 machine.config 配置文件中有定义，就无法通过 URL 加上 “?WSDL”

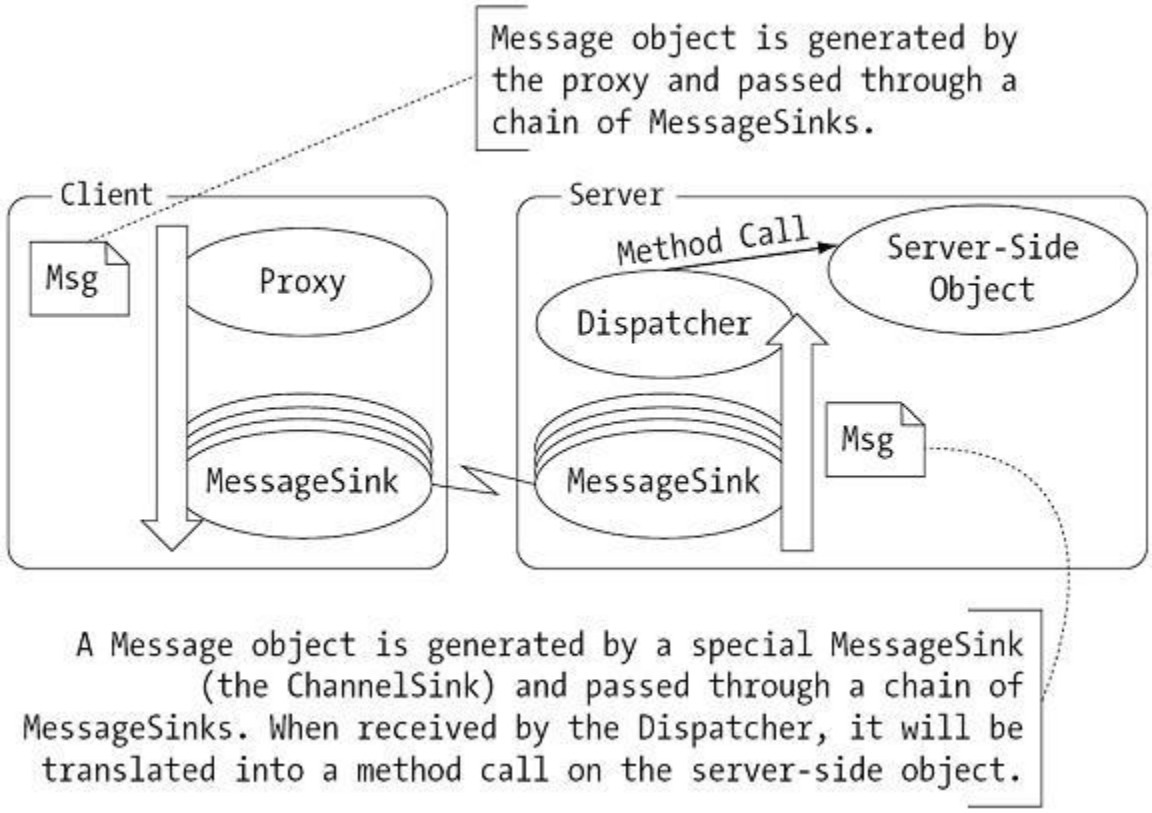
参数来产生 WSDL 描述，验证 Remote Objects 部署成功与否。因此，建议在 Server configuration file 中添加这一行。

根据上面的 configuration file，server 端的 Sink Chain 为：
HttpServerTransportSink – next →
SdlChannelSink – next →
SoapSeverFormatterSink – next →
LoggingSink – next →
BinaryServerFormatterSink – next →
DispatchChannelSink
******
Please note that source code is unavailable now. Sorry about that.

深度探索.Net Remoting 基础架构

1，.Net Remoting 基于如下 5 个核心对象类型：
Proxy：在 Client 端伪装为 Remote Objects 并转发对 Remote Objects 的调用。
Message：消息对象包含了执行 Remote Methods 调用的必要数据参数。
Message Sink/Channel Sink：在 Remote 调用中，Message Sink 允许定制消息处理流程，这是.Net Remoting 内置的可扩展特性。
Formatter：也是 Message Sink，用来序列化消息，已适于网络传输，如 SOAP。
Transport Channel：也是 Message Sink，用来传输序列化的消息到远程进程，如 HTTP。

下面是 Client application 对 Remote Object 的简要调用流程：



This figure is from the book named Advanced .Net Remoting.

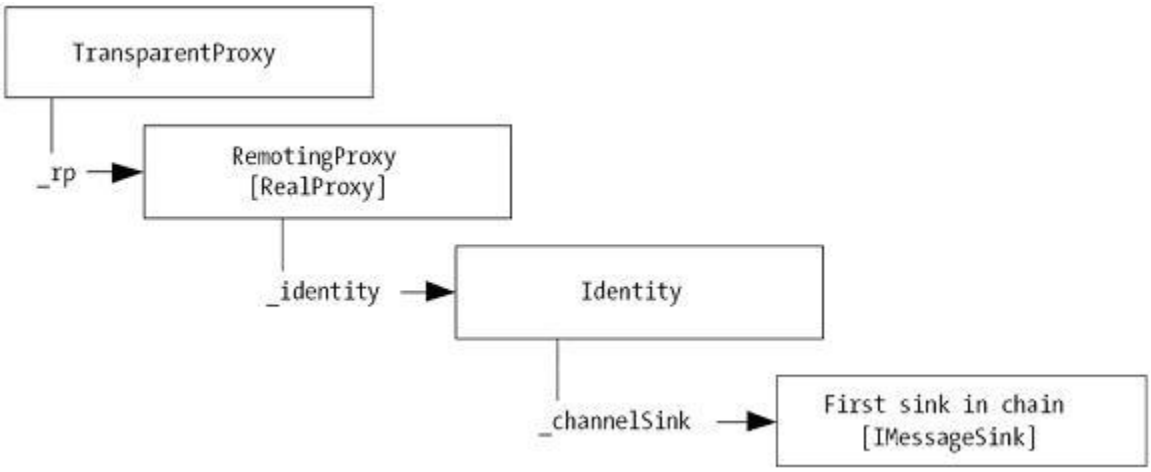
当访问 Remote Objects 时，Client 端 application 并不处理真实对象的引用，而是仅仅调用 Proxy 对象的方法。Proxy 对象提供与 Remote Objects 相同的接口，伪装成 Remote Objects。Proxy 对象自己并不执行任何方法，而是以消息对象（Message Object）的形式转发每一个方法调用给.Net Remoting Framework。

