

```

aes.Key = symmKey;
aes.IV = iv;

using (ICryptoTransform decryptor = aes.CreateDecryptor())
using (MemoryStream ms = new MemoryStream())
{
    var cs = new CryptoStream(ms, decryptor, CryptoStreamMode.Write);
    cs.Write(encryptedData, nBytes, encryptedData.Length - nBytes);
    cs.Close();

    rawData = ms.ToArray();

    Console.WriteLine("Bob decrypts message to: {0}",
        Encoding.UTF8.GetString(rawData));
}
aes.Clear();
}
}

```

运行应用程序，会在控制台上看到如下输出。来自 Alice 的消息被加密，Bob 用安全交换的对称密钥解密它。

```

Alice sends message: secret message
Alice creates this symmetric key with Bobs public key information:
5NWat8AemzFCYolIIae9S3Vn4AXyai4aL8ATFo41vbw=
Alice: message is encrypted: 3C5U9CpYxnoFTk3Ew2V0T5Po0Jgryc5R7Te8zt5N0=

Bob receives encrypted message
Bob creates this symmetric key with Alices public key information:
5NWat8AemzFCYolIIae9S3Vn4AXyai4aL8ATFo41vbw=
Bob decrypts message to: secret message

```

21.3 资源的访问控制

在操作系统中，资源(如文件和注册表键，以及命名管道的句柄)都使用访问控制列表来保护。图 21-3 显示了这个映射的结构。资源有一个关联的安全描述符。安全描述符包含了资源拥有者的信息，并引用了两个访问控制列表：自由访问控制列表(DACL)和系统访问控制列表(SACL)。DACL 用来确定谁有访问权，或者谁没有访问权；SACL 用来确定安全事件日志的审核规则。ACL 包含一个访问控制项(ACE)列表。ACE 包含类型、安全标识符和权限。在 DACL 中，ACE 的类型可以是允许访问或拒绝访问。可以用文件设置和获得的权限是创建、读取、写入、删除、修改、改变许可和获得拥有权。

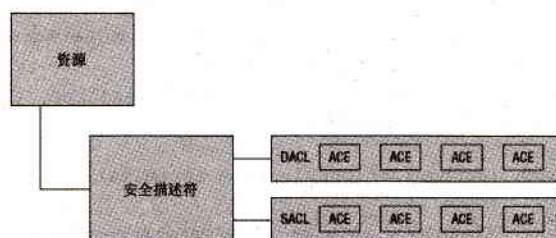


图 21-3

读取和修改访问控制的类在 `System.Security.AccessControl` 名称空间中。下面的程序说明了如何从文件中读取访问控制列表。

`FileStream` 类定义了 `GetAccessControl()` 方法, 该方法返回一个 `FileSecurity` 对象。`FileSecurity` 是一个 .NET 类, 它表示文件的安全描述符。`FileSecurity` 类派生自基类 `ObjectSecurity`、`CommonObjectSecurity`、`NativeObjectSecurity` 和 `FileSystemSecurity`。其他表示安全描述符的类有 `CryptoKeySecurity`、`EventWaitHandleSecurity`、`MutexSecurity`、`RegistrySecurity`、`SemaphoreSecurity`、`PipeSecurity` 和 `ActiveDirectorySecurity`。所有这些对象都可以使用访问控制列表保护。一般情况下, 对应的 .NET 类定义了 `GetAccessControl()` 方法, 返回相应的安全类; 例如, `Mutex.GetAccessControl()` 方法返回一个 `MutexSecurity` 类, `PipeStream.GetAccessControl()` 方法返回一个 `PipeSecurity` 类。

`FileSecurity` 类定义了读取、修改 DACL 和 SACL 的方法。`GetAccessRules()` 方法以 `AuthorizationRuleCollection` 类的形式返回 DACL。要访问 SACL, 可以使用 `GetAuditRules()` 方法。

在 `GetAccessRules()` 方法中, 可以确定是否应使用继承的访问规则(不仅仅是用对象直接定义的访问规则)。最后一个参数定义了应返回的安全标识符的类型。这个类型必须派生自基类 `IdentityReference`。可能的类型有 `NTAccount` 和 `SecurityIdentifier`。这两个类都表示用户或组。`NTAccount` 类按名称查找安全对象, `SecurityIdentifier` 类按唯一的安全标识符查找安全对象。

返回的 `AuthorizationRuleCollection` 包含 `AuthorizationRule` 对象。`AuthorizationRule` 对象是 ACE 的 .NET 表示。在这里的例子中, 因为访问一个文件, 所以 `AuthorizationRule` 对象可以强制转换为 `FileSystemAccessRule` 类型。在其他资源的 ACE 中, 存在不同的 .NET 表示, 例如 `MutexAccessRule` 和 `PipeAccessRule`。在 `FileSystemAccessRule` 类中, `AccessControlType`、`FileSystemRights` 和 `IdentityReference` 属性返回 ACE 的相关信息。



可从
wrox.com
下载源代码

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        static void Main(string[] args)
        {
            string filename = null;
            if (args.Length == 0)
                return;

            filename = args[0];

            FileStream stream = File.Open(filename, FileMode.Open);
            FileSecurity securityDescriptor = stream.GetAccessControl();
            AuthorizationRuleCollection rules =
                securityDescriptor.GetAccessRules(true, true,
                    typeof(NTAccount));

            foreach (AuthorizationRule rule in rules)
            {
                var fileRule = rule as FileSystemAccessRule;
                Console.WriteLine("Access type: {0}", fileRule.AccessControlType);
            }
        }
    }
}
```

```

Console.WriteLine("Rights: {0}", fileRule.FileSystemRights);
Console.WriteLine("Identity: {0}",
    fileRule.IdentityReference.Value);
Console.WriteLine();

```

代码段 FileAccessControl/Program.cs

运行应用程序，并传递一个文件名，就可以看到文件的访问控制列表。这里的输出列出了管理员和系统的全部控制权限、通过身份验证的用户的修改权限，以及属于 Users 组的所有用户的读取和执行权限。

```

Access type: Allow
Rights: FullControl
Identity: BUILTIN\Administrators

Access type: Allow
Rights: FullControl
Identity: NT AUTHORITY\SYSTEM

Access type: Allow
Rights: Modify, Synchronize
Identity: NT AUTHORITY\Authenticated Users

Access type: Allow
Rights: ReadAndExecute, Synchronize
Identity: BUILTIN\Users

```

设置访问权限非常类似于读取访问权限。要设置访问权限，几个可以得到保护的资源类提供了 `SetAccessControl()` 和 `ModifyAccessControl()` 方法。这里的示例代码调用 `File` 类的 `SetAccessControl()` 方法，以修改文件的访问控制列表。给这个方法传递一个 `FileSecurity` 对象。`FileSecurity` 对象用 `FileSystemAccessRule` 对象填充。这里列出的访问规拒绝 Sales 组的写入访问权限，给 Everyone 组提供了读取访问权限，并给 Developers 组提供了全部控制权限。



只有定义了 Windows 组 Sales 和 Developers，这个程序才能在系统上运行。可以修改程序，使用自己环境下的可用组。

```

private static void WriteAcl(string filename)
{
    var salesIdentity = new NTAccount("Sales");
    var developersIdentity = new NTAccount("Developers");
    var everyoneIdentity = new NTAccount("Everyone");

    var salesAce = new FileSystemAccessRule(salesIdentity, FileSystemRights.Write,
        AccessControlType.Deny);
    var everyoneAce = new FileSystemAccessRule(everyoneIdentity,
        FileSystemRights.Read,
        AccessControlType.Allow);
    var developersAce = new FileSystemAccessRule(developersIdentity,
        FileSystemRights.FullControl,

```

