

第 51 章

Enterprise Services

本章内容:

- Enterprise Services 的功能
- 使用 Enterprise Services
- 创建服务组件
- 部署 COM+应用程序
- 使用事务和 COM+
- 为 Enterprise Services 创建 WCF 外观
- 从 WCF 客户端中使用 Enterprise Services

Enterprise Services 是 Microsoft 应用程序服务器技术的另一个名称,它为分布式解决方案提供服务。Enterprise Services 基于已使用多年的 COM+技术。然而,除了把.NET 对象包装为 COM 对象,以使用这些服务,.NET 还对.NET 组件进行了扩展,使.NET 组件可以直接利用这些服务。通过.NET,将很容易访问.NET 组件的 COM+服务。

Enterprise Services 还可以与 WCF 集成。使用一个工具可以为服务组件自动创建 WCF 服务前端,并从 COM+客户程序中调用 WCF 服务。



本章使用样本数据库 Northwind,它可以从 Microsoft 下载页面 www.microsoft.com/downloads 上下载。

51.1 使用 Enterprise Services

如果了解 Enterprise Services 的历史,就很容易理解 Enterprise Services 的复杂性和不同的配置选项(如果解决方案的所有组件都用.NET 开发,则不需要许多配置选项)。所以本节首先介绍 Enterprise Services 的历史。

之后,概述这种技术提供的不同服务,以便探讨应用程序可以使用的功能。

本节描述 Enterprise Services 的历史、其功能基于的环境,以及重要功能,例如,自动事务处理、对象池、基于角色的安全性、队列组件和松散耦合事件。

51.1.1 简史

Enterprise Services 可以追溯到作为 Windows NT 4.0 的一个选项包发布的 Microsoft 事务服务器 (Microsoft Transaction Server, MTS)。MTS 通过提供 COM 对象的事务处理等服务, 来扩展 COM。可以通过配置元数据来使用这些服务: 组件的配置定义了是否需要事务处理。有了 MTS, 就不再需要以编程方式处理事务。但是, MTS 有一个重要缺陷。因为 COM 不可扩展, 所以 MTS 在进行扩展时, 要重写 COM 组件注册配置, 把组件的实例化指向 MTS, 在 MTS 中实例化 COM 对象时还需要一些特殊的 MTS API 调用。这个问题在 Windows 2000 中得到了解决。

Windows 2000 的一个最重要的功能是在 COM+ 技术中集成了 MTS 和 COM。在 Windows 2000 中, 因为 COM+ 基本服务可以识别 COM+ 服务(以前的 MTS 服务)需要的环境, 所以不再需要特殊的 MTS API 调用。在 COM+ 服务中, 除了分布式事务之外, 还增加了一些新的服务功能。

Windows 2000 包含 COM+ 1.0。自从 Windows XP 和 Windows Server 2003 以来 COM+ 1.5 可用。COM+ 1.5 新增了更多功能, 以提高可伸缩性和可用性, 包括应用程序池和循环, 以及可配置的隔离级别。

.NET Enterprise Services 允许在 .NET 组件中使用 COM+ 服务, 并为 Windows 2000 及其后续版本提供支持。当在 COM+ 应用程序中运行 .NET 组件时, 不需要使用 CCW(参阅第 26 章); 而应用程序作为 .NET 组件运行。在操作系统上安装 .NET 运行库时, 会给 COM+ 服务添加一些运行库扩展。如果安装了两个带有 Enterprise Services 的 .NET 组件, 且 A 组件使用 B 组件, 则不使用 COM 编组功能, .NET 组件可以直接彼此调用。

51.1.2 使用 Enterprise Services 的场合

业务应用程序在逻辑上可以分为表示层、业务层和数据服务层。表示服务层(presentation service layer)负责用户交互, 在该服务层上, 用户可以与应用程序交互, 来输入和查看数据。这一层使用的技术是 Windows 窗体、WPF 和 ASP.NET。业务服务层由业务规则和数据规则组成。数据服务层与持久存储器交互。在该层上可以通过组件来使用 ADO.NET。Enterprise Services 可以在业务服务层和数据服务层上使用。

图 51-1 显示了两个典型应用程序的情况。Enterprise Services 可以直接从使用 Windows 窗体或 WPF 的胖客户端中使用, 或从运行 ASP.NET 的 Web 应用程序中使用。

Enterprise Services 也是一种可伸缩的技术。使用组件负载均衡技术, 就可以在不同的系统上分布客户端的负载。

还可以在客户端系统上使用 Enterprise Services, 因为该技术包含在 Windows 7 和 Windows Vista 中。

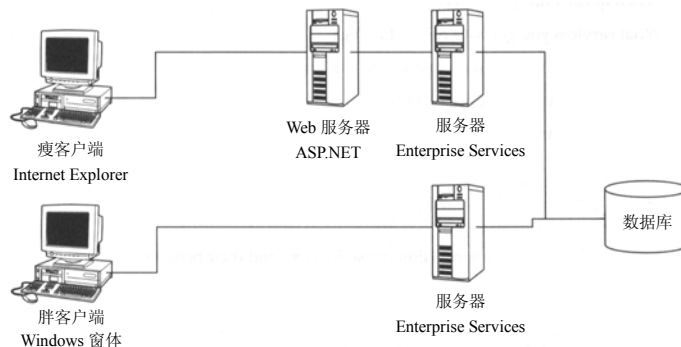


图 51-1

51.1.3 重要功能

下面看看使用 Enterprise Services 的优点，介绍它的重要功能，例如，自动事务处理、基于角色的安全性、队列组件和松散耦合事件。

1. 环境

Enterprise Services 所提供的服务的基本功能是上下文(context)，该上下文是 Enterprise Services 所有功能的基础。该上下文可以截获方法调用，在调用希望的方法调用之前执行某个服务功能。例如，在调用组件实现的方法之前，可以创建事务作用域或同步范围。图 51-2 显示了运行在两个不同上下文 X 和 Y 中的对象 A 和对象 B。通过代理在上下文之间截获调用。该代理可以使用 Enterprise Services 所提供的服务，该服务用后面的功能解释。

这里通过上下文，COM 组件和.NET 组件就可以参与同一个事务处理。这要归功于 ServicedComponent 基类，这个类本身派生自 MarshalByRefObject，用于集成.NET 和 COM+上下文。

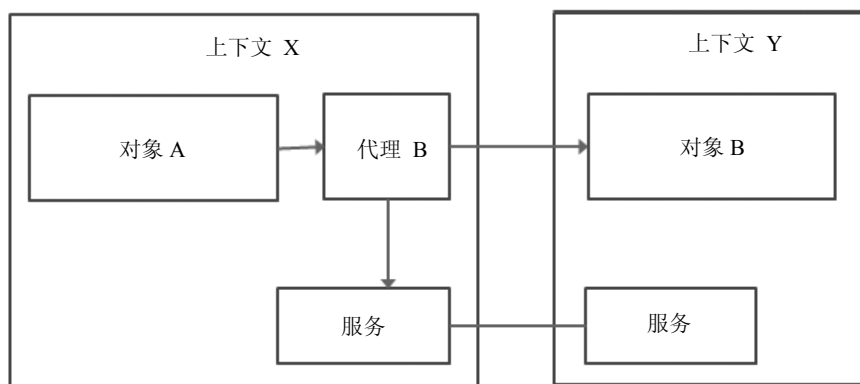


图 51-2

2. 自动事务处理

Enterprise Services 最常用的功能是自动事务处理。使用自动事务处理，就不需要在代码中启动和提交事务，而是可以把属性应用于一个类。使用[Transaction]属性和 Required、Supported、RequiresNew、NotSupported 选项，就可以用类对事务的要求标记类。如果用 Required 选项标记属性，在方法启动时就会自动创建一个事务，并在完成事务的根组件完成时提交或终止事务。