2，Proxy 对象的创建及其属性

当 Client 端通过 new 或调用 Activator.GetObject()方法来获取 Remote Object 引用时，.Net Remoting Framework 创建 2 个 Proxy Objects。第一个是 TransparentProxy 实例（from System.Runtime.Remoting.Proxies）。这个对象将从 new 关键字创建 Remote Objects 时返回。

无论何时当你访问 Remote Object 的方法时，实际上是调用 TransparentObject 对象的方法。这个 Proxy 拥有一个 RemotingProxy 对象的引用，RemotingProxy 继承抽象类 RealProxy。

在创建 Proxy 对象的过程中，需要引用 Client 端的 Messag Sink Chain，Sink Chain 的第一个 Sink 对象的引用保存在 RealProxy 对象的 Identity 属性。如下图所示：



通过 VS.Net 调试窗口，也可以观测到 Proxy 对象的一些重要属性：

remotingSqlHelper	{System.Runtime.Remoting.Proxies.__TransparentProxy}
[System.Runtime.Remoting.Proxies.__TransparentProxy	{System.Runtime.Remoting.Proxies.__TransparentProxy}
System.Object	{System.Runtime.Remoting.Proxies.__TransparentProxy}
_pInterfaceMT	0
_pMT	90469124
_rp	{System.Runtime.Remoting.Proxies.RemotingProxy}
[System.Runtime.Remoting.Proxies.RemotingProxy	{System.Runtime.Remoting.Proxies.RemotingProxy}
System.Object	{System.Runtime.Remoting.Proxies.RemotingProxy}
_defaultStub	2033461191
_defaultStubData	{-1}
_defaultStubValue	-1
_flags	3
_identity	{System.Runtime.Remoting.Identity}
_serverObject	<undefined value>
_tp	{System.Runtime.Remoting.Proxies.__TransparentProxy}
IdentityObject	{System.Runtime.Remoting.Identity}
System.Object	{System.Runtime.Remoting.Identity}
_channelSink	{System.Runtime.Remoting.Channels.BinaryClientFormatterSink}
[System.Runtime.Remoting.Channels.BinaryC	{System.Runtime.Remoting.Channels.BinaryClientFormatterSink}
System.Object	{System.Runtime.Remoting.Channels.BinaryClientFormatterSink}
_channelProtocol	Http
_includeVersioning	true
_nextSink	{System.Runtime.Remoting.Channels.Http.HttpClientTransportSi
_strictBinding	false
NextChannelSink	{System.Runtime.Remoting.Channels.Http.HttpClientTransportSi
NextSink	<error: an exception of type: {System.NotSupportedException}

通过在 Client/Server 的 MessageSink 中添加一些定制的 Sink 及其 Sink Provider，就可以扩展.Net Remoting Framework 的缺省功能，如 LOG 日志，加密，压缩等等特性。

信道、接收器、接收链和信道接受提供程序

为了扩展.Net Remoting，定制接收器（Sink）和信道接受提供程序（Channel Sink Provider），改变.Net Remoting 的缺省行为，需要先了解.Net Remoting 的相关概念及其运行机制。

下面先了解一些基本概念：

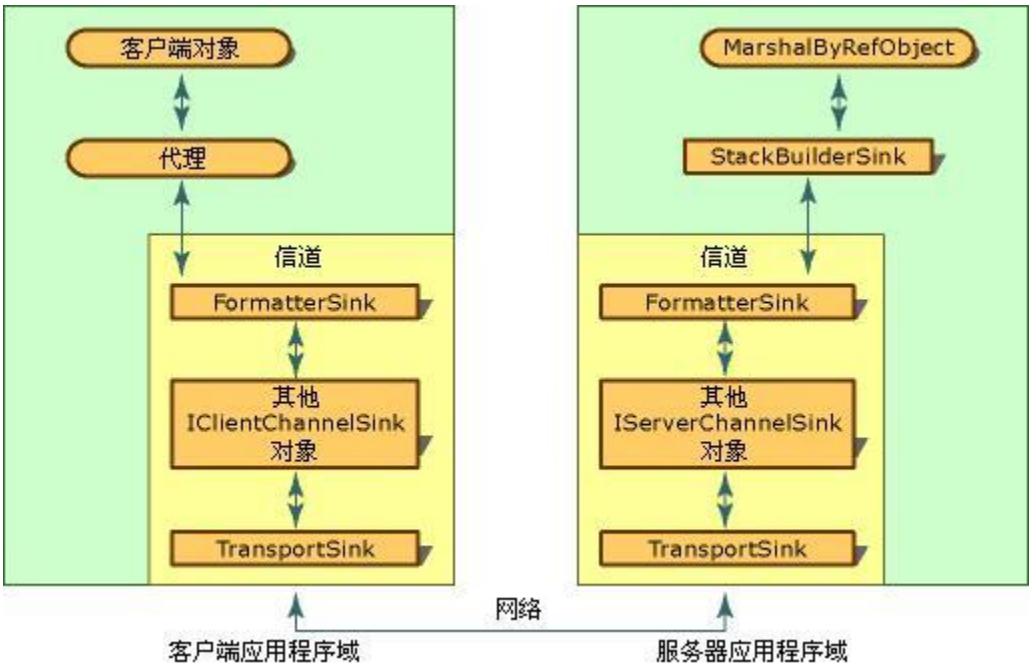
信道（Channel）一是跨远程处理边界（无论是在应用程序域、进程还是计算机之间）在应用程序之间传输消息的对象。信道可以在终结点上侦听入站消息，向另一个终结点发送出站消息，或者两者都可以。