```

AccessControlType.Allow);

var securityDescriptor = new FileSecurity();
securityDescriptor.SetAccessRule(everyoneAce);
securityDescriptor.SetAccessRule(developersAce);
securityDescriptor.SetAccessRule(salesAce);
File.SetAccessControl(filename, securityDescriptor);
}

```



打开 Properties 窗口，在 Windows 资源管理器中选择一个文件，选择 Security 选项卡，列出访问控制列表，就可以验证访问规则。

21.4 代码访问安全性

代码访问安全性的重要性是什么？在基于角色的安全性中，可以定义用户允许做什么。代码访问安全性指定了代码能做什么。.NET 4 简化了这个模型，删除了直到.NET 3.5 都存在的复杂的策略配置，添加了第 2 级安全透明性。安全透明级以前就存在，但第 2 级是.NET 4 新增的。安全透明区分开了允许进行授权调用(如调用本地代码)的代码和不允许进行授权调用的代码。代码分为 3 类：

- “security-critical(安全第一)”代码可以运行任意代码，这种代码不能由透明代码调用。
- “safe-critical(安全重要)”代码可以由透明代码调用，安全验证由这种代码执行。
- 透明(Transparent)代码可以执行的操作非常有限。这种代码只能运行在指定的权限集中，且运行在沙盒中。它不能包含不安全或无法验证的代码，也不能调用 security-critical 代码。

如果编写 Windows 应用程序，就不应用受限的代码权限。运行在桌面上的应用程序有完全信任权限，且可以包含任意代码。沙盒可用于 Silverlight 应用程序和 ASP.NET 应用程序，这些应用程序驻留在 Web 提供程序中，或者有自定义功能，如用 Managed Add-in Framework 运行插件。



Managed Add-in Framework 详见 Wrox 网站上的第 50 章和随书附赠光盘。

21.4.1 第 2 级安全透明性

可以用 SecurityRules 特性注解程序集，并设置 SecurityRuleSet.Level2，以应用.NET 4 新增的级别(这是.NET 4 中的默认级别)。为了向后兼容，可以把它设置为 Level1。

```
[assembly: SecurityRules(SecurityRuleSet.Level2)]
```

如果设置 SecurityTransparent 特性，完整的程序集就不会执行任何授权或不安全的操作。这个程序集只能调用其他透明代码或 safe-critical 代码。这个特性只能应用于完整的程序集。

```
[assembly: SecurityTransparent()]
```

AllowPartiallyTrustedCallers 特性位于透明代码和其他类别的代码之间。使用这个特性，代码默认为透明的，但个别类型和成员可以有其他特性：

```
[assembly: AllowPartiallyTrustedCallers()]
```

如果没有应用上述任何特性, 代码就是 `security-critical` 代码。但是, 可以给个别类型和成员应用 `SecuritySafeCritical` 特性, 从而使它们可以由透明代码调用。

```
[assembly: SecurityCritical()]
```

21.4.2 权限

如果代码运行在沙盒中, 沙盒就可以定义 .NET 权限, 以定义代码允许执行的操作。运行在桌面上的应用程序有完全信任权限, 而运行在沙盒中的应用程序只允许执行主机给沙盒授予的权限所定义的操作。还可以为从桌面应用程序中启动的应用程序域定义权限, 这需要使用沙盒 API。



应用程序域详见第 18 章。

权限是允许(或禁止)每个代码组执行的动作。例如, 权限包括“读取文件系统上的文件”、“写入 Active Directory”和“使用套接字打开网络连接”等。尽管有几个预定义的权限, 但也可以创建自己的权限。

.NET 权限独立于操作系统权限。NET 权限仅由 CLR 验证。程序集需要特定操作的权限(例如, `File` 类需要 `FileIOPermission`), CLR 验证该程序集是否被授予了该权限, 以便它可以继续执行。

可以应用于程序集或从代码中申请的权限有非常精细的权限列表。以下列表列出了 CLR 提供的代码访问权限。从中可以看出, 使用这些权限, 可以很好地控制代码允许做什么和不允许做什么:

- **DirectoryServicesPermission**——通过 `System.DirectoryServices` 类控制访问 Active Directory 的能力
- **DnsPermission**——控制使用 TCP/IP 域名系统(DNS)的能力
- **EnvironmentPermission**——控制读写环境变量的能力
- **EventLogPermission**——控制读写事件日志的能力
- **FileDialogPermission**——控制访问用户在 `Open` 对话框中访问已选择的文件的能力。该权限通常用于没有赋予 `FileIOPermission` 权限, 不能对文件进行有限的访问时
- **FileIOPermission**——控制处理文件的能力(其中包括读文件、写文件、向文件追加内容, 创建、更改和访问文件夹)
- **IsolatedStorageFilePermission**——控制访问私有虚拟文件系统的能力
- **IsolatedStoragePermission**——控制访问孤立存储器的能力, 存储器与个别用户相关, 并具有代码的标识的一些特征, 孤立存储器详见第 29 章
- **MessageQueuePermission**——控制通过 Microsoft Message Queue 使用消息队列的能力
- **PerformanceCounterPermission**——控制利用性能计数器的能力
- **PrintingPermission**——控制打印的能力
- **ReflectionPermission**——控制使用 `System.Reflection` 在运行时查找类型信息的能力
- **RegistryPermission**——控制读、写、创建和删除注册表键和值的能力
- **SecurityPermission**——控制执行、断言权限、调用非托管的代码、忽略验证和其他权力的能力
- **ServiceControllerPermission**——控制 Windows 服务的能力

- **SocketPermission**——控制在网络传输地址上创建或接受 TCP/IP 连接的能力
- **SQLClientPermission**——控制使用 SQL Server 的 .NET 数据提供程序访问 SQL Server 数据库的能力
- **UIPermission**——控制访问用户界面的能力
- **WebPermission**——控制连接 Web 或接受与 Web 之间连接的能力

对于上面的每一个权限类，通常可以指定级别更高的粒度。例如，**DirectoryServicesPermission** 类可以区分读和写访问权限，也可以确定允许访问或拒绝访问目录服务中的哪些项。

1. 权限集

权限集是权限的集合。在权限集中，不需要把每个权限都应用于代码；权限组合为权限集。例如，有 **FullTrust** 权限的程序集拥有对所有资源的全部访问权限。有内联网权限的程序集是受限的，也就是说，除了使用隔离的存储器之外，不能写入文件系统。可以创建包含所需权限的自定义权限集。

把权限赋予代码组，就不需要单独处理每个程序集。通常在程序块中应用权限，这就是 .NET 提供权限集合的概念的原因。这些是组合为指定的集合的代码访问权限列表，以下列表解释了可以即装即用的 7 个已命名权限集：

- **FullTrust**——没有权限的限制
- **SkipVerification**——不进行验证
- **Execution**——运行受保护的资源，但是不能访问它们
- **Nothing**——没有授予任何权限，代码不能执行
- **LocalIntranet**——指定权限全集的一个子集。例如，文件 IO 只能在程序集源自的共享上进行读取访问。在 .NET 3.5 和以前版本(在 .NET 3.5 SP1 之前)中，这个权限集在应用程序运行在网络共享上时使用
- **Internet**——未知来源的代码的默认策略，这是限制最严格的策略。例如，在这个权限集合下执行的代码没有文件 IO 能力，不能读写事件日志，也不能读写环境变量
- **Everything**——授予这个集合中列出的所有权限，除了包括忽略代码验证的权限。管理员可以改变这个权限集中的权限。默认策略要更严格时，可以使用这个权限集



只能修改 **Everything** 权限集的定义，而其他权限集是固定的，不能改变。当然，还可以创建自己的权限集。

2. 通过编程要求权限

程序集可以用声明或编程的方式要求权限。下面的代码段说明了如何使用 **DemandFileIOPermission()** 方法要求权限。如果导入 **System.Security.Permissions** 名称空间，就可以创建一个 **FileIOPermission** 对象，并调用这个对象的 **Demand()** 方法，以检查权限。这将验证方法的调用者，这里是 **DemandFileIOPermission()** 方法的调用者，是否具有必要的权限。如果 **Demand()** 方法失败，就抛出一个 **SecurityException** 类型的异常。可以不捕获异常并让调用者处理它。



可从
wrox.com
下载源代码

```
using System;
using System.Security;
using System.Security.Permissions;

[assembly: AllowPartiallyTrustedCallers()]

namespace Wrox.ProCSharp.Security
{
    [SecuritySafeCritical]
    public class DemandPermissions
    {
        public void DemandFileIOPermissions(string path)
        {
            var fileIOPermission = new FileIOPermission(PermissionState.Unrestricted);
            fileIOPermission.Demand();