在开发复杂的对象模型时，程序的这种声明方式具有特别的优势。这里，自动事务处理与手动为事务编程相对有优势。例如，假定有一个 Person 对象，以及几个与 Person 对象关联的 Address 和 Document 对象。现在要把 Person 对象和所有关联的对象都存储在一个事务中。通过编程方式进行事务处理，就意味着把一个事务对象传递给所有相关的对象，以便它们能参与同一个事务处理。以声明的方式使用事务，就不需要传递事务对象，因为这会通过上下文在后台上进行。

3. 分布式事务处理

Enterprise Services 不仅提供了自动事务处理，事务处理还可以分布在多个数据库上。Enterprise Services 事务处理通过分布式事务处理协调器(Distributed Transaction Coordinator, DTC)来登记。DTC

支持使用 XA 协议的数据库, XA 协议是一种分两个阶段提交的协议, 由 SQL Server 和 Oracle 支持。单个事务处理可以把写入的数据分布到 SQL Server 和 Oracle 数据库中。

分布式事务处理不仅对数据库有用, 而且单个事务处理还可以把写入的数据分布到数据库和消息队列上, 如果这两个操作中的一个失败, 另一个操作就会回滚。消息队列详见第 46 章。



Enterprise Services 支持可升级的事务处理。如果使用 SQL Server, 且在一个事务处理中只要一个激活的连接, 就创建一个本地事务处理。如果在同一个事务处理中激活了另一个事务处理资源, 该事务处理就升级为 DTC 事务处理。

本章后面将讨论如何创建需要事务处理的组件。

4. 对象池

对象池是 Enterprise Services 提供的另一个功能。这些服务使用线程池来回应客户端的请求。对象池可以用于初始化时间比较长的对象。使用对象池, 就会提前创建对象, 这样客户端就不需要一直等待直到初始化对象。

5. 基于角色的安全性

使用基于角色的安全性, 可以以声明方式定义角色, 定义从什么角色中可以使用哪些方法或组件。系统管理员给用户或用户组赋予这些角色。在程序中, 不需要处理访问控制列表, 而可以使用只是简单字符串的角色。

6. 队列组件

队列组件是消息队列的一个抽象层。客户端不是把消息发送给消息队列, 而是通过一个记录器调用方法, 该记录器提供的方法与在 Enterprise Services 中配置的.NET 类相同。该记录器再创建消息, 通过消息队列把它们传输给服务器应用程序。

如果客户端应用程序运行在断开连接的环境中(例如, 不总是连接到服务器的笔记本电脑), 或者发送给服务器的请求要在转发给另一个服务器(例如发送给商业伙伴的服务器)之前缓存, 队列组件和消息队列就很有用。

7. 松散耦合事件

第 8 章讨论了.NET 的事件模型, 第 26 章讨论了如何在 COM 环境中使用事件。通过这两种事件机制, 客户端和服务端之间就会建立一个牢固的连接。这与松散耦合事件(LCE)不同。在 LCE 中, COM+功能会插入客户端和服务端之间(如图 51-3 所示)。发布者会通过定义一个事件类, 用 COM+来注册它提供的事件。发布者不是把事件直接发送给客户端, 而是把事件发送给用 LCE 服务注册的事件类。LCE 服务会把事件转发给订阅者, 订阅者是为事件注册请求订阅的客户端应用程序。

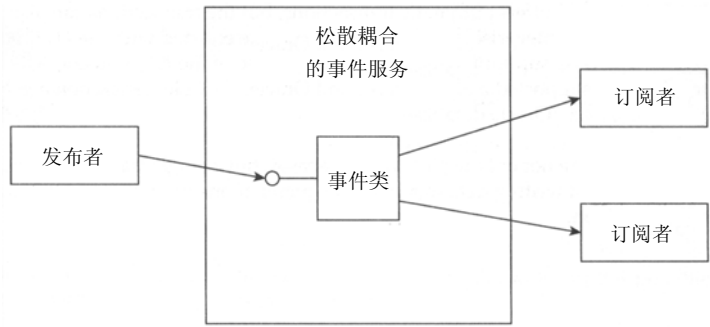


图 51-3

51.2 创建简单的 COM+应用程序

创建可以用 Enterprise Services 配置的 .NET 类时，必须引用 System.EnterpriseServices 程序集，把 System.EnterpriseServices 名称空间添加到 using 声明中。这个应用程序所使用的最重要的类是 ServicedComponent。

第一个示例仅说明创建服务组件的基本要求。首先创建一个 C# 库应用程序。所有 COM+ 应用程序都必须作为库应用程序编写，无论它们将运行在自己的进程中，还是运行在客户端的进程中，都是如此。把该库命名为 SimpleServer。引用 System.EnterpriseServices 程序集，给 Assemblyinfo.cs 文件和 Class1.cs 文件添加 “using System.EnterpriseServices;” 声明。

对于该项目还需要对库应用一个强名称。对于一些 Enterprise Services 功能，还需要在全局程序集缓存中安装程序集。强名称和全局程序集缓存在第 18 章讨论过。

51.2.1 ServicedComponent 类

每个服务组件类都必须派生于 ServicedComponent 基类。因为 ServicedComponent 类本身派生自 ContextBoundObject 类，所以实例会绑定到 .NET Remoting 环境上。

ServicedComponent 类包含一些可重写且受保护的方法，如表 51-1 所示。

表 51-1

受保护的方法	说 明
Activate()、Deactivate()	如果把对象配置为使用对象池，就调用 Activate() 和 Deactivate() 方法。从对象池中取出对象时，调用 Activate() 方法。在对象返回对象池之前，调用 Deactivate() 方法
CanBePooled()	这是对象池的另一个方法。如果对象的状态不一致，就可以在 CanBePooled() 方法重写的实现代码中返回 false。虽然这样对象就不会放回对象池，但是它被销毁。而对象池会创建一个新对象
Construct()	这个方法在实例化时调用，实例化时将一个构造字符串传递给对象。构造字符串可以由系统管理员修改。本章后面将使用构造字符串定义数据库连接字符串

51.2.2 程序集的属性

还需要一些 Enterprise Services 属性。ApplicationName 属性定义应用程序在 Component Services Explorer 中显示的名称。Description 属性的值在应用程序配置工具中显示为对应描述。

ApplicationActivate 属性允许使用 ActivationOption.Library 或 ActivationOption.Server 选项, 定义应用程序是应配置为库应用程序还是服务器应用程序。如果配置为库应用程序, 该应用程序就会在客户端的进程中加载。在这种情况下, 客户端可能是 ASP.NET 运行库。如果配置为服务器应用程序, 就启动应用程序的进程。进程的名称是 dllhost.exe。使用 ApplicationAccessControl 属性可以关闭安全功能, 这样每个用户就都可以使用组件。

把 class1.cs 文件重命名为 SimpleComponent.cs, 在名称空间声明的外部添加这些属性:


```
[assembly: ApplicationName("Wrox EnterpriseDemo")]
[assembly: Description("Wrox Sample Application for Professional C#")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(false)]
```

表 51-2 列出了可以用 Enterprise Services 应用程序定义的、最重要的程序集属性。

表 51-2	
属 性	说 明
ApplicationName	ApplicationName 属性定义 COM+应用程序的名称, 在配置组件后, 该名称显示在 Component Services Explorer 中
ApplicationActivation	ApplicationActivation 属性确定应用程序是应作为库运行在客户端应用程序库中, 还是应启动一个单独的进程。要配置的选项用 ActivationOption 枚举定义。ActivationOption.Library 指定在客户端的进程中运行应用程序; ActivationOption.Server 启动它自己的进程 dllhost.exe
ApplicationAccessControl	ApplicationAccessControl 特性定义应用程序的安全配置。使用布尔值可以设置启用或禁用访问控制。使用 Authentication 属性可以设置私有级别: 即客户端是应在每个方法调用中验证, 还是仅在连接时验证, 还可以确定发送的数据是否应加密