信道必须实现 IChannel 接口，该接口提供诸如 ChannelName 和 ChannelPriority 这样的属性。专用于在特定端口上侦听特定协议的信道实现 IChannelReceiver，而专用于发送信息的信道实现 IChannelSender（注：IChannelReceiver 和 IChannelSender 接口都继承 IChannel 接口）。TcpChannel 和 HttpChannel 对象都实现这两种接口，因此它们可用于发送或接收信息。

.Net Remoting Framework 提供了 TcpChannel 和 HttpChannel 两个信道的实现。

在发送消息之前或接收消息之后，信道（Channel）沿着信道接收对象链（a chain of channel sink objects）发送每个消息。该接收链（sink chain）包含基本信道功能所需的**接收器（Sink）**（如格式化程序接收器 FormatterSink、传输接收器 TransportSink 或堆栈生成器接收器 StackBuilderSink），但是您可以自定义信道接收链以便用消息或流执行特殊任务。

信道接收链（Sink Chain）处理任何发送到应用程序域或从应用程序域发送的消息。此时，您只有消息，但是您可以任意操作该消息，而后面的处理将使用您在处理之后返回给系统的消息。这是实现日志记录服务、任何种类的筛选器或者客户端或服务器上的加密或其他安全措施的理所当然的位置。以下插图显示基本信道接收链的结构。



每个**信道接收器（Channel Sink）**都处理流，然后将流传递到下一个信道接收器，这意味着您的接收器之前或之后的对象应当知道该如何处理传递给它们的流。（Sink/Channel Sink 是同一个描述。）

StackBuilderSink 对象是服务器上远程对象前的最后一个消息接收器。

每个信道接收器（Channel Sink）或者实现 IClientChannelSink，或者实现 IServerChannelSink。客户端上的第一个信道接收器还必须实现 IMessageSink。它通常实现 IClientFormatterSink（它从 IMessageSink、IChannelSinkBase 和

IClientChannelSink 继承），并被称作格式化程序接收器（Formatter Sink），因为它将传入的消息转换为流（IMessage 对象）。

信道接收提供程序（Channel Sink Provider）—（实现 IClientChannelSinkProvider、IClientFormatterSinkProvider 或 IServerChannelSinkProvider 接口的对象）负责创建远程处理消息所流过的信道接收器（Channel Sink）。当远程类型被激活后，将从信道（Channel）中检索信道接收提供程序（Channel Sink Provider）；然后在该接收提供程序上调用 CreateSink 方法以检索链上的接收器中的第一个 Sink。

信道接收器（Channel Sink）还负责在客户端和服务器之间传输消息。信道接收器也链接在一起而形成一个链。当在接收提供程序上调用 CreateSink 方法时，该方法应该执行以下操作：

- 创建它自己的信道接收器。
- 在链中的下一个接收提供程序上调用 CreateSink。
- 确保下一个接收器和当前的接收器链接在一起。
- 将其接收器返回到调用方。

信道接收器负责将在它们上面进行的所有调用转发到链中的下一个接收器，并且应当提供用于存储对下一个接收器的引用的机制。

自定义信道接收器（Custom Channel Sinks）

自定义信道接收器被插入到格式化程序接收器（FormatterSink）和最后一个传输接收器（TransportSink）之间的对象链中。

传输接收器（TransportSink）

传输接收器是客户端上的链中最后一个接收器和服务器端上的链中第一个接收器。除了传输序列化的消息，传输接收器还负责将标头（Header）发送到服务器并在调用从服务器返回时检索标头和流。这些接收器内置在信道中，并且无法扩展。

简要总结.Net Remoting Infrastructure 机制：

1，代理对象（Proxy）负责转发对 Remote Objects 的调用。
2，消息对象（Message Objects）用来调用 Remote Methods 的数据。
3，信道接收器（Sink/Channel Sink）用来远程方法调用（Remote method calls）处理消息。
4，信道接收提供程序（Channel Sink Provider）—一般用来将接收器（Sink）插入到信道接收链（Sink Chain）中。
5，格式化程序接收器 FormatterSink—用来序列化/反序列化消息格式，进行传递。
6，传输接收器 TransportSink—用来在进程或 AppDomain 之间传递序列化的消息。

Reference:

1, MSDN, .NET Framework 开发员指南，接收器和接收链

灵活管理 Remote Objects 生存期（lifetime）

.Net Remoting Framework 提供了一套完整的机制来管理 Server 端的 Remote Objects 的生存期。关于这方面的详细信息，请参考 Wayfarer 的 blog, Microsoft .Net Remoting[高级篇]之一：Marshal、Disconnect 与生命周期以及跟踪服务（http://www.cnblogs.com/wayfarer/archive/2004/08/05/30437.aspx ），讲解得非常清楚。