            //...
        }
    }
}
```

代码段 DemandPermissionDemo/DemandPermissions.cs

FileIOPermission 类包含在 **System.Security.Permissions** 名称空间中。这个名称空间包含了权限的全集,除此之外,它还为声明性的权限特性提供了类,为用于创建权限对象的参数提供了枚举(例如,在创建 **FileIOPermission** 类时,指定是需要完全访问还是只读访问)。

当代码试图违反被赋予的权限时,要捕获 CLR 抛出的异常,可以捕获 **SecurityException** 类型的异常,通过它可以访问许多条有用的信息,其中包括可读的栈踪迹(**SecurityException.StackTrace**)和对抛出异常的方法的引用(**SecurityException.TargetSite**)。**SecurityException** 异常甚至还能提供 **SecurityException.PermissionType** 属性,这个属性返回导致发生安全异常的 **Permission** 对象的类型。

如果仅对文件 I/O 使用 .NET 类,就不需要请求 **FileIOPermission** 权限,因为 .NET 类在进行文件 I/O 时会要求该权限。但如果封装了本地 API 调用,如 **CreateFileTransacted()** 方法,就需要自己请求该权限。另外,还可以使用这个机制从调用者那里请求自定义权限。

3. 使用沙盒 API 包含未授权的代码

桌面应用程序默认有完全信任权限。使用沙盒 API,可以创建一个不具备完全信任权限的应用程序域。

为了说明沙盒 API 的用法,先创建一个 C# 库项目 **RequiredFileIOPermissionDemo**。这个库包含 **RequiredPermissionDemo** 类和 **RequiredFilePermissions()** 方法。这个方法根据代码是否有文件权限,返回 **true** 或 **false**。在这段实现代码中, **File** 类创建了一个文件,其中通过参数把路径传送给 **path** 变量。如果文件写入操作失败,就抛出一个 **SecurityException** 类型的异常。**File** 类检查 **FileIOSecurity**,与前面的 **DemandPermissionDemo** 示例相同。如果安全检查失败, **FileIOSecurity** 类的 **Demand()** 方法就抛出一个 **SecurityException** 异常。这里捕获 **SecurityException** 异常,以从 **RequiredFilePermissions()** 方法中返回 **false**。



可从
wrox.com
下载源代码

```
using System;
using System.IO;
using System.Security;

[assembly: AllowPartiallyTrustedCallers()]
```

```

namespace Wrox.ProCSharp.Security
{
    [SecuritySafeCritical]
    public class RequirePermissionsDemo : MarshalByRefObject
    {
        public bool RequireFilePermissions(string path)
        {
            bool accessAllowed = true;

            try
            {
                StreamWriter writer = File.CreateText(path);
                writer.WriteLine("written successfully");
                writer.Close();
            }
            catch (SecurityException)
            {
                accessAllowed = false;
            }

            return accessAllowed;
        }
    }
}

```

代码段 `RequireFileIOPermissionsDemo/RequirePermissionsDemo.cs`

使用沙盒 API 的宿主应用程序是 `AppDomainHost` 项目，它是一个简单的 C# 控制台应用程序。沙盒 API 是 `AppDomain.CreateDomain()` 方法的一个重载版本，它在沙盒中创建了一个新的应用程序域。这个方法需要 4 个参数，包括应用程序域的名称、从当前应用程序域中提取的凭证、`AppDomainSetup` 信息和一个权限集。所创建的权限集只包含 `SecurityException` 异常和 `SecurityPermissionFlag.Execution` 标记，以便允许执行代码——除以之外没有别的目的。在新的沙盒应用程序域中，实例化 `DemandPermission` 程序集中 `DemandPermissions` 类型的对象。

跨应用程序域的调用需要 .NET Remoting。因此 `RequirePermissionDemo` 类需要派生自基类 `MarshalByRefObject`。对返回的 `ObjectHandle` 进行解包，会对其他应用程序域中的对象返回一个透明代理，以调用 `RequiredFilePermissions()` 方法。



.NET Remoting 参见 Wrox 网站上的第 54 章和随书附赠光盘。



可从
wrox.com
下载源代码

```

using System;
using System.Runtime.Remoting;
using System.Security;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
{
    class Program
    {
        static void Main()
        {

```

```

PermissionSet permSet = new PermissionSet(PermissionState.None);
permSet.AddPermission(new SecurityPermission(
    SecurityPermissionFlag.Execution));

AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
AppDomain newDomain = AppDomain.CreateDomain(
    "Sandboxed domain", AppDomain.CurrentDomain.Evidence, setup, permSet);
ObjectHandle oh = newDomain.CreateInstance("RequireFileIOPermissionsDemo",
    "Wrox.ProCSharp.Security.RequirePermissionsDemo");
object o = oh.Unwrap();
var io = o as RequirePermissionsDemo;
string path = @"c:\temp\file.txt";
Console.WriteLine("has {0}permissions to write to {1}",
    io.RequireFilePermissions(path) ? null : "no ",
    path);
}
}

```

代码段 AppDomainHost/Program.cs

运行这个应用程序，可以看到，结果是所调用的程序集没有创建文件的权限。如果在所创建的应用程序域中给权限集添加 `FileIOPermissionSet`，如下面的代码所示，写入文件的操作就会成功。

```

PermissionSet permSet = new PermissionSet(PermissionState.None);
permSet.AddPermission(new SecurityPermission(
    SecurityPermissionFlag.Execution));
permSet.AddPermission(new FileIOPermission(
    FileIOPermissionAccess.AllAccess, "c:/temp"));

```

4. 隐式的权限

在授予权限时，通常有一条隐式的语句也可以赋予其他权限。例如，如果赋予了访问 C:\ 的权限 `FileIOPermission`，就有一个也可以访问 C:\ 的子目录的隐式假设。

如果要检查授予的权限是否以子集的方式隐式地赋予了其他的权限，就可以使用下面的代码：



可从
wrox.com
下载源代码

```

class Program
{
    static void Main()
    {
        CodeAccessPermission permissionA =
            new FileIOPermission(FileIOPermissionAccess.AllAccess, @"C:\");
        CodeAccessPermission permissionB =
            new FileIOPermission(FileIOPermissionAccess.Read, @"C:\temp");
        if (permissionB.IsSubsetOf(permissionA))
        {
            Console.WriteLine("PermissionB is a subset of PermissionA");
        }
    }
}

```

代码段 ImplicitPermissions/Program.cs

代码的执行结果如下：

```
PermissionB is a subset of PermissionA
```

21.5 使用证书发布代码

可以利用数字证书来对程序集进行签名,让软件的消费者验证软件发布者的身份。根据使用应用程序的地点,可能需要证书。例如,用户利用 ClickOnce 安装应用程序,可以验证证书,以信任发布者。Microsoft 通过 Windows Error Reporting,使用证书来找出哪个供应商映射到错误报告。



ClickOnce 参见第 17 章。Windows Error Reporting 参见附录 A。

在商业环境中,可以从 Verisign 或 Thawte 之类的公司中获取证书。从软件厂商购买证书(而不是创建自己的证书)的优点是,那些证书可以证明软件的真实性和有很高的可信度,软件厂商是可信的第三方。但是,为了测试,.NET 提供了一个命令行实用程序,使用它可以创建测试证书。创建证书和使用证书发布软件的过程相当复杂,但是本节用一个简单的示例说明这个过程。

设想有一个名叫 ABC 的公司。公司的软件产品(Simple.exe)应该值得信赖。首先,创建一个测试证书,方法是输入下面的命令:

```
makecert -sv abckey.pvk -r -n "CN=ABC Corporation" abccorptest.cer
```

这条命令为 ABC 公司创建了一个测试证书,并把它保存到 abccorptest.cer 文件中。-sv abckey.pvk 参数创建一个密钥文件,来存储私钥。在创建密钥文件时,需要输入一个必须记住的密码。

创建证书后,就可以用软件发布者证书测试工具(Cert2spc.exe)创建一个软件发布者测试证书:

```
>cert2spc abccorptest.cer abccorptest.spc
```

有了存储在 spc 文件中的证书和存储在 pvk 文件中的密钥文件,就可以用 pvk2pfx 实用程序创建一个包含证书和密钥文件的 pfx 文件:

```
>pvk2pfx -pvk abckey.pvk -spc abccorptest.spc -pfx abccorptest.pfx
```

现在可以用 signtool.exe 实用程序标记程序集了。使用 sign 选项来标记,用 -f 指定 pfx 文件中的证书,用 -v 指定输出详细信息:

```
>signtool sign -f abccorptest.pfx -v simple.exe
```

为了建立对证书的信任,可使用证书管理器 certmgr 或 MMC 插件 Certificates,通过 Trusted Root Certification Authorities 和 Trusted Publishers 安装它。之后就可以使用 signtool 验证签名是否成功:

```
>signtool verify -v -a simple.exe
```

21.6 小结

本章讨论了与 .NET 应用程序相关的几个安全性方面。用基于角色的安全性进行身份验证和授权,可以确定哪些用户可以访问应用程序中的特性。用户用标识和主体表示,这些类实现了 IIdentity 和 IPrincipal 接口。角色的验证可以在代码中完成,也可以使用特性以简单的方式完成。

本章介绍了加密方法,说明了数据的签名和加密,以安全的方式交换密钥。.NET 提供几个加密算法,包括对称加密算法和非对称加密算法。

使用访问控制列表还可以确定如何读取和修改对操作系统资源(如文件)的访问权限。ACL 的编程方式与安全管道、注册表键、Active Directory 项以及许多其他操作系统资源的编程方式相同。

如果应用程序在不同的区域和不同的语言中使用,就可以参阅下一章介绍的.NET 的全球化和本地化功能。

第 22 章

本地化

本章内容:

- 使用表示区域性和区域的类
- 应用程序的全球化
- 应用程序的本地化

价值 1.25 亿美元的 NASA 的火星气象卫星在 1999 年 9 月 23 日失踪了,其原因是一个工程组为一个关键的太空操作以米为单位,而另一个工程组以英寸单位。当编写的应用程序要在世界各国发布时,必须考虑不同的区域性和区域。

不同的区域性在日历、数字和日期格式上各不相同。按照字母 A~Z 给字符串排序也会导致不同的结果,因为不同的文化差异。为了使应用程序可应用于全球市场,就必须对应用程序进行全球化和本地化。

本章将介绍 .NET 应用程序的全球化和本地化。全球化(globalization)用于国际化的应用程序:使应用程序可以在国际市场上销售。采用全球化策略,应用程序应根据区域性、不同的日历等支持不同的数字和日期格式。本地化(localization)用于为特定的区域性翻译应用程序。而字符串的翻译可以使用资源,如 .NET 资源或 WPF 资源字典。

.NET 支持 Windows 和 Web 应用程序的全球化和本地化。要使应用程序全球化,可以使用 System.Globalizatiom 名称空间中的类;要使应用程序本地化,可以使用 System.Resources 名称空间支持的资源。

22.1 System.Globalizatiom 名称空间

System.Globalizatiom 名称空间包含了所有的区域性类和区域类,以支持不同的日期格式、不同的数字格式,甚至如 GregorianCalendar 类、HebrewCalendar 类和 JapaneseCalendar 类等表示的不同日历。使用这些类可以根据不同的地区显示不同的表示法。

本节讨论使用 System.Globalizatiom 名称空间时要考虑的如下问题:

- Unicode 问题
- 区域性和区域
- 显示所有区域性及其特征的例子
- 排序

22.1.1 Unicode 问题

因为一个 Unicode 字符有 16 位, 所以共有 65 536 个 Unicode 字符。这对于当前在信息技术中使用的所有语言够用吗? 例如, 汉语就需要 80 000 多个字符。但是, Unicode 可以解决这个问题。使用 Unicode 必须区分基本字符和组合字符。可以给一个基本字符添加若干个组合字符, 组成一个可显示的字符或一个文本元素。

例如, 冰岛的字符 Ogonek, 就可以使用基本字符 0x006F(拉丁小字母 o)、组合字符 0x0328(组合 Ogonek)和 0x0304(组合 Macron)组合而成, 如图 22-1 所示。组合字符在 0x0300~0x0345 之间定义, 对于美国和欧洲市场, 预定义字符有助于处理特殊的字符。字符 Ogonek 也可以用预定义字符 0x01ED 来定义。

ō

=

o

+

.

+

¯

0x01ED

0x006F

0x0928

0x0904

图 22-1

对于亚洲市场, 只有汉语需要 80 000 多个字符, 但没有这么多的预定义字符。在亚洲语言中, 总是要处理组合字符。其问题在于获取显示字符或文本元素的正确数字, 得到基本字符, 而不是组合字符。System.Globalization 名称空间提供的 StringInfo 类可以用于处理这个问题。

表 22-1 列出了 StringInfo 类的静态方法, 这些方法有助于处理组合字符。

表 22-1

方 法	说 明
GetNextTextElement()	返回指定字符串的第一个文本元素(基本字符和所有的组合字符)
GetTextElementEnumerator()	返回一个允许迭代字符串中的所有文本元素的 TextElementEnumerator 对象
ParseCombiningCharacters()	返回一个引用字符串中的所有基本字符的整型数组



一个显示字符可以包含多个 Unicode 字符。要解决这个问题, 如果编写的应用程序要在国际市场销售, 就不应使用数据类型 char, 而应使用 string。string 可以包含由基本字符和组合字符组成的文本元素, 但 char 不能。

22.1.2 区域性和区域

世界分为多个区域性和区域, 应用程序必须知道这些区域性和区域的差异。区域性是基于用户的语言和区域性习惯的一组偏爱特性。RFC 1766(www.ietf.org/rfc/rfc1766.txt)定义了区域性的名称, 这些名称根据语言和国家或区域的不同在世界各地使用。例如 en-AU、en-CA、en-GB 和 en-US 分别用于表示澳大利亚、加拿大、英国和美国的英语。

在 System.Globalization 名称空间中, 最重要的类是 CultureInfo。这个类表示区域性, 定义了日历、数字和日期的格式, 以及和区域性一起使用的排序字符串。

RegionInfo 类表示区域设置(如货币), 说明该区域是否使用米制系统。在某些区域中, 可以使用多种语言。例如, 西班牙区域就有 Basque(eu-ES)、Catalan(ca-ES)、Spanish(es-ES)和 Galician(gl-ES)

区域性。类似于一个区域可以有多种语言，一种语言也可以在多个区域使用，例如，墨西哥、西班牙、危地马拉、阿根廷和秘鲁等都使用西班牙语。

本章的后面将介绍一个示例应用程序，以说明区域性和区域的这些特征。

1. 特定、中立和不变的区域性

在 .NET Framework 中使用区域性，必须区分 3 种类型：特定、中立和不变的区域性。

特定的区域性与真正存在的区域性相关，这种区域性用上一节介绍的 RFC 1766 定义。特定的区域性可以映射到中立的区域性。例如，de 是特定区域性 de-AT、de-DE、de-CH 等的中立区域性，de 是德语(German)的简写，AT、DE 和 CH 分别是奥地利(Austria)、德国(Germany)和瑞士(Switzerland)等国家的简写。

在翻译应用程序时，通常不需要为每个区域翻译，因为奥地利和瑞士等国使用的德语没有太大的区别。所以可以使用中立的区域性来本地化应用程序，而不需要使用特定的区域性。

不变的区域性独立于真正的区域性。在文件中存储格式化的数字或日期，或通过网络把它们发送到服务器上时，最好使用独立于任何用户设置的区域性。

图 22-2 显示了区域性类型的相互关系。

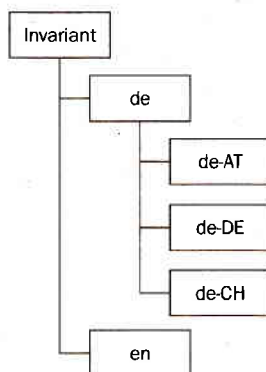


图 22-2

2. CurrentCulture 和 CurrentUICulture

设置区域性时，必须区分用户界面的区域性和数字及日期格式的区域性。区域性与线程相关，并且通过这两种区域性类型，就可以把两种区域性设置应用于线程。Thread 类提供了 CurrentCulture 和 CurrentUICulture 属性。CurrentCulture 属性用于设置与格式化和排序选项一起使用的区域性，而 CurrentUICulture 属性用于设置用户界面的语言。

使用 Windows 控制面板中的“区域和语言”选项，就可以改变 CurrentCulture 的默认设置，如图 22-3 所示。使用这个配置，还可以改变区域性的默认数字、时间和日期格式。

CurrentUICulture 属性不依赖于这个配置，而依赖于操作系统的语言。这有一个例外：如果 Windows 7、Windows Vista 或 Windows XP 安装了多语言用户界面(Multi-language User Interface, MUI)，就可以用区域配置改变用户界面的语言，这会影晌 CurrentUICulture 属性。



图 22-3

这些设置都使用默认值，在许多情况下，不需要改变默认值。如果需要改变区域性，只需把线程的两个区域性改为 Spanish 区域性，如下面的代码段所示：