51.2.3 创建组件

在 SimpleComponent.cs 文件中, 可以创建服务组件类。对于服务组件, 最好定义一个接口作为客户端和组件之间的协定。虽然这不是一个苛刻的要求, 但一些 Enterprise Services 功能(例如, 在方法或接口级别上设置基于角色的安全性)需要接口。用 Welcome()方法创建 IGreeting 接口。可以从 Enterprise Services 功能中访问的服务组件类和接口都需要应用 CpmVisible 特性:



可从
wrox.com
下载源代码

```
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IGreeting
    {
        string Welcome(string name);
    }
}
```

代码段 SimpleServer/IGreeting.cs

SimpleComponent 类派生自 SercivedComponent 基类, 并实现 IGreeting 接口。Sercived Component 类作为所有服务组件类的基类, 为激活阶段和构造阶段提供一些方法。把 EventTrackingEnabled 特

性应用于这个类，使之能用 Component Services Explorer 监控对象。在默认情况下禁用监控功能，因为使用这个功能会降低性能。Description 特性仅指定显示在 Explorer 上的文本：



可从
wrox.com
下载源代码

```
using System.EnterpriseServices;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [EventTrackingEnabled(true)]
    [ComVisible(true)]
    [Description("Simple Serviced Component Sample")]
    public class SimpleComponent: ServicedComponent, IGreeting
    {
        public SimpleComponent()
        {
        }
    }
}
```

代码段 SimpleServer/SimpleComponent.cs

Welcome()方法仅返回“Hello, ”和传递给参数的名称。这样在 Component Services Explorer 中运行组件的同时可以查看一些可见的结果，Thread.Sleep()方法模拟一些处理时间：

```
public string Welcome(string name)
{
    // simulate some processing time
    System.Threading.Thread.Sleep(1000);
    return "Hello, " + name;
}
}
```

除了应用一些特性和从 ServicedComponent 派生类之外，不需要对使用 Enterprise Services 功能的类做什么特别的工作。剩下的就是构建和部署客户应用程序。

在第一个示例组件中设置 EventTrackingEnabled 特性。有一些更常用的特性会影响服务组件的配置，如表 51-3 所示。

表 51-3

特 性 类	说 明
EventTrackingEnabled	设置 EventTrackingEnabled 特性，将允许使用 Component Services Explorer 监控组件。因为把这个特性设置为 true 会产生额外的开销，所以默认情况下关闭事件跟踪功能
JustInTimeActivation	使用这个特性，可以把组件配置为在调用者实例化类时不激活，而在调用第一个方法时激活。另外，使用这个特性，组件可以自动失效
ObjectPooling	如果与方法调用的时间相比，组件的初始化时间比较长，就可以用 ObjectPooling 特性配置对象池。使用这个特性，可以定义影响池中对象数的最大值和最小值
Transaction	Transaction 特性定义组件的事务特征。这里组件定义了是否需要、支持或不支持事务

51.3 部署

拥有服务组件的程序集必须用 COM+配置。这个配置可以自动进行,或通过手工注册程序集来完成。

51.3.1 自动部署

如果启动使用服务组件的.NET 客户端应用程序,就会自动配置 COM+应用程序。所有派生自 `ServicedComponent` 类的类都是这样。诸如 `EventTrackingEnabled` 的应用程序特性和类特性定义该配置的特征。

自动部署有一个重要的缺点。在自动部署时,客户端应用程序需要管理权限。如果调用服务组件的客户端应用程序是 ASP.NET 应用程序,那么 ASP.NET 运行库一般没有管理权限。因为这个缺点,所以自动部署仅在开发阶段有用。而在开发阶段,自动部署是一个极大的优势。在每次构建程序后,都不需要进行手动部署。

51.3.2 手动部署

手动部署程序集可以通过.NET 服务安装工具 `regcvcs.exe`(一种命令行实用程序)进行。输入下面的命令:

```
regcvcs SimpleServer.dll
```

就会把 `SimpleServer` 程序集注册为一个 COM+应用程序,并根据其属性配置包含的组件,创建一个可以由访问.NET 组件的 COM 客户端使用的类型库。

在配置好程序集后,就可以选择 **Administrative Tools | Component Services** 命令,启动 **Component Services Explorer**。在应用程序的左边树型视图中选择 **Component Services | Computers | My Computer | COM+ Application**,验证应用程序是否已配置。



在 Windows Vista 上,必须启动 MMC,添加 Component Services 插件,才能看到组件服务管理器。

51.3.3 创建安装软件包

使用组件服务管理器,可以为服务器系统或客户端系统创建安装软件包。服务器的安装软件包包含用于把应用程序安装到另一个服务器上的程序集和配置设置。如果在运行在另一个系统上的应用程序中调用服务组件,就必须在客户端系统上安装一个代理。客户端的安装软件包包含代理的程序集和配置。

要创建安装软件包,可以启动组件服务管理

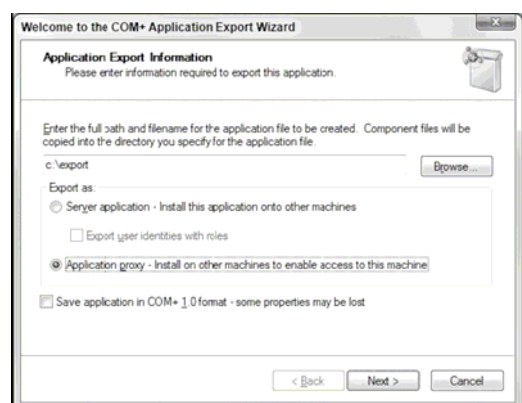


图 51-4

器，选择 COM+ 应用程序，选择菜单项 Action | Export，单击第一个对话框中的 Next 按钮，打开如图 51-4 所示的对话框。在这个对话框中可以导出 Server application 或 Application proxy。在 Server application 选项中，还可以进行配置，导出带有角色的用户标识。这个选项只能在目标系统与创建安装软件包的系统在同一域中时使用，因为配置的用户标识放在安装软件包中。使用 Application proxy 选项，会创建客户端系统的安装软件包。



如果应用程序配置为库应用程序，创建 Application proxy 的选项就不可用。

要安装代理，只需启动安装软件包中的 setup.exe。注意应用程序代理不能安装在应用程序所安装在的同一个系统中。在安装完应用程序代理后，在组建服务管理器中就有一项表示应用程序代理。在应用程序代理中，能配置的唯一选项是 Activation 选项卡中的服务器名，如下一节所述。

51.4 组件服务管理器

在成功配置后，就可以在组件服务管理器的树型视图中查看 Wrox EnterPriseDemo 应用程序名。这个名称由 [ApplicationName] 特性设置。选择 Action | Properties 命令，打开如图 51-5 所示的对话框。名称和描述都已使用属性配置了。在选择 Activations 选项卡时，可以看到该应用程序被配置为服务器应用程序，因为它使用 [ApplicationActivation] 特性定义，选择 Security 选项卡，其中没有选择 Enforce access checks for this application 选项，因为把 [ApplicationAccessControl] 特性设置为 fasle。

通过这个应用程序还可以设置其他一些选项：

- **Security**——在安全配置中，可以启用或禁用访问检查。如果启用安全功能，就可以把访问检查设置为应用程序级别、组件、接口和方法级别。还可以把数据包私密性作为调用的身份验证级别，加密在网络上发送的消息。当然，这会增加系统开销。
- **Identity**——在服务器应用程序中，使用 Identity 选项卡可以为驻留应用程序的进程配置要使用的用户账户。在默认情况下，这是一个交互式的用户。这个设置在调试应用程序时非常有用，但如果应用程序正在服务器上运行，该设置就不能在产品系统上使用，因为没有人能登录。在把应用程序安装到产品系统上之前，应为应用程序使用特定的用户，来测试应用程序。
- **Activation**——Activation 选项卡允许把应用程序配置为库或服务器应用程序。COM+ 1.5 的两个新选项是把应用程序作为 Windows 服务运行和使用 SOAP 访问应用程序。第 25 章讨论过 Windows 服务。选择 SOAP 选项，将使用在 Internet Information Server 中配置的.NET