.Net Remoting Framework 支持通过配置文件，如 web.config，来管理 Remote Objects 得生存期。如下所示（MSDN）：

```
<lifetime
leaseTime="leasetime"
sponsorshipTimeout="sponsorshipTimeOut"
renewOnCallTime="renewOnCallTime"
leaseManagerPollTime="pollTime"
/>
```

参数简单解释：

leaseTime — 指定该应用程序的初始租约时间。默认的 leaseTime 为 5 分钟。

SponsorshipTimeout — 指定租约管理器得到租约到期通知时等待主办方响应的时间。如果主办方未能在指定时间内响应，则由垃圾回收器处置该远程对象。默认的 sponsorshipTimeout 为 2 分钟。

RenewOnCallTime — 指定对象上每个函数调用的租约时间的延长时间。默认的 renewOnCallTime 为 2 分钟。

LeaseManagerPollTime — 指定租约管理器在检查到期租约后休眠的时间。默认的 leaseManagerPollTime 为 10 秒。（注：整个 application 共用一个租约管理器 LeaseManager）

虽然使用 configuration 文件很简单，但是这些设置会对 application 内的所有 Remote Objects 起作用，不管你乐意还是不乐意。

这里主要是推荐 **Ingo Rammer《Advanced .Net Remoting》**中提出的管理 Remote Objects 生存期的方法，通过扩展 MarshalByRefObject 类来实现灵活调整 Remote Objects 的生存期。本人觉得在实际应用中蛮有价值，并且非常简单。通过扩展 MarshalByRefObject 类，可以很好的解决上述问题。

1，扩展 MarshalByRefObject 类

这样，你不需要让每一个 Remote Objects 重载 MarshalByRefObject.InitializeLifetimeService()方法，并且今后还可以根据需要在 ExtendedMBRObject 类中添加一些通用的功能。

用法：application 内所有 MBR 的 Remote Objects 继承 ExtendedMBRObject，而不是直接继承 MarshalByRefObject 类。

简单说明：ExtendedMBRObject 重载 MarshalByRefObject.InitializeLifetimeService() 方法，按 Remote Objects 的名称来检查 configuration 文件中 appSetting 的设置，因此不同的 Remote Object 可以有不同的生存期参数的设置。如果在 appSetting 的没有该 Remote Object 生存期参数的设定，则采用 Remoting 提供的默认值。如果 LeaseTime 设置为 **infinity**，则 Remote Object 生存期为无限时间。

Source code 如下（细节方面稍有改动）：

==========

```
using System;
using System.Configuration;
using System.Runtime.Remoting.Lifetime;

namespace ComponentHost
{
    public class ExtendedMBRObject: MarshalByRefObject
    {
        public override object InitializeLifetimeService()
        {
            string myName = this.GetType().FullName;

            string leasetime =
                ConfigurationSettings.AppSettings[myName + ".LeaseTime"];

            string renewoncall =
                ConfigurationSettings.AppSettings[myName + ".RenewOnCallTime"];

            string sponsorshiptimeout =
                ConfigurationSettings.AppSettings[myName + ".SponsorShipTimeout"];

            if (leasetime !=null &&
leasetime.ToLower().Trim() == "infinity")
            {
                return null;
            }
            else
            {
                ILease tmp = (ILease) base.InitializeLifetimeService();
                if (tmp.CurrentState == LeaseState.Initial)
                {
                    if (leasetime != null)
                    {
                        tmp.InitialLeaseTime =
                            TimeSpan.FromMilliseconds(Double.Parse(leasetime));
                    }
```

```
                    if (renewoncall != null)
                    {
                        tmp.RenewOnCallTime =
                            TimeSpan.FromMilliseconds(Double.Parse(renewoncall));
                    }

                    if (sponsorshiptimeout != null)
                    {
                        tmp.SponsorshipTimeout =
                            TimeSpan.FromMilliseconds(Double.Parse(sponsorshiptimeout));
                    }

                }
                return tmp;
            }
        }
    }
}
```

2，设置 Configuration 文件（这里以 Web.config 为例）

针对需要调整默认生存期的 Remote Objects 进行配置即可。

```
<appSettings>
<add key="MyNamespace.MyRemoteObject.LeaseTime" value="5000" />
<add key="MyNamespace.MyRemoteObject.RenewOnCallTime" value="1000" />
<add key="MyNamespace.InfinitelyLivingSingleton.LeaseTime" value="infinity" />
</appSettings>
```

Reference:

1. Ingo Rammer, Advanced .Net Remoting.
2. Wayfarer, Microsoft .Net Remoting[高级篇]之一：Marshal、Disconnect 与生命周期以及跟踪服务, http://www.cnblogs.com/wayfarer/archive/2004/08/05/30437.aspx
3. MSDN

深度管理 Remote Objects 的生存期

在《**灵活管理 Remote Objects 生存期（lifetime）**》一文中，提及了 Remote Objects 生存期管理的一些基本方面，已经可以满足一般基于.Net Remoting 的应用。如果你觉得那些关于 Remote Objects 的生存期管理机制还不满足需求，则可以考虑实现 Client 端或 Server 端的 Sponser 对象。

这里将深入讨论 Remote Objects 生存期管理的 **Sponsor 机制**，Sponsorship is confusing, but also powful。（MSDN 译：Sponsor – 主办方， Lease — 生存期租约，LeaseManager —租约管理器）

