基于消息与.Net Remoting 的分布式架构

张逸

www.agiledon.com

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

一、消息的定义

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

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

```
public interface IMessage:ICloneable
{
    IMessageItemSequence GetMessageBody();
    string GetMessageID();
    string GetMessageName();

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

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

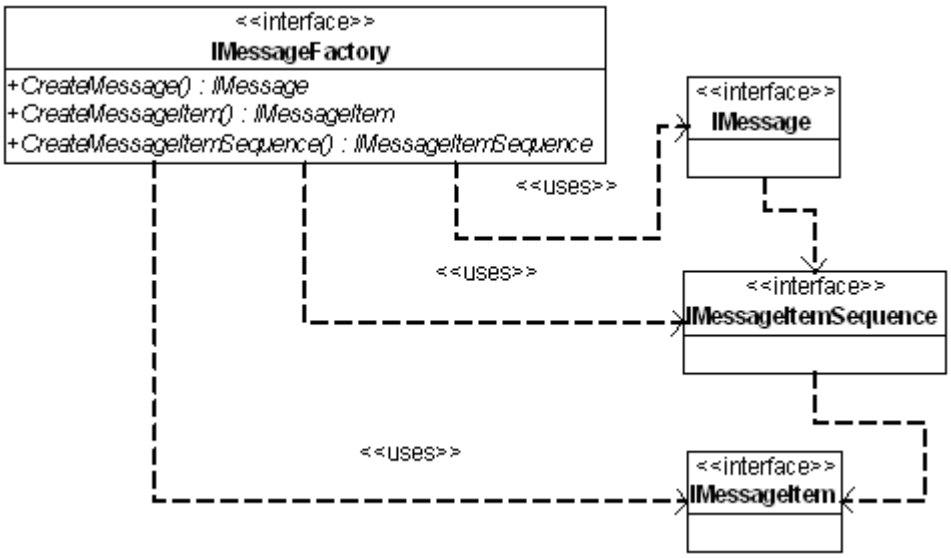
```
public interface IMessageItemSequence:ICloneable
{
    IMessageItem GetItem(string aName);
    void SetItem(string aName,IMessageItem aMessageItem);

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

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

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

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

各个消息对象之间的关系如下：



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

```
IMessageFactory factory = new MessageFactory();
IMessageItemSequence body = factory.CreateMessageItemSequence();
body.SetValue("name1","value1");
body.SetValue("name2","value2");

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

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

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

body.SetItem("item",item);
```

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

在接收消息端就可以通过获得 body 的消息体内容，进行相关业务的处理。

二、.Net Remoting 服务

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

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

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

然而，Request 服务处理消息的方式虽然一致，但毕竟服务实现的业务，即对消息处理的具体实现，却是不相同的。对我们要实现的服务，可以分为两大类： 业务服务与 Request 服务。实现的过程为：首先，具体的业务服务向 Request 服务发出 Request 请求，Request 服务侦听到该请求，然后 交由其侦听的服务来具体处理。

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

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

```
}
```

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

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

    private IListenService m_ListenService;

    #region IService Members

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

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

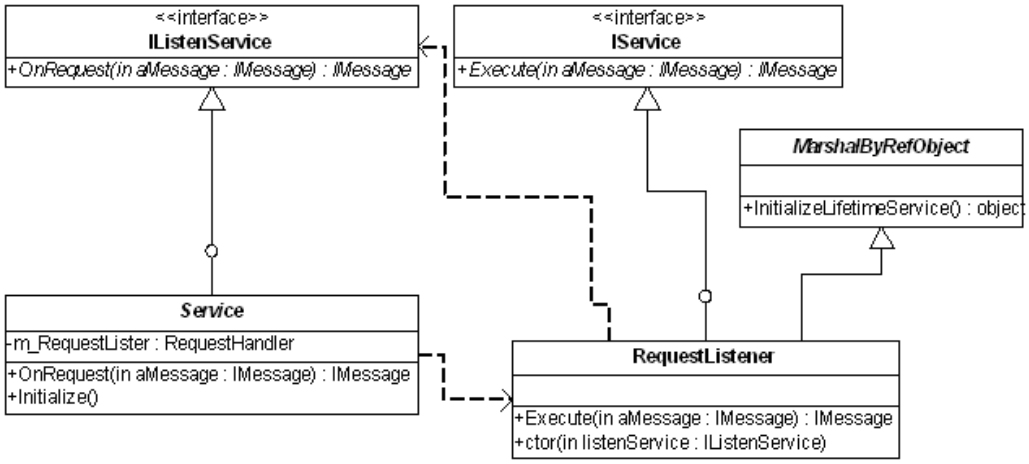
在 RequestListener 服务中，继承了 MarshalByRefObject 类，同时实现了 IService 接口。通过该类的构造函数，接收 IListService 对象。

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

```
public abstract class Service:IListenService
```

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

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

    #endregion

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

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



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

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

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

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

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

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

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

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

注意，在实例化的时候是将 this 作为 IListenService 对象传递给 RequestListener。因此，此时被 Marshal 的服务对象， 保留了业务服务本身即 Service 的指引。可以看出，

在 Service 和 RequestListener 之间，采用了“双重委派”的机制。

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

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

下面我们来看看 FindService()方法的实现：

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

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

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

```
    return service;
}
```