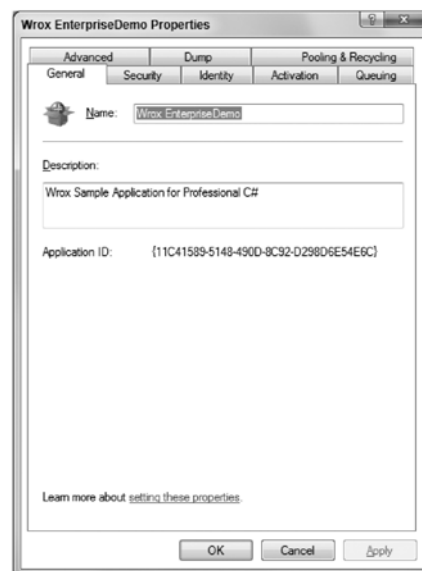


图 51-5

Remoting 来访问组件。本章的后面不使用 .NET Remoting，而使用 WCF 访问组件。WCF 已在第 43 章讨论过。

对于应用程序代理，Remote server name 选项是可以配置的唯一选项。使用这个选项可以设置服务器名。在默认情况下，DCOM 协议用作网络协议。但是，如果在服务器配置选择 SOAP，就通过 .NET Remoting 进行通信。

- **Queuing**——使用消息队列的服务组件需要 Queuing 配置。
- **Advanced**——在 Advanced 选项卡中，可以指定应用程序是否应在客户端处于未激活状态一段时间后停止。还可以指定是否锁定某个配置，这样就不会有人在无意中修改它。
- **Dump**——如果应用程序崩溃，就可以指定应把转储文件存储在目录的什么地方。Dump 对于利用 C++ 开发的组件很有用。
- **Pooling & Recycling**——这是 COM+ 1.5 的一个新选项。利用这个选项可以配置应用程序是否根据生命周期、需要的内存、调用的次数等重新启动(循环)。

使用 Component Services 浏览器还可以查看和配置组件本身。在打开应用程序的子元素时，可以查看 Wrox.ProCSharp.EnterpriseServices.SimpleComponent 组件。选择 Action | Properties 命令会打开如图 51-6 所示的对话框。

使用这个对话框，可以配置如下选项：

- **Transactions**——使用 Transactions 选项卡可以指定组件是否需要事务。下一个示例将使用这个功能。
- **Security**——如果应用程序启用安全功能，利用这个配置就可以定义允许使用组件的角色。
- **Activation**——Activation 配置允许设置对象池，指定构造字符串。
- **Concurrency**——如果组件不是线程安全的，concurrency 就可以设置为 Required 或 Requires New。这样 COM+ 运行库一次就仅允许一个线程访问组件。

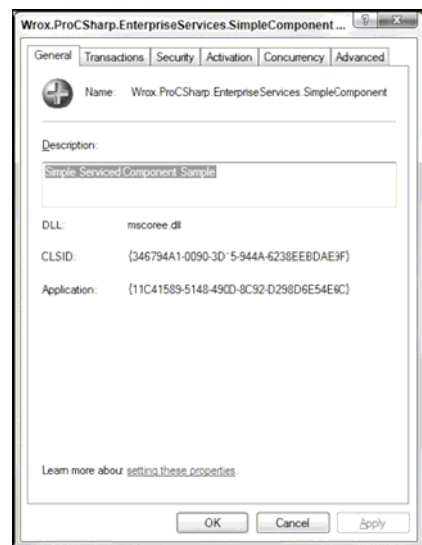


图 51-6

51.5 客户端应用程序

在构建服务组件库后，就可以创建客户端应用程序，这可以是简单的 C# 控制台应用程序。在为客户端创建项目后，必须引用服务组件中的 SimpleServer 程序集和 System.EnterpriseServices 程序集。接着编写代码，实例化一个新的 SimpleComponent 实例，并调用 Welcome() 方法。在下面的代码中，调用 Welcome() 方法 10 次。在垃圾收集器采取措施前，using 语句帮助释放分配给实例的资源。通过 using 语句，就会在 using 语句块的结尾处调用服务组件的 Dispose() 方法。



可从
wrox.com
下载源代码

```
using System;

namespace Wrox.ProCSharp.EnterpriseServices
{
```

```

class Program
{
    static void Main()
    {
        using (SimpleComponent obj = new SimpleComponent())
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine(obj.Welcome("Stephanie"));
            }
        }
    }
}

```

代码段 ClientApplication/Program.cs

如果在配置服务器之前启动客户端应用程序，服务器就会自动配置。服务器的自动配置通过属性指定的值来进行。下面进行测试。注销服务组件，并再次启动客户程序。如果在客户端应用程序的启动过程中配置服务组件，启动时间就会较长。这个功能只能用在开发阶段。自动部署还需要管理权限。如果从 Visual Studio 中启动应用程序，就应在有管理权限的情况下启动 Visual Studio。

在运行应用程序时，可以用 Component Services Explorer 监控服务组件。在树型视图中选择 Components，再选择 View | Detail 命令，就可以查看设置[EventTrackingEnabled]特性后实例化对象的个数。

可以看出，创建服务组件就是从 ServicedComponent 基类中派生一个类，再设置一些特性，来配置应用程序。下面要学习如何同时使用事务和服务组件。

51.6 事务

自动事务处理是 Enterprise Services 中最常用的功能。使用 Enterprise Services 可以把组件标记为需要事务，接着从 COM+运行库中创建事务。组件中所有能识别事务的对象(如 ADO.NET 连接)都在事务中运行。



事务的概念详见第 23 章。

51.6.1 事务的特性

服务组件可以用[Transaction]特性标记，以定义组件是否需要事务，以及如何进行事务处理。

图 51-7 显示了不同的事务配置的多个组件。客户调用组件 A。因为组件 A 用 TransactionOption.Required 配置，而且以前不存在事务，所以创建一个新的事务 1。组件 A 调用组件 B，组件 B 调用组件 C。因为组件 B 用 TransactionOption.Supported 配置，而组件 C 的配置是 TransactionOption.Required，所以 3 个组件(A、B 和 C)都使用同一个事务环境。如果组件 B 用 TransactionOption.NotSupported 配置，组件 C 就会得到一个新事务。因为组件 D 用 TransactionOption.RequiresNew 配置，所以在组件 A 调用组件 D 时，会创建一个新事务。

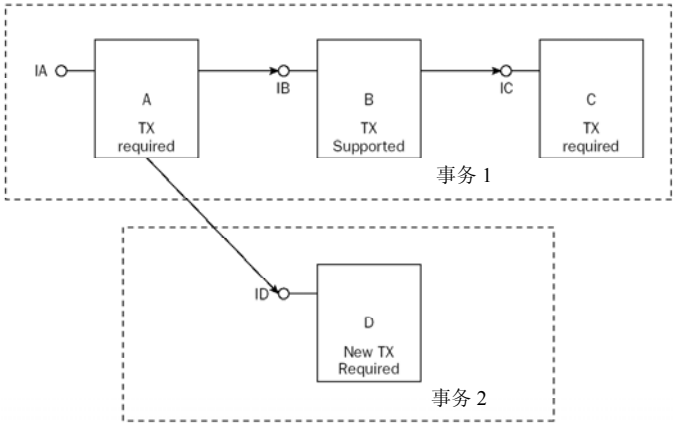


图 51-7

表 51-4 给出了用 TransactionOption 可以设置的不同值。

表 51-4

TransactionOption 枚举的值	说 明
Required	把[Transaction]特性设置为 TransactionOption.Required，表示组件在事务中运行。如果已经创建了一个事务，组件就运行在同一个事务中。如果不存在任何事务，就创建一个事务
RequiresNew	TransactionOption.RequiresNew 总是会创建一个新的事务。组件从来不参与调用者所在的事务
Supported	使用 TransactionOption.Supported，组件就不需要事务。但是，如果这些组件需要事务，事务就会跨越调用者和被调用的组件
NotSupported	TransactionOption.NotSupported 选项表示无论调用者是否有事务，组件从来都不运行在事务中
Disabled	TransactionOption.Disabled 表示忽略当前上下文的事务