当 Remote Object 创建时，Sponsor 向 Remote Object 提出注册。当 Remote Object 的生存期 TTL 要结束时，LeaseManager 负责与注册的 Sponsor 联系。如果 Sponsor 返回一个时间段 TimeSpan，通知 Remote Object 的生存期延长指定时间段；如果 Sponsor 返回 TimeSpan.Zero，则 Remote Object 结束其生存状态。

Sponsor 对象本身是一个 MarshalByRefObject 对象，并且实现 ISponsor 接口。另外，Sponsor 对象可以存放在 Remote Object 所在的 Server 端、其他的 Server 端、或者是 Client 端 application，但是必须满足.Net Remoting Framework 可以访问到 Sponsor 对象。（注意：如果 Sponsor 存放在 Client 端，并且 Client 在 firewall 后面，Server 端的 LeaseManager 就无法访问到 Sponsor 了。）

一个典型的 Sponsor 对象如下：

```
public class MySponsor: MarshalByRefObject, ISponsor
{
private bool NeedsRenewal()
{

// check some internal conditions
return true;
}

public TimeSpan Renewal(System.Runtime.Remoting.Lifetime.ILease lease)
{
if (NeedsRenewal())
{
return TimeSpan.FromMinutes(5);
}
else
{
return TimeSpan.Zero;
}
}
}
```

一般而言，对于 CAO（客户端激活对象）对象，Sponsor 对象位于 Client 端。对于 SAO（服务端激活对象）则相反。下面分别了解两种情况的 Sponsor 对象：

1. Client 端 Sponsor

为了让 Remote Objects 能够与 Client 端的 Sponsor 打交道，需要在 Client 配置文件中 Channel 设置 port 属性。如果没有这个属性，Channel 就无法接受来自 Server 的回调。

<channel ref=”http” port=”0”>

Port=”0”表示允许.Net Remoting Framework 选择Client 端任一空闲的port。

另外，我们知道在调用CAO远程对象时，还需要为配置文件的client 项指定url 属性。

注册Sponsor对象，避免Remote Objects过早的释放（Client 端application 部分示例代码）：

```
string filename = "client.exe.config";
RemotingConfiguration.Configure(filename);
// 实例化CAO 对象
SomeCAO cao = new SomeCAO();
// Get an object’s LifetimeService, which will be an ILease object.
ILease le = (ILease) cao.GetLifetimeService();
MySponsor sponsor = new MySponsor();
// Register the sponsor with the server object’s lease
le.Register(sponsor);
…调用Remote method 或处理其他事情
// Unresister the sponsor with the server objects
le.Unregister(sponsor);
```

当application 准备让server 释放CAO 对象实例时，application 通过告诉sponsor对象停止更新CAO 对象实例的TTL。通常情况下，application 通过调用Lease.Unregister（）方法来取消sponsor对象的注册，这样CAO 对象实例就不会与sponsor联系了。

注意：当你决定使用client 端的sponsor时，需要确保server 可以访问到client，这样client 就不能在firewall或proxy 后面。

2. Server端 Sponsor

Server端的Sponsor对象一般用来管理SAO对象的生存期，在实现方式上基本与Client端Sponsor对象一致。

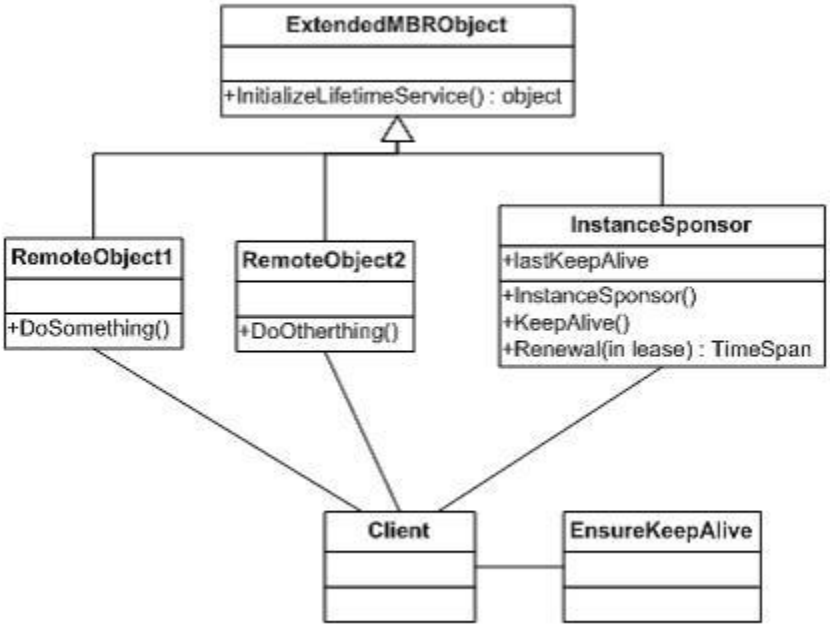
但是，需要注意的是Remote sponsor对象也是MarshalByRefObject对象，因此也和Remote Objects一样，需要管理其生存期。一般情况下，我们需要Server端sponsor对象与Client端application保持相同的生存期，当Client端application关闭时，要求尽快释放Server端sponsor对象。

Ingo Rammer《Advanced .Net Remoting》中提出了一种方案来实现Server端sponsor对象。

(1) 通过Client端后台线程EnsureKeepAlive对象不断调用sponsor.KeepAlive()方法，确保sponsor对象存活。

(2) 为了防止Client端application意外关闭或出现网络连接故障,无法正确调用Unregister()方法来取消注册，造成Remote Objects和sponsor对象一直存活。为了解决这一潜在问题，在sponsor.KeepAlive()方法中记录更新lastKeepAlive的时间，并在sponsor.Renewal()方法判断renew请求是否在指定的时间间隔内，如果是，则返回指定的TTL时间段，反之，则