```
System.Globalization.CultureInfo ci = new
    System.Globalization.CultureInfo("es-ES");
System.Threading.Thread.CurrentThread.CurrentCulture = ci;
System.Threading.Thread.CurrentThread.CurrentUICulture = ci;
```

前面已学习了区域性的设置，下面讨论 `CultureInfo` 设置对数字和日期格式的影响。

3. 数字格式

`System` 名称空间中的数字结构 `Int16`、`Int32` 和 `Int64` 等都有一个重载的 `ToString()` 方法。这个方法可以根据地域创建不同的数字表示法。对于 `Int32` 结构，`ToString()` 方法有下述 4 个重载版本：

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string,
    IFormatProvider);
```

不带参数的 `ToString()` 方法返回一个没有格式化选项的字符串，也可以给 `ToString()` 方法传递一个字符串和一个实现 `IFormatProvider` 接口的类。

该字符串指定表示法的格式。而这个格式可以是标准数字格式化字符串或者图形数字格式化字符串。对于标准数字格式化，字符串是预定义的，其中 C 表示货币符号，D 表示输出为小数，E 表示输出用科学计数法表示，F 表示定点输出，G 表示一般输出，N 表示输出为数字，X 表示输出为十六进制数。对于图形数字格式化字符串，可以指定位数、节和组分隔符、百分号等。图形数字格式字符串####，####表示两个 3 位数块被一个组分隔符分开。

`IFormatProvider` 接口由 `NumberFormatInfo`、`DateTimeFormatInfo` 和 `CultureInfo` 类实现。这个接口定义了 `GetFormat()` 方法，它返回一个格式对象。

`NumberFormatInfo` 类可以为数字定义自定义格式。使用 `NumberFormatInfo` 类的默认构造函数，可以创建独立于区域性的对象或不变的对象。使用这个类的属性，可以改变所有格式选项，如正号、百分号、数字组分隔符和货币符号等。从静态属性 `InvariantInfo` 返回一个与区域性无关的只读 `NumberFormatInfo` 对象。`NumberFormatInfo` 对象的格式化值取决于当前线程的 `CultureInfo` 类，该线程从静态属性 `CurrentInfo` 返回。

下一个示例使用一个简单的控制台项目。在这段代码中，第一个示例显示了在当前线程的区域性格式中所显示的数字(这里是 `English-US`，是操作系统的设置)。第二个示例使用了带有 `IFormatProvider` 参数的 `ToString()` 方法。`CultureInfo` 类实现 `IFormatProvider` 接口，所以创建一个使用法国区域性的 `CultureInfo` 对象。第 3 个示例改变了当前线程的区域性。使用 `Thread` 实例的 `CurrentCulture` 属性，把区域性改为德国区域性：



可从
WTOX.COM
下载源代码

```
using System;
using System.Globalization;
using System.Threading;

namespace NumberAndDateFormatting
{
    class Program
    {
        static void Main(string[] args)
        {
            NumberFormatDemo();
        }

        private static void NumberFormatDemo()
        {
            int val = 1234567890;

            // culture of the current thread
            Console.WriteLine(val.ToString("N"));

            // use IFormatProvider
            Console.WriteLine(val.ToString("N", new CultureInfo("fr-FR")));

            // change the culture of the thread
            Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");
            Console.WriteLine(val.ToString("N"));
        }
    }
}
```

代码段 NumberAndDateFormatting/Program.cs

结果如下所示。可以把这个结果与前面列举的美国、英国、法国和德国区域性的结果进行比较。

```
1,234,567,890.00
1 234 567 890,00
1.234.567.890,00
```

4. 日期格式

对于日期，也提供了与数字相同的支持。`DateTime` 结构有一些把日期转换为字符串的方法。公共实例的 `ToLongDateString()`、`ToLongTimeString()`、`ToShortDateString()` 和 `ToShortTimeString()` 方法

都使用当前区域性来创建字符串表示法。使用 `ToString()` 方法，可以指定另一种区域性：

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

使用 `ToString()` 方法的字符串参数，可以指定预定义格式字符或自定义格式字符串，把日期转换为字符串。`DateTimeFormatInfo` 类指定了可能的值。`DateTimeFormatInfo` 类指定的格式字符串有不同的含义。例如，D 表示长日期格式，d 表示短日期格式，ddd 表示星期的缩写，dddd 表示星期的全称，yyyy 表示年份，T 表示长时间格式，t 表示短时间格式。使用 `IFormatProvider` 参数可以指定区域性。使用不带 `IFormatProvider` 参数的重载方法，表示所使用的是当前线程的区域性：

```
DateTime d = new DateTime(2009, 06, 02);

// current culture
Console.WriteLine(d.ToLongDateString());

// use IFormatProvider
Console.WriteLine(d.ToString("D", new CultureInfo("fr-FR")));

// use culture of thread
CultureInfo ci = Thread.CurrentThread.CurrentCulture;
Console.WriteLine("{0}: {1}", ci.ToString(), d.ToString("D"));
ci = new CultureInfo("es-ES");
Thread.CurrentThread.CurrentCulture = ci;
Console.WriteLine("{0}: {1}", ci.ToString(), d.ToString("D"));
```

这个示例程序的结果说明了使用线程的当前区域性的 `ToLongDateString()` 方法，其中给 `ToString()` 方法传递一个 `CultureInfo` 实例，则显示其法国版本，把线程的 `CultureInfo` 属性改为 `es-ES`，则显示其西班牙版本，如下所示。

```
Tuesday, June 02, 2009
mardi 2 juin 2009
en-US: Tuesday, June 02, 2009
es-ES: martes, 02 de junio de 2009
```

22.1.3 使用区域性

为了全面介绍区域性，下面使用一个 WPF 应用程序示例，该应用程序列出所有的区域性，描述区域性属性的不同特征。图 22-4 显示了该应用程序在 Visual Studio 2010 WPF 设计器中的用户界面。

在应用程序的初始化阶段，所有可用的区域性都添加到应用程序左边的树形视图控件中。这个初始化在 `AddCulturesToTree()` 方法中进行，该方法在 `Window` 类 `CultureDemoForm` 的构造函数中调用：



可从
wrox.com
下载源代码

```
public CultureDemoWindow()
{
    InitializeComponent();
    AddCulturesToTree();
}
```

代码段 CultureDemo/MainWindow.xaml.cs

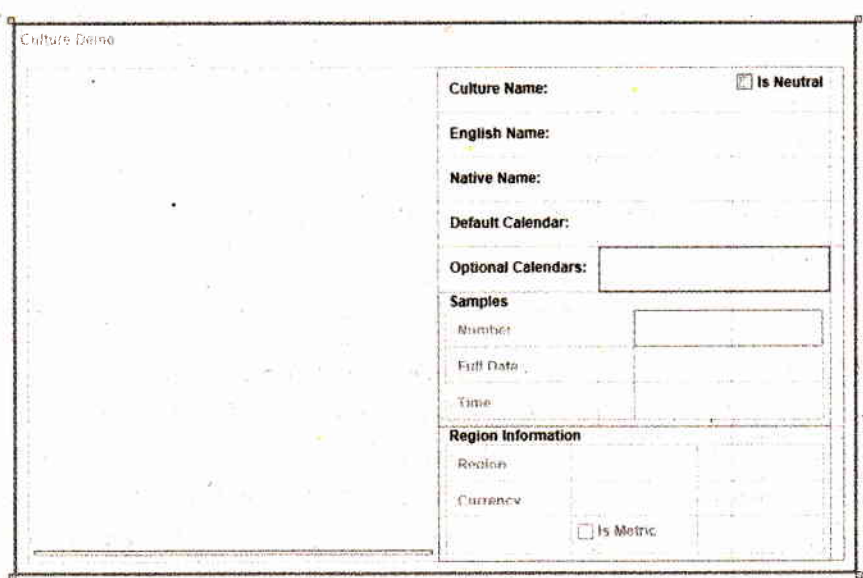


图 22-4

在 `AddCulturesToTree()` 方法中, 从通过静态方法 `CultureInfo.GetCultures()` 中获取所有区域性。给这个方法传递 `CultureTypes.AllCultures`, 就会返回所有可用区域性的未排序数组。该数组用一个 Lambda 表达式排序, 这个 Lambda 表达式要传递给 `Array.Sort()` 方法的第二个参数的 `Comparison` 委托。在 `foreach` 循环中, 把每个区域性添加到树形视图中。为每种区域性创建一个 `TreeViewItem` 对象, 因为 WPF 的 `TreeView` 类使用 `TreeViewItem` 对象来显示。将 `TreeViewItem` 对象的 `Tag` 属性设置为 `CultureInfo` 对象, 以便以后访问这个树型视图中的 `CultureInfo` 对象。