51.6.2 事务的结果

设置环境的 consistent 位和 done 位会影响事务。如果把 consistent 位设置为 true，就表示组件对事务的结果很满意。如果所有参与事务处理的组件都成功了，就提交该事务。如果把 consistent 位设置为 false，则组件对事务的结果不满意，在启动事务的根对象完成时，就会终止事务。如果设置了 done 位，就可以在方法调用结束后停用对象，用下一个方法调用创建新实例。

使用 ContextUtil 类的 4 个方法可以设置 consistent 位和 done 位，结果如表 51-5 所示。

表 51-5

ContextUtil 方法	consistent 位	done 位
SetComplete()	true	true
SetAbort()	false	true

EnableCommit()	true	false
DisableCommit()	false	false

在.NET 中,还可以对方法应用[AutoComplete]特性,来设置 consistent 位和 done 位,而不是调用 ContextUtil()方法。使用这个特性,如果方法成功,就自动调用 ContextUtil.SetComplete()方法。如果方法失败,并抛出一个异常,则通过[AutoComplete]特性调用 ContextUtil.SetAbort()方法。

51.7 示例应用程序

在这个示例应用程序中,要模拟一个简化的场景:把一些新订单写入 Northwind 样本数据库中。如图 51-87 所示,多个组件和 COM+应用程序一起使用。从客户端应用程序中调用 OrderControl 类,创建新的订单。OrderControl 类使用 OrderData 组件,该组件负责在 Northwind 数据库的 Order 表中创建一个新条目。OrderData 组件使用 OrderLineData 组件把 Order Detail 条目写入数据库中。OrderData 和 OrderLineData 组件都必须参与同一个事务处理。

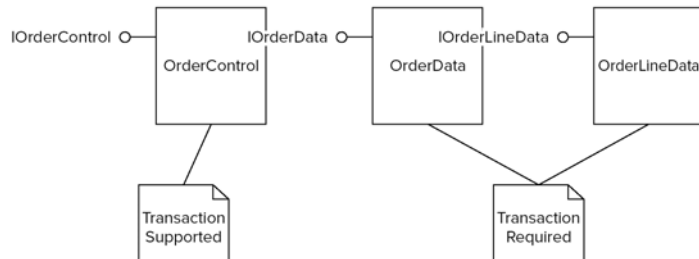


图 51-8

首先要创建一个 C#组件库,命名为 NorthwindCompnnent。用密钥文件给程序集签名,定义 Enterprise Services 应用程序特性,如下面的代码所示:



可从
wrox.com
下载源代码

```
[assembly: ApplicationName("Wrox.NorthwindDemo")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(false)]
```

代码段 NorthwindComponents/AssemblyInfo.cs

51.7.1 实体类

接着添加 Order 和 OrderLine 实体类,它们表示 Northwind 数据库中 Order 表和 Order Detail 表中的列。实体类是数据存储器,表示对应用程序域很重要的数据,在本例中就是下订单。Order 类有一个静态方法 Create(),它创建并返回 Order 类的一个新实例,用传递给该方法的参数初始化这个实例。类 Order 类还有一些只读属性,如 OrderId、CustomerId、OrderData、ShipAddress、ShipCity 和 ShipCountry。OrderId 属性的值在 Order 类的创建阶段未知,但因为 Northwind 数据库中的 Order 表有一个自动递增的属性,所以 OrderId 属性的值仅在订单写入数据库后才已知。SetOrderId()方法用于在订单写入数据库后设置相应的 id。因为这个方法由同一程序集中的类调用,所以把这个方法的访问级别设置为 internal。AddOrderLine()方法把订单细节添加到订单中。



可从
wrox.com
下载源代码

```
using System;
```

```

using System.Collections.Generic;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [Serializable]
    public class Order
    {
        public static Order Create(string customerId, DateTime orderDate,
                                   string shipAddress, string shipCity, string shipCountry)
        {
            return new Order
            {
                CustomerId = customerId,
                OrderDate = orderDate,
                ShipAddress = shipAddress,
                ShipCity = shipCity,
                ShipCountry = shipCountry
            };
        }

        public Order()
        {
        }

        internal void SetOrderId(int orderId)
        {
            this.OrderId = orderId;
        }

        public void AddOrderLine(OrderLine orderLine)
        {
            orderLines.Add(orderLine);
        }

        private readonly List<OrderLine>orderLines = new List<OrderLine >();

        public int OrderId { get; private set; }
        public string CustomerId { get; private set; }
        public DateTime OrderDate { get; private set; }
        public string ShipAddress { get; private set; }
        public string ShipCity { get; private set; }
        public string ShipCountry { get; private set; }

        public OrderLine[] OrderLines
        {
            get
            {
                OrderLine[] ol = new OrderLine[orderLines.Count];
                orderLines.CopyTo(ol);
                return ol;
            }
        }
    }
}

```

代码段 NorthwindComponents/Order.cs

第二个实体类是 `OrderLine`，它有一个类似于 `Order` 类的方法的静态 `Create()` 方法。除此之外，这个类仅包含一些属性，如 `ProductId`、`UnitPrice` 和 `Quantity`。



可从
wrox.com
下载源代码

```
using System;
```

```

namespace Wrox.ProCSharp.EnterpriseServices
{
    [Serializable]
    public class OrderLine
    {
        public static OrderLine Create(int productId, float unitPrice, int quantity)
        {
            return new OrderLine
            {
                ProductId = productId,
                UnitPrice = unitPrice,
                Quantity = quantity
            };
        }
        public OrderLine()
        {
        }

        public int ProductId { get; set; }
        public float UnitPrice { get; set; }
        public int Quantity { get; set; }
    }
}

```

代码段 NorthwindComponents/OrderLine.cs

51.7.2 OrderControl 组件

OrderControl 类表示一个简单的业务服务组件。在这个示例中，在 **IOrderControl** 接口中只定义了一个方法，即 **NewOrder()**。**NewOrder()**方法的实现代码只实例化 **OrderData** 数据服务组件的一个新实例，并调用 **Insert()**方法把 **Order** 对象写入数据库中。在比较复杂的情况下，这个方法可以扩展为把一个日志项写入数据库中，或者调用队列组件，把 **Order** 对象发送给消息队列。



可从
wrox.com
下载源代码

```

using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderControl
    {
        void NewOrder(Order order);
    }

    [Transaction(TransactionOption.Supported)]
    [EventTrackingEnabled(true)]
    [ComVisible(true)]
    public class OrderControl: ServicedComponent, IOrderControl
    {
        [AutoComplete()]
        public void NewOrder(Order order)
        {
            using (OrderData data = new OrderData())
            {

```

```

        data.Insert(order);
    }
}
}
}

```

代码段 NorthwindComponents/OrderControl.cs

51.7.3 OrderData 组件

OrderData 类负责把 Order 对象的值写入数据库中。IOrderUpdate 接口定义 Insert() 方法。可以扩展这个接口，使之也支持 Update() 方法，其中该方法会更新数据库中已有的项。



可从
wrox.com
下载源代码

```

using System;
using System.Data.SqlClient;
using System.EnterpriseServices;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderUpdate
    {
        void Insert(Order order);
    }
}