返回 TimeSpan.Zero。



注意 Server 和 Client 端主要类如上图所示，Source Code 请参考《Advance .Net Remoting》Chapter 6（Remote Object 和 sponsor 可以为 CAO 或 SAO 对象）。

Reference:

1. Ingo Rammer, Advanced .Net Remoting.
2. Rickie, 《灵活管理 Remote Objects 生存期（lifetime）》

.Net Remoting 中 Remote Server 的 Port 占用/释放问题

这一问题一般出现在 Console application/Windows Service 承载 Remote Objects 时，要求 Remote Server 指定特定 port。IIS 在承载 Remote Objects 并不需要指定特定 port，因此一般不会出现 SocketException 异常信息。

1. 启动承载 Remote Objects 的 Console application，发现指定的 port 状态为：LISTENING。显然 Remote Server 开始监听该 port，可以接受 Client 端的请求。

2. 在关闭承载 Remote Objects 的 Console application 后，发现指定的 port 状态为：TIME_WAIT。

如果现在启动该 Console application，就会抛出如下异常：
An unhandled exception of type 'System.Runtime.Remoting.RemotingException' occurred in mscorlib.dll

Additional information: Remoting configuration failed with the exception System.Reflection.TargetInvocationException: Exception has been thrown by the target of an invocation. ---> System.Net.Sockets.SocketException: Only one usage of each socket address (protocol/network address/port)
is normally permitted

SocketException 异常：每一个 socket address (protocol/network address/port) 只能有一个使用。

3. 个人观点

关于 TIME_WAIT 状态，这是 windows 系统设计的，防止来自旧的连接（old connection）的剩余 packets 干扰新的连接（new connection）。因此，默认会等待 4 分钟，让那些剩余的 packets 丢弃掉。

因此，不要试图去解决这一问题。等待 4 分钟左右的时间，确认该 port 确定已释放，再启动 Remote Server，如 Console application/Windows Service 等。

.Net Remoting 配置文件的用法

.NET Remoting configuration files allow you to specify parameters for most aspects of the remoting framework. These files can define tasks as simple as registering a channel and specifying a Type as a server-activated object, or can be as complex as defining a whole chain of IMessageSinks with custom properties.
通过.Net Remoting 配置文件可以为 Remote Objects 设定许多参数，如 Channel、SAO 服务端激活对象类型（Singleton/SingleCall）等等，方便以后在不用修改代码或重新编译的情况下，改变 Remote Objects 的行为。

1，如下是 Server 端典型的 Remoting 配置文件：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.runtime.remoting>
<application>
<channels>
<channel ref="http"/>
</channels>
<service>
<wellknown mode="Singleton"
type="ComponentHost.CustomerManager, ComponentHost"
objectUri="CustomerManager.soap" />
</service>

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

```
</configuration>
```

（1）当 Remote Objects 部署在 Console/Windows Form、Windows Services 下时（上面的配置文件 channel 需要设置 port 属性），相应 Server 端声明 Remote Objects 的代码可以简化为：

```
string filename = "server.exe.config";
RemotingConfiguration.Configure(filename);
```

（2）如果 Remote Objects 部署在 IIS 时，根本就不需要任何代码声明。但是需要将上述配置文件命名为：web.config，并且将 Remote Objects 的 DLL 文件安置在 web application 的 BIN 文件夹。

一般在实际应用中，基本上将 Remote Objects 部署在 IIS 环境中，好处是（I）不需要编写额外的代码；（II）只要启动机器，远程对象就启动了。不需要你半夜三更跑到公司去登录，然后启动发生故障的远程服务；（III）容易与 IIS 认证服务进行集成；（IV）可能还有更多优点，我现在没有想到。

（3）如果需要声明多个远程对象，只需要在<service>与</service>之间添加相应的 Remote Objects 配置信息即可。

（4）另外需要注意 type 属性为：<namespace>.<class>, <assembly>

2，如下是 Client 端典型的配置文件：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.runtime.remoting>
<application>

<client>
<wellknown type="ComponentHost.CustomerManager, RemotingTest"
url="http://localhost/ComponentHost/CustomerManager.soap" />
</client>

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

要注意 type 属性的设定：<namespace>.<class>, <assembly>
如果 Client 通过 SoapSuds 产生 Remote Objects 的元数据 assembly，或者是 Shared Assembly（如 Interface 或 Abstract Class），这里<assembly>则为上述 assembly 的名称。
如果是通过 SoapSuds 产生 Source code，则<assembly>为 Client 应用程序名（无 exe 后缀）。

同时，Client 端 application 调用 Remote Objects 时，可以省掉：注册通道、Activator.GetObject()/RemotingConfiguration.RegisterActivatedServiceType()等代码，取而代之的代码为：

```
string filename = "clientApplication.exe.config";
RemotingConfiguration.Configure(filename);
```

下面通过 new 来创建 Remote Object 实例。

3，标准的.Net Remoting Configuration 配置文件

MSDN 中有.Net Remoting Configuration file 中全部元素/属性的完整的详细说明，需要的时候再查阅了。一般情况下，知道下面这些属性就够用了。

```
<configuration>
<system.runtime.remoting>
<application>
<lifetime /> －－ 配置 Remote Objects 生存期的信息
<channels /> －－ 配置与远程对象进行通信的信道
<service />
<client />
</application>
</system.runtime.remoting>
</configuration>
```

简单说明：

（1）<service> －－ 仅在 Server 端配置

```
<service>
<wellknown /> －－ 配置要发布的 SAO（已知）对象的信息
<activated /> －－ 配置要发布的 CAO 客户端激活对象的信息
</service>
```

（2）<client> －－ 仅在 Client 端配置，与 Server 端<service>对应

```
<client>
<wellknown />
<activated />
</client>
```

When using CAOs, the <client> property has to specify the URI to the server for all underlying <activated> entries.