`TreeViewItem` 对象添加到树中的什么地方取决于区域性类型。如果区域性没有父区域性, 它就会添加到树的根节点上。要查找父区域性, 必须把所有区域性保存到一个字典中。相关内容参见前面章节, 其中第 10 章介绍了字典, 第 8 章介绍了 Lambda 表达式。

```
// add all cultures to the tree view
public void AddCulturesToTree()
{
    var culturesByName = new Dictionary<string, TreeViewItem>();

    // get all cultures
    var cultures = CultureInfo.GetCultures(CultureTypes.AllCultures);
    Array.Sort(cultures, (c1, c2) => c1.Name.CompareTo(c2.Name));

    var nodes = new TreeViewItem[cultures.Length];

    int i = 0;
    foreach (var ci in cultures)
    {
        nodes[i] = new TreeViewItem();
        nodes[i].Header = ci.DisplayName;
        nodes[i].Tag = ci;
        culturesByName.Add(ci.Name, nodes[i]);

        TreeViewItem parent;
        if (!String.IsNullOrEmpty(ci.Parent.Name) &&
```

```

        culturesByName.TryGetValue(ci.Parent.Name, out parent))
    {
        parent.Items.Add(nodes[i]);
    }
    else
    {
        treeCultures.Items.Add(nodes[i]);
    }
    i++;
}
}

```

在用户选择树中的一个节点时，就会调用 `TreeView` 类的 `SelectedItemChanged` 事件的处理程序。在这里，这个处理程序在 `TreeCultures_SelectedItemChanged()` 方法中实现。在这个方法中，先调用 `ClearTextFields()` 方法清除所有字段，再选择 `TreeViewItem` 对象的 `Tag` 属性，从树中获取 `CultureInfo` 对象。接着使用 `CultureInfo` 对象的属性 `Name`、`NativeName` 和 `EnglishName` 设置一些文本字段。如果 `CultureInfo` 对象是一个可以使用 `IsNeutralCulture` 属性进行查询的中立区域性，就设置相应的复选框。

```

private void TreeCultures_SelectedItemChanged(object sender,
        RoutedPropertyChangedEventArgs<object>e)
{
    ClearTextFields();

    // get CultureInfo object from tree
    CultureInfo ci = (CultureInfo)((TreeViewItem)e.NewValue).Tag;

    textCultureName.Text = ci.Name;
    textNativeName.Text = ci.NativeName;
    textEnglishName.Text = ci.EnglishName;

    checkIsNeutral.IsChecked = ci.IsNeutralCulture;
}

```

然后获取区域性的日历信息。`CultureInfo` 类的 `Calendar` 属性返回特定区域性的默认 `Calendar` 对象。因为 `Calendar` 类没有对应的名称属性，所以需要使用基类的 `ToString()` 方法获取类的名称，并删除要在文本字段 `textCalendar` 中显示的这个字符串的名称空间。

因为一种区域性可能支持多种日历，所以 `OptionalCalendars` 属性返回额外支持的 `Calendar` 对象数组。这些可选的日历显示在列表框 `listCalendars` 中。派生自 `Calendar` 的 `GregorianCalendar` 类还有一个 `CalendarType` 属性，它列出了 `Gregorian` 日历的类型。这个类型可以是 `GregorianCalendarTypes` 枚举的一个值：`Arabic`、`MiddleEastFrench`、`TransliteratedFrench`、`USEnglish` 或 `Localized`，这取决于区域性。使用 `Gregorian` 日历，类型还可以显示在列表框中。



可从
wrox.com
下载源代码

```

// default calendar
textCalendar.Text = ci.Calendar.ToString().
        Remove(0, 21).Replace("Calendar", "");

// fill optional calendars
listCalendars.Items.Clear();
foreach (Calendar optCal in ci.OptionalCalendars)
{
    StringBuilder calName = new StringBuilder(50);
    calName.Append(optCal.ToString());
    calName.Remove(0, 21);
    calName.Replace("Calendar", "");
}

```

```
// for GregorianCalendar add type information
GregorianCalendar gregCal = optCal as GregorianCalendar;
if (gregCal != null)
{
    calName.AppendFormat(" {0}", gregCal.CalendarType.ToString());
}
listCalendars.Items.Add(calName.ToString());
```

代码段 CultureDemo/MainWindow.xaml.cs

接着, 在 if 语句中使用 “!ci.IsNeutralCulture”, 以检查区域性是否为特定区域性(不是中立区域性)。使用 ShowSamples() 方法显示数字和日期示例。这个方法将在下一段代码中实现。使用 ShowRegionInformation() 方法显示区域的一些信息。对于不变的区域性, 只能显示数字和日期示例, 不能显示区域信息。因为不变的文件与实际的语言无关, 所以它与区域也无关。

```
// display number and date samples
if (!ci.IsNeutralCulture)
{
    groupSamples.IsEnabled = true;
    ShowSamples(ci);

    // invariant culture doesn't have a region
    if (String.Compare(ci.ThreeLetterISOLanguageName, "IVL", true) == 0)
    {
        groupRegion.IsEnabled = false;
    }
    else
    {
        groupRegion.IsEnabled = true;
        ShowRegionInformation(ci.Name);
    }
}
else // neutral culture: no region, no number/date formatting
{
    groupSamples.IsEnabled = false;
    groupRegion.IsEnabled = false;
}
}
```

为了显示一些本地化的数字和日期, 把 CultureInfo 类型的选中对象传递给 ToString() 方法的 IFormatProvider 参数。

```
private void ShowSamples(CultureInfo ci)
{
    double number = 9876543.21;
    textSampleNumber.Text = number.ToString("N", ci);

    DateTime today = DateTime.Today;
    textSampleDate.Text = today.ToString("D", ci);

    DateTime now = DateTime.Now;
    textSampleTime.Text = now.ToString("T", ci);
}
```

为了显示与 RegionInfo 对象相关的信息, 通过在 ShowRegionInformation() 方法中传递选中的区

域性标识符, 构造一个 `RegionInfo` 对象, 然后访问 `DisplayName`、`CurrencySymbol`、`ISOCurrencySymbol` 和 `IsMetric` 属性, 以显示这些信息。

```
private void ShowRegionInformation(string culture)
{
    var ri = new RegionInfo(culture);
    textRegion.Text = ri.DisplayName;
    textCurrency.Text = ri.CurrencySymbol;
    textCurrencyISO.Text = ri.ISOCurrencySymbol;
    checkIsMetric.IsChecked = ri.IsMetric;
}
```

启动应用程序, 在树形视图中就会看到所有的区域性, 选择一个区域性后, 就会列出该区域性的特征, 如图 22-5 所示。

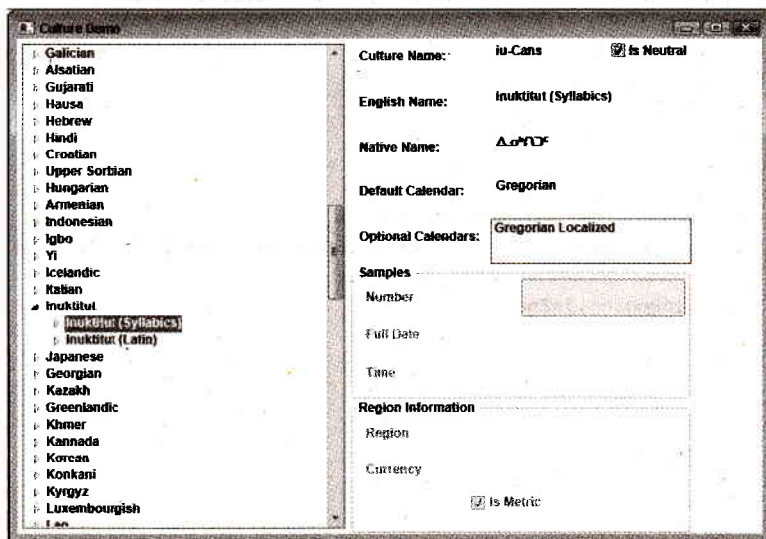


图 22-5

22.1.4 排序

排序字符串取决于区域性。一些区域性有不同的排列顺序。例如在芬兰, 字符 V 和 W 就是相同的。在默认情况下, 为排序而比较字符串的算法要使用与区分区域性的排序方式, 其中排序依赖于区域性。

为了说明芬兰的排序方式, 下面的代码创建一个控制台应用程序小示例, 其中对数组中尚未排序的美国州名进行排序。因为我们将使用 `System.Collections.Generic`、`System.Threading` 和 `System.Globalization` 名称空间中的类, 所以必须声明这些名称空间。下面的 `DisplayName()` 方法用于在控制台上显示数组或集合中的所有元素:



可从
wrox.com
下载源代码

```
static void DisplayNames(string title, IEnumerable<string>e)
{
    Console.WriteLine(title);
    foreach (string s in e)
        Console.WriteLine(s + "-");
}
```

```
Console.WriteLine();
```

代码段 SortingDemo/Program.cs

在 `Main()` 方法中, 在创建了包含一些美国州名的数组后, 就把线程的 `CurrentCulture` 属性设置为 `Finnish` 区域性, 这样, 下面的 `Array.Sort()` 方法就使用芬兰的排列顺序。调用 `DisplayName()` 方法在控制台上显示所有的州名:

```
static void Main()
{
    string[] names = {"Alabama", "Texas", "Washington",
                     "Virginia", "Wisconsin", "Wyoming",
                     "Kentucky", "Missouri", "Utah", "Hawaii",
                     "Kansas", "Louisiana", "Alaska", "Arizona"};

    Thread.CurrentThread.CurrentCulture =
        new CultureInfo("fi-FI");

    Array.Sort(names);
    DisplayNames("Sorted using the Finnish culture", names);
}
```

在以芬兰排列顺序第一次显示美国州名后, 数组将再次排序。如果希望排序独立于用户的区域性, 就可以使用不变的区域性。在已排序的数组要发送到服务器上, 或存储到某个地方时, 就可以采用这种方式。

为此, 给 `Array.Sort()` 方法传递第二个参数。`Sort()` 方法希望第二个参数是实现 `IComparer` 接口的一个对象。`System.Collections` 名称空间中的 `Comparer` 类实现 `IComparer` 接口。`Comparer.DefaultInvariant` 返回一个 `Comparer` 对象, 该对象使用不变的区域性比较数组值, 以进行独立于区域性的排序。

```
// sort using the invariant culture
Array.Sort(names, System.Collections.Comparer.DefaultInvariant);
DisplayNames("Sorted using the invariant culture", names);
}
```

这个程序的输出显示了用 `Finnish` 区域性进行排序的结果和独立于区域性的排序结果。在使用独立于文件的排序方式时, `Virginia` 排在 `Washington` 的前面。用 `Finnish` 区域性进行排序时, `Virginia` 排在 `Washington` 的后面。

```
Sorted using the Finnish culture
Alabama-Alaska-Arizona-Hawaii-Kansas-Kentucky-Louisiana-Missouri-Texas-Utah-
Washington-Virginia-Wisconsin-Wyoming -

Sorted using the invariant culture
Alabama-Alaska-Arizona-Hawaii-Kansas-Kentucky-Louisiana-Missouri-Texas-Utah-
Virginia-Washington-Wisconsin-Wyoming -
```



如果对集合进行的排序应独立于区域性, 该集合就必须用不变的区域性进行排序。在把排序结果发送给服务器或存储在文件中时, 这种方式尤其有效。

除了依赖地域的格式化和测量系统之外, 文本和图片也可能因区域性的不同而不同。此时就需

要使用资源。

22.2 资源

像图片或字符串表这样的资源可以放在资源文件或附属程序集中。在本地化应用程序时，这种资源非常有用，.NET 对本地化资源的搜索提供了内置支持。

在说明如何使用资源本地化应用程序之前，先讨论如何创建和读取资源，而无需考虑语言因素。

22.2.1 创建资源文件

资源文件包含图片、字符串表等条目。要创建资源文件，或者使用一般的文本文件，或者使用那些利用 XML 的.resX 文件。下面从一个简单的文本文件开始。

内嵌字符串表的资源可以使用一般的文本文件来创建。该文本文件只是把字符串赋予键。键是用来从程序中获取数值的名称。键和数值中都可以包含空格。

这个例子显示了 Wrox.ProCSharp.Localization.MyResources.txt 文件中的一个简单字符串表：



可从
wrox.com
下载源代码

```
Title = Professional C#
Chapter = Localization
Author = Christian Nagel
Publisher = Wrox Press
```

代码段 Resources/Wrox.ProCSharp.Localization.MyResources.txt



在保存带 Unicode 字符的文本文件时，必须将文件和相应的编码一起保存。为此，可以在 Save 对话框中选择 Unicode 编码。

22.2.2 资源文件生成器

可以使用资源文件生成器 Resgen.exe(实用程序)在 Wrox.ProCSharp.Localization.MyResources.txt 的外部创建一个资源文件，输入如下代码：

```
resgen Wrox.ProCSharp.Localization.MyResources.txt
```

会创建 Wrox.ProCSharp.Localization.MyResources.resources 文件。最终的资源文件可以作为一个外部文件添加到程序集中，或者内嵌到 DLL 或 EXE 中。Resgen 还可以创建基于 XML 的.resX 资源文件。构建 XML 文件的一种简单方法是使用 ResGen 本身：

```
resgen Wrox.ProCSharp.Localization.MyResources.txt
      Wrox.ProCSharp.Localization.MyResources.resX
```

这条命令创建了 XML 资源文件 Wrox.ProCSharp.Localization.MyResources.resX。22.3 节将讨论如何使用 XML 资源文件。

Resgen 支持强类型化的资源。强类型化的资源用一个访问资源的类表示。这个类可以用 resgen 实用程序的/str 选项创建：

```
resgen /str:C#,Wrox.ProCSharp.Localization,MyResources,MyResources.cs
Wrox.ProCSharp.Localization.MyResources.resX
```

在/str 选项中,按照语言、名称空间、类名和源代码文件名的顺序定义资源。

Resgen 实用程序不支持添加图片。在.NET Framework SDK 示例中,有一个 ResXGen 示例。使用 ResXGen 可以在.resX 文件中引用图片。还可以使用 ResourceWriter 类或 ResXResourceWriter 类以编程方式把图片添加到资源中。

22.2.3 ResourceWriter

除了使用 Resgen 实用程序构建资源文件外,编写程序来创建资源也很简单。ResourceWriter 是来自 System.Resources 名称空间的一个类,它可以用于编写二进制资源文件;ResXResourceWriter 类编写基于 XML 的资源文件。这两个类也支持图片和任何其他可串行化的对象。在使用 ResXResourceWriter 类时,必须引用 System.Windows.Forms 程序集。

下面的代码使用构造函数和文件名 Demo.resx 创建一个 ResXResourceWriter 对象 rw。在创建了一个实例后,使用 ResXResourceWriter 类的 AddResource()方法可以添加至多 2GB 的资源。AddResource()方法的第一个参数指定资源名,第二个参数指定数值。可以使用 Image 类的一个实例来添加图片资源。要使用 Image 类,必须引用 System.Drawing 程序集,还要添加 using 指令,以打开 System.Drawing 名称空间。

下面打开 logo.gif 文件,创建一个 Image 对象。必须把图片复制到可执行文件的目录下,或者在 Image.ToFile()方法的参数中指定图片的完整路径。using 语句指定应在 using 块的尾部自动释放图像资源。把其他简单的字符串资源添加到 ResXResourceWriter 对象中。ResXResourceWriter 类的 Close()方法会自动调用 ResXResourceWriter.Generate()方法,最后把资源写入 Demo.resx 文件中:



```
using System;
using System.Resources;
using System.Drawing;

class Program
{
    static void Main()
    {
        var rw = new ResXResourceWriter("Demo.resx");
        using (Image image = Image.FromFile("logo.gif"))
        {
            rw.AddResource("WroxLogo", image);
            rw.AddResource("Title", "Professional C#");
            rw.AddResource("Chapter", "Localization");
            rw.AddResource("Author", "Christian Nagel");
            rw.AddResource("Publisher", "Wrox Press");
            rw.Close();
        }
    }
}
```

代码段 CreateResource/Program.cs

启动这个小程序,创建嵌入了图像 logo.gif 的资源文件 Demo.resx,这个文件将用于下面的一个 Windows 应用程序。

22.2.4 使用资源文件

使用 C# 命令行编译器 `csc.exe` 和 `/resource` 选项, 或直接使用 Visual Studio, 可以把资源文件添加到程序集中。为了说明如何在 Visual Studio 中使用资源文件, 下面创建一个控制台应用程序 `ResourceDemo`。

在 Solution Explorer 窗口的上下文菜单(Add | Add Existing Item 命令)中, 把前面创建的资源文件 `Demo.resx` 添加到这个项目中。默认情况下, 把这个资源的 Build Action 设置为 `Embedded Resource`, 这样, 这个资源就嵌入到输出的程序集中。