```

代码段 NorthwindComponents/OrderData.cs

OrderData 类的[Transaction]特性值是 TransactionOption.Required。这表示组件在任何情况下都运行在事务中。要么事务由调用者创建，并且 OrderData 类使用同一个事务；要么创建一个新事务。这里会创建一个新事务，因为主调组件 OrderControl 没有事务。

在服务组件中，只能使用默认的构造函数。但是，可以使用组件服务管理器配置一个发送给组件的构造字符串(如图 51-9 所示)。选择组件配置的 Activation 选项卡，可以更改构造字符串。在设置[ConstructiunEnable]特性时，选中 Enable object constructiun 选项，因为它与 OrderData 类一起使用。[ConstructiunEnable]特性的 Default 属性定义默认的连接字符串，在注册程序集后，该字符串会显示在 Activation 设置中。设置这个特性还需要重载 ServicedComponent 基类中的 Construct() 方法。在对象实例化时由 COM+ 运行库调用这个方法，同时把构造字符串作为参数传递。把构造字符串设置为 connectionString 变量，该变量在以后用于连接数据库。

```

[Transaction(TransactionOption.Required)]
[EventTrackingEnabled(true)]
[ConstructionEnabled
    (true, Default="server=(local);database=northwind;trusted_connection=true")]
[ComVisible(true)]
public class OrderData: ServicedComponent, IOrderUpdate
{
    private string connectionString;

    protected override void Construct(string s)
    {
        connectionString = s;
    }
}

```


Insert()方法是组件的核心。这里使用 ADO.NET 把 Order 对象写入数据库中。ADO.NET 详见第 30 章。对于这个示例，我们创建一个 SqlConnection 对象，在该对象中，使用 Construct()方法设置的连接字符串，用于初始化该对象。

把[AutoComplete()]属性应用于下面的方法，使用前面论述的自动事务处理。

```
[AutoComplete()]
public void Insert(Order order)
{
    var connection = new SqlConnection(
        connectionString);
```

connection.CreateCommand()方法创建一个 SqlCommand 对象，在该对象中，把 CommandText 属性设置为一条 SQL INSERT 语句，用于把新记录添加到 Orders 表中。ExecuteNonQuery()方法执行这条 SQL 语句：

```
try
{
    var command = connection.CreateCommand();
    command.CommandText = "INSERT INTO Orders (CustomerId, " +
        "OrderDate, ShipAddress, ShipCity, ShipCountry)" +
        "VALUES(@CustomerId, @OrderDate, @ShipAddress, @ShipCity, " +
        "@ShipCountry)";
    command.Parameters.AddWithValue("@CustomerId", order.CustomerId);
    command.Parameters.AddWithValue("@OrderDate", order.OrderDate);
    command.Parameters.AddWithValue("@ShipAddress", order.ShipAddress);
    command.Parameters.AddWithValue("@ShipCity", order.ShipCity);
    command.Parameters.AddWithValue("@ShipCountry", order.ShipCountry);

    connection.Open();

    command.ExecuteNonQuery();
```

因为把 OrderId 定义为数据库中自动递增的值，把 Order Details 写入数据库时需要这个 ID，所以 OrderId 使用@@IDENTITY 读取。接着调用 SetOrderId()方法把它设置为 Order 对象：

```
command.CommandText = "SELECT @@IDENTITY AS 'Identity'";
object identity = command.ExecuteScalar();
order.SetOrderId(Convert.ToInt32(identity));
```

在订单写入数据库后，使用 OrderLineData 组件把订单的所有订单行都写入数据库中：

```
using (OrderLineData updateOrderLine = new OrderLineData())
{
    foreach (OrderLine orderLine in order.OrderLines)
    {
        updateOrderLine.Insert(order.OrderId, orderLine);
    }
}
```

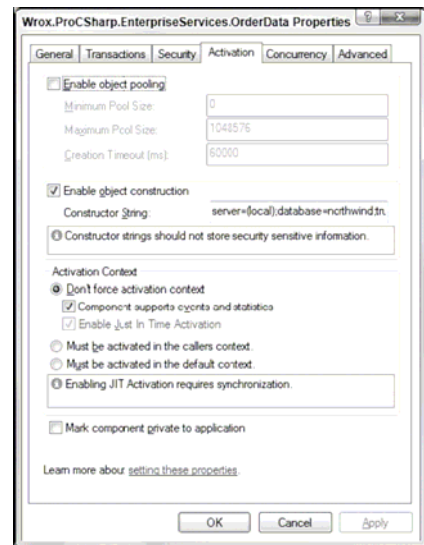


图 51-9

最后,无论 try 块中的代码成功执行,还是出现异常,都会关闭连接。

```
        finally
        {
            connection.Close();
        }
    }
}
```

51.7.4 OrderLineData 组件

OrderLineData 组件的实现类似于 OrderData 组件的实现。使用 ConstructionEnabled 特性定义数据库连接字符串:



```
using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;
using System.Data;
using System.Data.SqlClient;

namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderLineUpdate
    {
        void Insert(int orderId, OrderLine orderDetail);
    }

    [Transaction(TransactionOption.Required)]
    [EventTrackingEnabled(true)]
    [ConstructionEnabled(
        true, Default="server=(local);database=northwind;trusted_connection=true")]
    [ComVisible(true)]
    public class OrderLineData: ServicedComponent, IOrderLineUpdate
    {
        private string connectionString;

        protected override void Construct(string s)
        {
            connectionString = s;
        }
    }
}
```

代码段 NorthwindComponents/OrderLineData.cs

在本示例中,在 OrderLineData 类的 Insert()方法中,不使用 AutoComplete 特性作为阐述定义事务处理结果的另一种方式,而是说明如何使用 ContextUtil 类设置 consistent 位和 done 位。在 Insert()方法的最后,根据在数据库中插入数据是否成功,调用 SetComplete()方法。如果在抛出异常的地方有错误,SetAbort()方法就把 consistent 位设置为 false,这样该事务就和所有参与该事务处理的组件一起撤销。

```
public void Insert(int orderId, OrderLine orderDetail)
{
    var connection = new SqlConnection(connectionString);
    try
```

```

    {
        var command = connection.CreateCommand();
        command.CommandText = "INSERT INTO [Order Details] (OrderId, " +
            "ProductId, UnitPrice, Quantity)" +
            "VALUES(@OrderId, @ProductId, @UnitPrice, @Quantity)";
        command.Parameters.AddWithValue("@OrderId", orderId);
        command.Parameters.AddWithValue("@ProductId", orderDetail.ProductId);
        command.Parameters.AddWithValue("@UnitPrice", orderDetail.UnitPrice);
        command.Parameters.AddWithValue("@Quantity", orderDetail.Quantity);

        connection.Open();

        command.ExecuteNonQuery();

    }
    catch (Exception)
    {
        ContextUtil.SetAbort();
        throw;
    }
    finally
    {
        connection.Close();
    }
    ContextUtil.SetComplete();
}
}
}

```

51.7.5 客户端应用程序

构建组件后，就可以创建客户端应用程序。为了进行测试，可以创建一个控制台应用程序。在引用 NorthwindComponent 程序集和 System.EnterpriseServices 程序集后，就可以使用 Order.Create() 静态方法创建一个新的 order 对象。使用 order.AddOrderLine() 给该订单添加一个新订单行。OrderLine.Create() 方法接受产品 ID、价格和数量创建订单行。在实际应用程序中，添加 Product 类比使用产品 ID 会更有用，但本例仅用于演示一般的事务。

最后，创建 OrderControl 服务组件类，调用 NewOrder() 方法：



```

var order = Order.Create("PICCO", DateTime.Today, "Georg Pipp",
    "Salzburg", "Austria");
order.AddOrderLine(OrderLine.Create(16, 17.45F, 2));
order.AddOrderLine(OrderLine.Create(67, 14, 1));

using (var orderControl = new OrderControl())
{
    orderControl.NewOrder(order);
}