Note：When using CAOs from more than one server, you have to create several <client> properties in your configuration file.

当调用 CAO 远程对象时，必须设定<client>的 url 属性。如果 CAO 来自不同的 Server，则需要在配置文件中定义多个<client>。如下所示：

```
<client url="http://localhost/MyServer>
<activated type="Server.MyRemote, Client" />
</client>
```

4，定制 Client/Server Channel 元素

（1）Client Side

```
<channel ref="http">
<clientProviders>
    <formatter ref="binary" />
</clientProviders>
</channel>
```

其中，formatter ref=”binary” or “soap”。formatter ref 指要在通道上发送的消息格式，在此示例中为二进制，以增强性能。

（2）Server Side

```
<channel ref="http">
<serverProviders>
    <provider ref="wsdl" />
    <formatter ref="binary" typeFileterLevel="Full" />
    <formatter ref="soap" typeFileterLevel="Full" />
</serverProviders>
</channels>
```

typeFilterLevel 表示当前自动反序列化级别，支持的值包括 Low（默认值）和 Full。

Reference:

1. Ingo Rammer, Advanced .Net Remoting.
2. MSDN

Message Queuing

消息队列(MSMQ) 作为一种系统服务，为我们提供了非常好的分布操作机制。但不知道什么原因，我很少看到有使用该技术的项目。MSMQ 究竟有什么好处呢？

- **脱机操作**：发送或者接收方可以使用消息队列作为局部或全局缓存，在另一方脱机的情况下发送或接受数据。这个应用在分布体系中很重要，可以避免某应用服务器当机造成整个业务系统瘫痪，由于数据被存储在队列中，瘫痪的服务器重启依然可以继续业务处理，也避免数据丢失造成的业务空洞。举个例子，在电子商务系统中，我们将创建订单和处理订单用消息队列连接起来，一来可以避免处理订单时间过长造成的等待，二来也不会因为处理订单系统故障而造成客户无法下达订单。
- **分解耦合**：在项目的第一个版本开发中，我们往往会将多个逻辑或应用放到一起。随着业务的增长，我们不得不将这些逻辑分离成多个应用服务器(AppServer)，来达到性能重构的目的。使用消息队列作为分解耦合的手段有个好处就是，被分解的应用无需关心其关联应用是被部署到本机还是其他服务器上。
- **应用隔离**：我们只所以使用 ORM/DAO 一个很重要的原因就是数据库无关性，便于在不同类型的存储服务器间迁移。使用消息队列隔离应用，每个应用都无需知道其他应用的位置和细节，彼此之间也不存在引用和依赖关系。在约定数据对象类型后，每个应用都可以独自变化和升级。
- **多播应用**：MSMQ 3.0 提供的这个功能，让我们开发可插入的分布应用时，可以不再使用单向的链式处理结构。将消息同时广播给不同的应用，很大程度上提高了处理性能，每个应用服务器都可以第一时间处理消息，无需等待任何其他应用完成。

这只是我个人对消息队列的一点浅显理解，有关其更详细信息，可以参考 MSDN。

.net+msmq 快速访问数据库

发布时间：2003.02.14 10:09 来源：赛迪网 作者：张悦

msmq 是微软消息队列的英文缩写。那么什么是消息队列？消息队列是 Windows 2000（nt 也有 msmq，win95/98/me/xp 不含消息队列服务但是支持客户端的运行）操作系统中通讯的基础，也是用于创建分布式、松散连接通讯应用程序的开发工具。这些应用程序可以通过不同种类的网络进行通讯，也可以与脱机的计算机通讯。消息队列分为用户创建的队列和系统队列，用户队列分为：

·“公共队列”在整个可传递消息的“消息队列”网络中复制并传输，并且有可能由网络连接的所有站点访问。

·“专用队列”不在整个网络中发布。相反，它们仅在所驻留的本地计算机上可用。专用队列只能由知道队列的完整路径名或标签的应用程序访问。

·“管理队列”包含确认在给定“消息队列”网络中发送的消息回执的消息。指定希望 MessageQueue 组件使用的管理队列。

·“响应队列”包含目标应用程序接收到消息时返回给发送应用程序的响应消息。指定希望 MessageQueue 组件使用的响应队列。

系统队列分为：

·“日记队列”可选地存储发送消息的副本和从队列中移除的消息副本。

·“死信队列”存储无法传递或已过期的消息的副本。

·“专用系统队列”是一系列存储系统执行消息处理操作所需的管理和通知消息的专用队列。

现在大家对消息队列有了简单的了解后，就要使用 msmq 进行软件开发需要安装 msmq。安装完后就该进入实际的开发阶段。先打开 vs.net ide 中的“服务起资源管理器”展开你想建立消息队列的计算机名，再展开“消息队列”，右击它，在弹出菜单中选择“新建”建立一个新的消息队列，并为它指定一个名字，这个名字可以随意。也可以通过编程来完成，代码如下：

```
system.Messaging.MessageQueue.Create(".\Private$\MyPrivateQueue")'建立专用队列
System.Messaging.MessageQueue.Create("myMachine\MyQueue")'建立公共队列
```

其实我认为使用哪种方法并不重要，重要的是搞清楚专用队列和公共队列的差别（其他队列不是必须的）。在本例中是通过“服务器资源管理器”分别在服务器上建立了专用队列和公共队列。