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

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

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

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

假设把进程分为服务端和客户端，那么对消息处理的步骤如下：

1、 在客户端调用 MyService 的 SendRequest()方法发送 Request 消息；

2、 查找被 Marshal 的服务，即 RequestListener 对象，此时该对象应包含对应的业务服务对象 MyService；

3、 在服务端调用 RequestListener 的 Execute()方法。该方法则调用业务服务 MyService 的 OnRequest()方法。

在这些步骤中，除了第一步在客户端执行外，其他的步骤均是在服务端进行。

三、业务服务对于消息的处理

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

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

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

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

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

```
protected abstract void AddRequestHandlers();
protected void AddRequestHandler(
    string aMessageName,
    RequestHandler handler)
{
    lock (this.m_EventHandlers)
    {
        if (!this.m_EventHandlers.Contains(aMessageName))
        {
            this.m_EventHandlers.Add(aMessageName,handler);
        }
    }
}

protected RequestHandler FindRequestHandler(string aMessageName)
{
    lock (this.m_EventHandlers)
    {
```

```
        RequestHandler handler =
(RequestHandler)m_EventHandlers[aMessageName];
        return handler;
    }
}
```

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

```
public class MyService:Service
{
    public MyService(string aServiceName):base(aServiceName)
    {}
    protected override void AddRequestHandlers()
    {
        this.AddRequestHandler("Test1",new RequestHandler(Test1));
        this.AddRequestHandler("Test2",new RequestHandler(Test2));
    }
    private void Test1(string aMessageName,
          IMessageItemSequence aMessageBody,
          ref  IMessageItemSequence aReplyMessageBody)
    {
        Console.WriteLine("MessageName:{0}\n",aMessageName);
        Console.WriteLine("MessageBody:{0}\n",aMessageBody);
        aReplyMessageBody.SetValue("result","Success");
    }
    private void Test2(string aMessageName,IMessageItemSequence
aMessageBody,ref  IMessageItemSequence aReplyMessageBody)
    {
        Console.WriteLine("Test2" + aMessageBody.ToString());
    }
}
```

Test1 和 Test2 方法均为匹配 RequestHandler 委托签名的方法，然后在

AddRequestHandlers()方法中，通过调用 AddRequestHandler()方法将这些方法与 MessageName 对应起来，添加到 m_EventHandlers 中。

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

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

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

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

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

    return message;
}
```

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

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

     protected override void AddRequestHandlers()
```

```
    {
        this.AddRequestHandler("Test1",new RequestHandler(Test1));
    }

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

}
```

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

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

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

    protected override void AddRequestHandlers()
    {}
}
```

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

```
IMessageFactory factory = new MessageFactory();
```

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

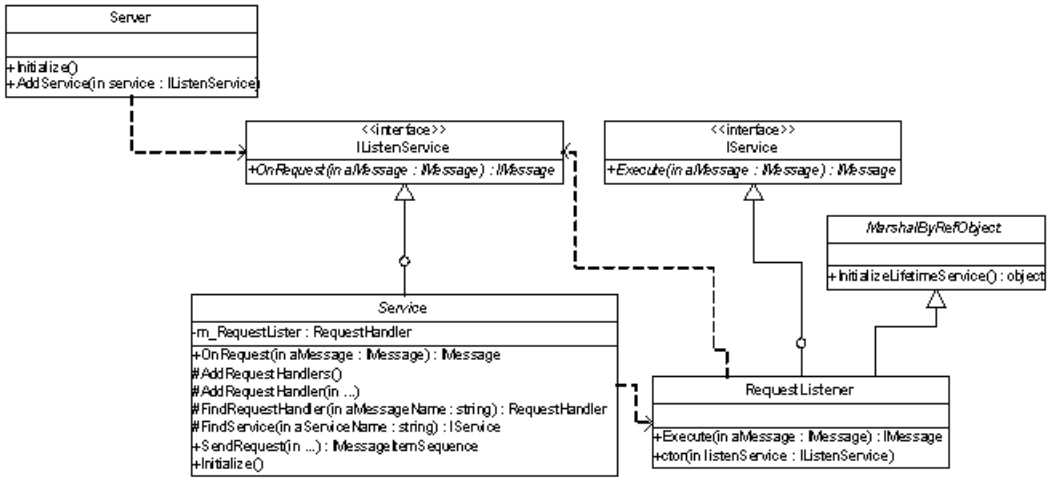
对于 service.SendRequest()的执行而言，会先调用 TestService 的 Test1 方法；然后再通过该方法向 MyService 发送，最终调用 MyService 的 Test1 方法。

我们还需要另外定义一个类，负责添加服务，并初始化这些服务：

```
public class Server
{
    public Server()
    {
        m_Services = new ArrayList();
    }
    private ArrayList m_Services;
    public void AddService(IListenService service)
    {
        this.m_Services.Add(service);
    }

    public void Initialize()
    {

        IDictionary tcpProp = new Hashtable();
        tcpProp["name"] = "tcp9090";
        tcpProp["port"] = 9090;

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

```
        {
            service.Initialize();
        }
    }
}
```

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



在服务端，可以调用 Server 类来初始化这些服务：

```
static void Main(string[] args)
{
    MyService service = new MyService("MyService");
    TestService service1 = new TestService("TestService");


    Server server = new Server();
    server.AddService(service);
    server.AddService(service1);

    server.Initialize();
    Console.ReadLine();
}
```

四、结论

利用这个基于消息与.Net Remoting 技术的分布式架构，可以将企业的业务逻辑转换为对

消息的定义和处理。要增加和修改业务，就体现在对消息的修改上。服务间的通信机制则完全 交给整个架构来处理。如果我们将每一个委托所实现的业务（或者消息）理解为 Contract，则该结构已经具备了 SOA 的雏形。当然，该架构仅仅处理了消 息的传递，而忽略了对底层事件的处理（类似于 Corba 的 Event Service），这个功能我想留待后面实现。

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