```

代码段 ClientApplication/Program.cs

可以尝试把不存在的产品写入 OrderLine(使用不在 Products 表中列出的产品 ID)中。在这种情况下，终止事务，并且不把数据写入数据库中。

在事务处于激活状态时，通过在组件服务管理器的树型视图中选择 Distributed Transaction Coordinator，可以查看事务，如图 51-10 所示。



可能必须给 OrderData 类的 Insert()方法添加睡眠时间,以查看活动事务;否则,事务处理就完成得太快,无法显示。



如果在服务组件事务中运行时对它进行调试,就应注意对于该服务组件,默认的事务超时是 60 秒。在 Component Services 浏览器中单击 My Computer, 选择 Action | Properties 命令, 打开 Options 选项卡, 就可以为整个系统修改这个默认值。除了更改整个系统的值,也可以为依次为每个组件选择 Transaction 选项, 配置事务超时。

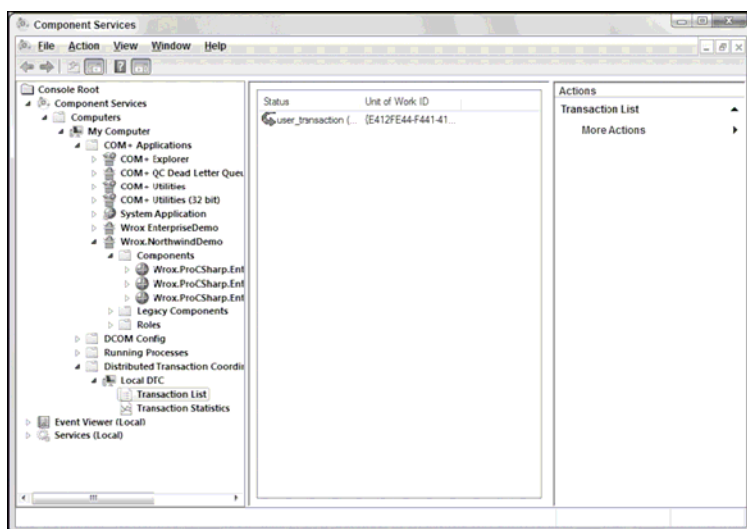


图 51-10

51.8 集成 WCF 和 Enterprise Services

Windows Communication Foundation(WCF)是.NET Framework 4 中首选的通信技术, WCF 详见第 43 章。 .NET Enterprise Services 提供了一个与 WCF 集成的巨大模型。

51.8.1 WCF 服务外观

给 Enterprise Services 应用程序添加 WCF 外观, 可以使用 WCF 客户端访问服务组件。除了使用 DCOM 协议之外, 还可以通过 WCF 使用不同的协议, 如 HTTP(含 SOAP)、TCP(二进制格式)。

要在 Visual Studio 2010 中创建 WCF 外观, 可以选择 Tools | WCF Services Configuration Editor 命令, 再选择 File | Integrate | COM+ Application 命令, 最后选择 COM+应用程序 Wrox.NorthwindDemo、Wrox.ProCSharp.EnterpriseServices.OrderControl 组件和 IOrderControl 接口, 如图 51-11 所示。



除了使用 Visual Studio 创建 WCF 外观之外，还可以使用命令行实用程序 comsvc-config.exe。这个实用程序在 <Windows> \Microsoft.NET\ Framework\v4.0 目录下。

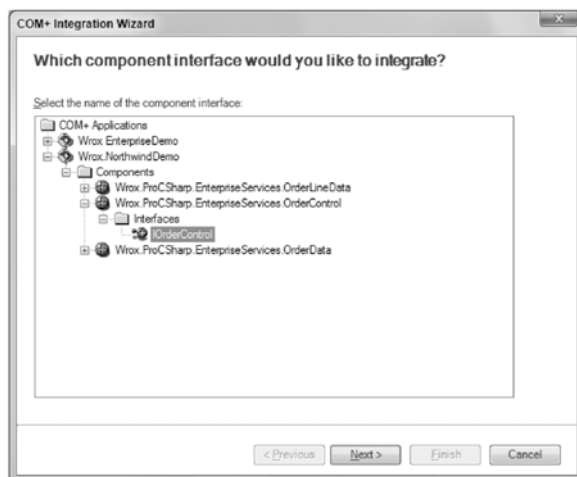


图 51-11

在下一个对话框中，可以选择 IOrderControl 接口中可用于 WCF 客户端的所有方法。IOrderControl 接口只有一个方法即 NewOrder()，如下所示。

下一个对话框如图 51-12 所示，它允许配置宿主选项。在宿主选项中，可以指定 WCF 服务运行在哪个进程中。选中 COM+ hosted 单选框时，WCF 外观运行在 COM+ 的 dllhost.exe 进程中。只有在应用程序配置为服务器应用程序 ApplicationActivation(ActivationOption.Server)时才可以配置这个选项。

Web hosted 选项指定 WCF 信道在 IIS 的一个进程或 WAS (Windows Activation Services, Windows 活动服务)工作进程中侦听。WAS 是 Windows Vista 和 Windows Server 2008 中新增的。选中 Web hosted in-process 单选框表示 Enterprise Services 组件库运行在 IIS 或 WAS 工作进程中。只有在应用程序配置为库应用程序 ApplicationActivation (ActivationOption.Library)时这个配置才可用。

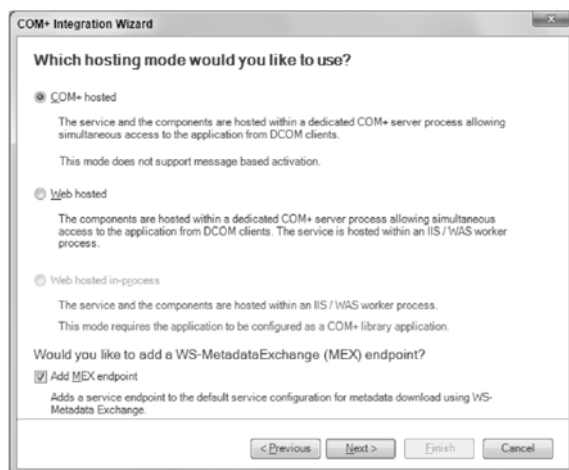


图 51-12

勾选 Add MEX endpoint 复选框，会把一个 MEX(Metadata Exchange，元数据交换)端点添加到 WCF 配置文件中，这样客户端程序员就可以使用 WS-MEX 访问服务的元数据。



MEX 参见第 43 章。

在如图 51-13 所示的下一个对话框中，可以指定通信模式，访问 WCF 外观。根据需要，如果客户端跨防火墙访问服务，或者需要独立于平台的通信，HTTP 就是最佳选择。TCP 为 .NET 客户端提供了跨计算机的更快通信，如果客户端应用程序运行在与服务相同的系统上，命名管道就是最快的选项。