程序功能：本程序分为两部分，包括服务器程序（安装在 sql server 服务器上）和客户端程序。客户端的作用是用来编写 t-sql 语句并将 t-sql 语句放在消息中，并将消息发送到 sql server 服务器上的消息队列中去。服务器程序检查指定的消息队列当发现有新消息到达时，就开始执行消息中的内容，由于消息中的内容是 t-sql 语句所以服务器端实际上是执行对数据库的操作。

客户端程序：

```
public Sub client ()
        Dim tM As New System.Messaging.MessageQueue()
        tM.Path = ".\Private$\jk"
'"FORMATNAME:PUBLIC=3d3dc813-c555-4fd3-8ce0-79d5b45e0d75"
'与指定计算机中的消息队列建立连接，
```

```
        Dim newMessage As New
System.Messaging.Message(TextBox1.Text)'接受文本筐的 t-sql 语句
        newMessage.Label = "This is the label"'消息名字,
        tM.Send(newMessage)'发送消息
    End Sub
```

服务端程序:

```
public Sub server ()
        Dim NewQueue As New
System.Messaging.MessageQueue("./Private$/jk")'"FORMATNAME:
PUBLIC=3d3dc813-c555-4fd3-8ce0-79d5b45e0d75"'与指定计算机中的消息队
列建立连接,
        Dim m As System.Messaging.Message
'查看消息队列中的消息
        m = NewQueue.Receive
        m.Formatter = New System.Messaging.XmlMessageFormatter(New
String()
{"System.String,mscorlib"})
        Dim st As String
        st = m.Body'消息队列中消息的消息内容。既 sql 语句
        Dim con As New OleDb.OleDbConnection("输入自己的数据库连接字
符串")
        con.Open()
        Dim com As New OleDb.OleDbCommand(st, con)'执行消息中的 sql
语句
        com.ExecuteNonQuery()
        con.Close()
End Sub
```

我为什么要使用消息队列来处理数据库的操作？这个问题我一直没回答，现在我就来回答这个问题。在本程序中你会发现在 sub client（）中我并没连接数据库和请求数据，而是通过发消息来操作数据库的，这个好处是节省了两部分时间：一是对数据库连解请求数据的时间，二是从数据库返回数据的时间。在很多情况下其实我们并不需要看见具体的数据就知道该怎么修改数据库中的数据。例如要删除张三的记录，就可以将一条简单的删除语句放入消息中，发给服务器让服务器程序去处理对数据的更改。

此外消息队列的另一个主要用途也是当前 erp 软件中必不可少的，就是在断开连接时保存信息，当连接恢复时发送消息。消息在如下两种情况中无法迅速地传递到它们的队列：当队列驻留的计算机无法工作时，或当路由消息所需的域控制器无法工作时。“消息队列”可让您应对这些情况，使得在从网络上断开连接或必要的计算机或控制器无法工作时，仍可以继续发送消息。在这些情形下，消息暂时存储在本地计算机或传递路由上的某个计算机的队列中，直到完成传递所需的资源重新联机。

例如，假设有一个记录所有在出差的销售人员发送的订单的中央队列。这些销售人员每天的大部分时间都以断开连接的方式工作，记录来自客户站点的订单信息，并且每天拨号连接一次，将所有这些信息传输

到中央队列中。因为消息在发送方断开连接时仍可发送到队列，所以销售人员可以在记录客户信息时立即发送他们的消息，但系统会缓存这些消息直到晚间进行拨号连接为止。

在断开连接时要怎么保存消息呢？向断开连接的队列发送消息同向可用队列发送消息的过程几乎完全相同。当要向其发送的队列不可用时，不必进行任何特殊的配置以使组件将消息存储在临时队列中。在 client 代码的 tM.Path = "./Private$/jk"后面有一条注释语句，其实这条语句就是实现向断开连接的队列发送消息的功能。只要将 tM.Path = "./Private$/jk"这条语句换成 tM.Path = "FORMATNAME:PUBLIC=3d3dc813-c555-4fd3-8ce0-79d5b45e0d75"其中 PUBLIC 后面的数字是要发送到计算机的 guid 数字。这个数字可以打开那台计算机的消息队列的属性看见。使用这种方法就可以在断开连接的情况下保证对服务器的操作是有效。现在运行这个程序后，打开 Windows2000 中的"开始"-》"程序"-》"管理工具"-》"计算机管理"。在"计算机管理"窗口中展开"服务和应用程序"-》"消息队列"-》"传出队列"，你将在右边的窗口中看见你建立的消息。（如果你使用 tM.Path = "./Private$/jk"语句，在"计算机管理"窗口中展开"服务和应用程序"-》"消息队列"-》"专用队列"可以看见你建立的队列。）

其实消息队列的编程并不复杂，但它在网络环境的程序开发中是非常有用的，可以简化大量的开发过程和节省开发时间。而且消息队列的编程有很大的灵活性，几乎可以解决网络编程的大部分问题。比如聊天程序，远程控制程序。

本文针对消息队列做了一个简单的介绍，并举了一个例来说明怎么在.net 下使用消息编程，达到快速高效稳定的对数据库进行操作。最后补充要说的是在 internet 中也一样可以使用消息队列，只需要将 tM.Path = "FORMATNAME:PUBLIC=3d3dc813-c555-4fd3-8ce0-79d5b45e0d75 语句后面的数字变成消息队列所在服务器的数字就可以了。但是要提醒大家的是使用消息在传输时将占有大量的带宽，所以在不是必须的时候，internet 下的编程不要使用消息。

在 vb.net、win2000 、sql server 2000 下通过。