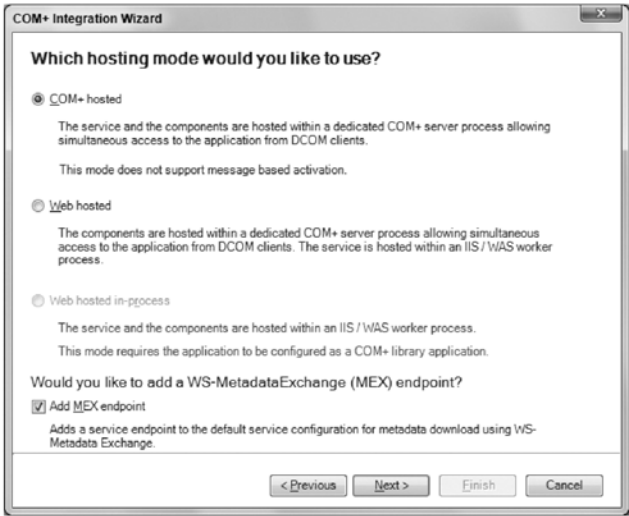


图 51-13

下一个对话框需要服务的基地址，该服务取决于所选择的通信协议，如图 51-14 所示。

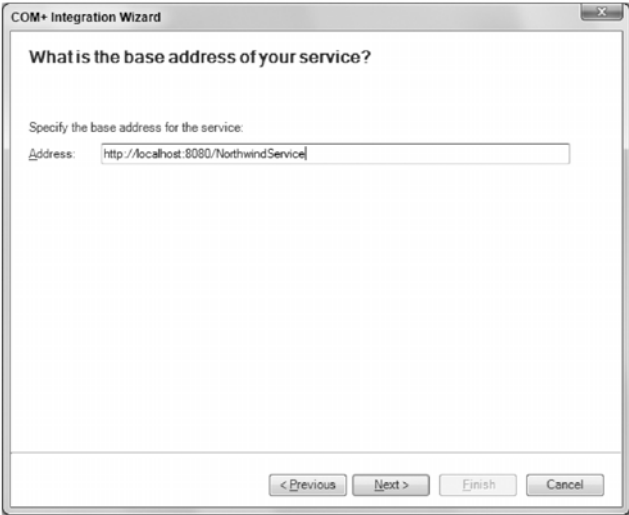


图 51-14

最后一个对话框显示端点配置的位置。配置的基目录是<Program Files>\ComPlus Applications，其后是应用程序的唯一 ID。这个目录包含 application.config 文件。这个配置文件列出 WCF 的行为和端点。

<service>元素指定带端点配置的 WCF 服务。因为用 comTransactionalBinding 配置把该绑定设置为 wsHttpBinding，所以事务可以从调用程序流向服务组件。利用其他网络和客户端要求，可以指定另一个绑定，详见第 43 章。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="ComServiceMexBehavior">
          <serviceMetadata httpGetEnabled="true"/>
          <serviceDebug/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <bindings>
      <netNamedPipeBinding>
        <binding name="comNonTransactionalBinding"/>
        <binding name="comTransactionalBinding" transactionFlow="true"/>
      </netNamedPipeBinding>
      <wsHttpBinding>
        <binding name="comNonTransactionalBinding"/>
        <binding name="comTransactionalBinding" transactionFlow="true"/>
      </wsHttpBinding>
    </bindings>
    <comContracts>
      <comContract contract="{E1B02E09-EE48-3B6B-946F-E6A8BAEC6340}"
        name="IOrderControl"
        namespace=
          "http://tempuri.org/E1B02E09-EE48-3B6B-946F-E6A8BAEC6340"
        requiresSession="true">
        <exposedMethods>
          <add exposedMethod="NewOrder"/>
        </exposedMethods>
      </comContract>
    </comContracts>
    <services>
      <service behaviorConfiguration="ComServiceMexBehavior"
        name="{196F39D0-4F47-454A-BC16-955C2C54B6F5},
          {A16C0740-C2A0-38C9-9FD3-7C583B3B42FA}">
        <endpoint address="IOrderControl" binding="wsHttpBinding"
          bindingConfiguration="comTransactionalBinding"
          contract="{E1B02E09-EE48-3B6B-946F-E6A8BAEC6340}"/>
        <endpoint address="mex" binding="mexHttpBinding"
          contract="IMetadataExchange"/>
      </service>
    </services>
    <host>
      <baseAddresses>
        <add baseAddress=
          "net.pipe://localhost/Wrox.NorthwindDemo/Wrox.ProCSharp.
            EnterpriseServices.OrderControl"/>
      </baseAddresses>
    </host>
  </system.serviceModel>
</configuration>
```

```

        <add baseAddress=
            "http://localhost:8088/NorthwindService"/>
        </baseAddresses>
    </host>
</service>
</services>
</system.serviceModel>
</configuration>

```

在启动服务器应用程序之前，需要修改安全性，允许用户运行应用程序，注册侦听端口。否则普通的用户就不能注册侦听器端口。在 Windows 7 上，可以用 netsh 命令来注册。http 选项更改 HTTP 协议的 ACL。端口号和服务名用 URL 定义，用户选项指定启动侦听器服务的用户名。

```
netsh http add urlacl url=http://+:8088/NorthwindService user=username
```

51.8.2 客户端应用程序

创建一个新的控制台应用程序 WCFClientApp。由于服务提供一个 MEX 端点，因此可以选择 Visual Studio 中的 Project | Add Service Reference 命令，添加一个服务引用，如图 51-15 所示。

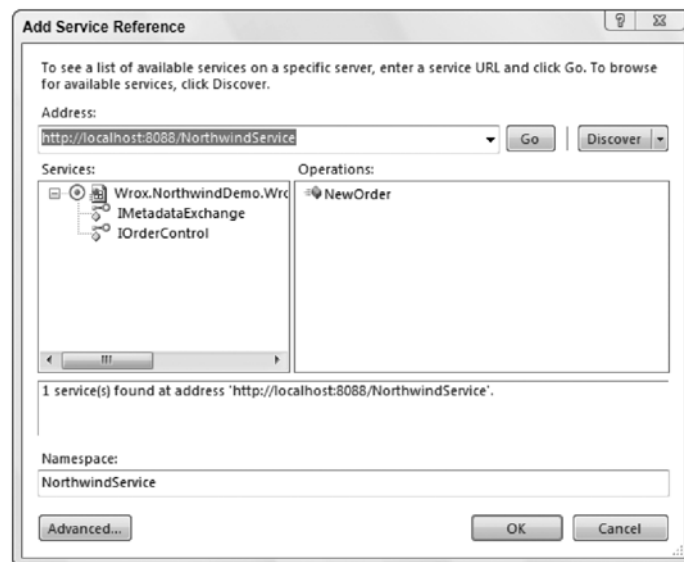


图 51-15



如果该服务驻留在 COM+ 中，就必须在访问 MEX 数据之前启动应用程序。如果服务驻留在 WAS 中，应用程序会自动启动。

利用服务引用创建一个代理类，引用 System.ServiceModel 和 System.Runtime.Serialization 程序集，把引用服务的应用程序配置文件添加到客户端应用程序中。

现在就可以使用生成的实体类和代理类 OrderControlClient，给服务组件发送一个订单请求。

```

static void Main()
{

```



```

var order = new Order
{
    CustomerId = "PICCO",
    OrderDate = DateTime.Today,
    ShipAddress = "Georg Pippis",
    ShipCity = "Salzburg",
    ShipCountry = "Austria"
};
var line1 = new OrderLine
{
    ProductId = 16,
    UnitPrice = 17.45F,
    Quantity = 2
};
var line2 = new OrderLine
{
    ProductId = 67,
    UnitPrice = 14,
    Quantity = 1
}
OrderLine[] orderLines = { line1, line2 };
order.orderLines = orderLines;

var occ = new OrderControlClient();
occ.NewOrder(order);
}

```

51.9 小结

本章讨论了 Enterprise Services 提供的丰富功能，如自动事务处理、对象池、队列组件和松散耦合事件。

为了创建服务组件，必须引用 System.EnterpriseServices 程序集。所有服务组件的基类都是 ServicedComponent。在这个类中，上下文就可以截获方法调用。可以使用特性指定要使用的截获功能。我们还学习了如何使用特性配置应用程序及其组件，如何使用[Transaction]特性管理事务，以及指定组件的事务要求。本章还讨论了如何创建 WCF 外观，使 Enterprise Services 与新通信技术 WCF 集成。

本章介绍了如何使用操作系统提供的 Enterprise Services 功能。下一章将讨论如何使用操作系统中另一个用于通信的功能：消息队列。