在项目设置(Application | Assembly information 命令)中, 把应用程序的 Neutral Language 设置为主要语言, 如 `English(United States)`, 如图 22-6 所示。改变这个设置, 会在 `assemblyinfo.cs` 文件中添加 `[NeutralResourceLanguageAttribute]` 属性:

```
[assembly: NeutralResourceLanguageAttribute("en-US")]
```

设置这个选项会提高 `ResourceManager` 的性能, 因为它会更快地找到 `en-US` 的资源, 该资源还会用作默认的回退。使用这个特性也可以通过构造函数的第二个参数指定默认资源的位置。使用 `UltimateResourceFallbackLocation` 枚举可以指定默认资源要在主程序集或附属程序集(Main Assembly 和 Satellite 值)中存储。

构建项目后, 使用 `ildasm` 查看生成的程序集时, 会在程序集清单中看到 `mresource` 特性, 如图 22-7 所示。它声明了程序集中资源的名称。如果把 `mresource` 声明为 `public`(与本例一样), 该资源就会从程序集中导出, 且可以用于其他程序集的类中。如果把 `mresource` 声明为 `private`, 则表示该资源不能导出, 只能用于该程序集内部。

要访问嵌入的资源, 可以使用 `System.Resources` 名称空间中的 `ResourceManager` 类。把以嵌入的资源为参数的程序集传递给 `ResourceManager` 类的构造函数。在本例中, 因为把资源嵌入到正在执行的程序集中, 所以应把 `Assembly`.

`GetExecutingAssembly()` 方法的结果作为构造函数的第二个参数。第一个参数是资源的根名。根名由名称空间和资源文件名(不带资源扩展名)组成。如前所述, 使用 `ildasm` 来显示该名称。为此, 只需删除资源的扩展名 `resources` 即可。还可以使用 `System.Reflection.Assembly` 类的 `GetManifestResourceNames()` 方法

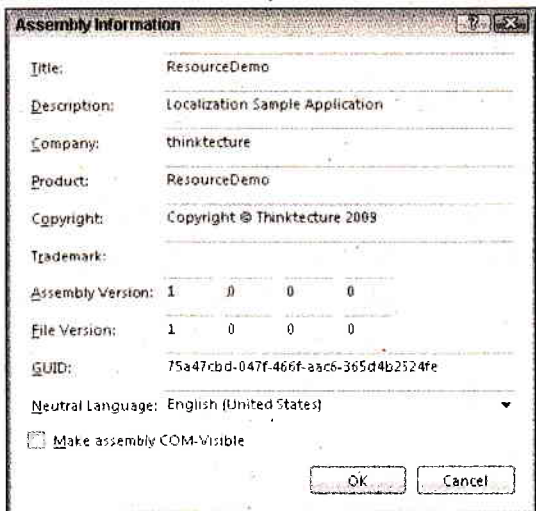


图 22-6

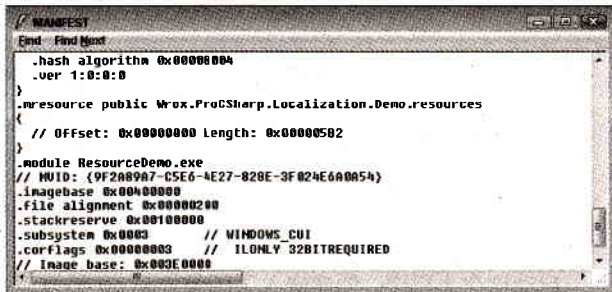


图 22-7

通过编程方式获取该名称。



可从
wrox.com
下载源代码

```
using System;
using System.Drawing;
using System.Reflection;
using System.Resources;

namespace Wrox.ProCSharp.Localization
{
    class Program
    {
        static void Main()
        {
            var rm = new ResourceManager("Wrox.ProCSharp.Localization.Demo",
                                         Assembly.GetExecutingAssembly());
```

代码段 ResourceDemo/Program.cs

使用 `ResourceManager` 实例 `rm`，通过指定 `GetObject()` 和 `GetString()` 方法的键，就可以获得所有的资源：

```
Console.WriteLine(rm.GetString("Title"));
Console.WriteLine(rm.GetString("Chapter"));
Console.WriteLine(rm.GetString("Author"));
using (Image logo = (Image)rm.GetObject("WroxLogo"))
{
    logo.Save("logo.bmp");
}
}
```

通过强类型化的资源，可以简化前面编写的代码；不需要实例化 `ResourceManager`，也不需要使用索引符访问资源，而只需使用属性访问资源名：

```
private static void StronglyTypedResources()
{
    Console.WriteLine(Demo.Title);
    Console.WriteLine(Demo.Chapter);
    Console.WriteLine(Demo.Author);
    using (Bitmap logo = Demo.WroxLogo)
    {
        logo.Save("logo.bmp");
    }
}
```

要使用托管资源编辑器创建强类型化的资源，可以把 `Access Modifier` 从 `No Code Generation` 重置为 `Public` 或 `Internal`。使用 `Public` 选项，生成的类就使用公共访问修饰符，并且它可以在其他程序集中使用。而使用 `Internal` 选项，生成的类就使用内部访问修饰符，并且它只能在程序集内部访问。

设置这个选项后，就会创建 `Demo` 类(它与资源同名)。这个类的静态属性为所有的资源提供了强类型化的资源名。通过实现静态属性，就可以使用 `ResourceManager` 对象，该对象在第一次访问时实例化，并缓存：



可从
wrox.com
下载源代码

```
//-----
// <auto-generated>
//     This code was generated by a tool.
//     Runtime Version:4.0.21006.1
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </auto-generated>
//-----

namespace Wrox.ProCSharp.Localization {
    using System;
    /// <summary>
    ///     A strongly-typed resource class, for looking up localized strings, etc.
    /// </summary>
    /// This class was auto-generated by the StronglyTypedResourceBuilder
    /// class via a tool like ResGen or Visual Studio.
    /// To add or remove a member, edit your .ResX file then rerun ResGen
    /// with the /str option, or rebuild your VS project.
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
        "System.Resources.Tools.StronglyTypedResourceBuilder", "4.0.0.0")]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
    internal class Demo {

        private static global::System.Resources.ResourceManager resourceMan;

        private static global::System.Globalization.CultureInfo resourceCulture;

        [global::System.Diagnostics.CodeAnalysis.SuppressMessageAttribute(
            "Microsoft.Performance", "CA1811:AvoidUncalledPrivateCode")]
        internal Demo() {
        }

        /// <summary>
        ///     Returns the cached ResourceManager instance used by this class.
        /// </summary>
        [global::System.ComponentModel.EditorBrowsableAttribute(
            global::System.ComponentModel.EditorBrowsableState.Advanced)]
        internal static global::System.Resources.ResourceManager ResourceManager {
            get {
                if (object.ReferenceEquals(resourceMan, null)) {
                    global::System.Resources.ResourceManager temp =
                        new global::System.Resources.ResourceManager(
                            "Wrox.ProCSharp.Localization.Demo", typeof(Demo).Assembly);
                    resourceMan = temp;
                }
                return resourceMan;
            }
        }

        /// <summary>
        ///     Overrides the current thread's CurrentUICulture property for all
        ///     resource lookups using this strongly typed resource class.
        /// </summary>
        [global::System.ComponentModel.EditorBrowsableAttribute(
            global::System.ComponentModel.EditorBrowsableState.Advanced)]
    }
```

```

internal static global::System.Globalization.CultureInfo Culture {
    get {
        return resourceCulture;
    }
    set {
        resourceCulture = value;
    }
}

/// <summary>
/// Looks up a localized string similar to Chapter.
/// </summary>
internal static string Chapter {
    get {
        return ResourceManager.GetString("Chapter", resourceCulture);
    }
}

//...

internal static System.Drawing.Bitmap WroxLogo {
    get {
        object obj = ResourceManager.GetObject("WroxLogo", resourceCulture);
        return ((System.Drawing.Bitmap) (obj));
    }
}
}

```

代码段 ResourceDemo/Demo.Designer.cs

22.2.5 System.Resources 名称空间

在举例之前，本节先复习一下 System.Resources 名称空间中处理资源的类：

- **ResourceManager** 类可以用于从程序集或资源文件中获取当前区域性的资源。使用 **ResourceManager** 类还可以获取特定区域性的 **ResourceSet** 类。
- **ResourceSet** 类表示特定区域性的资源。在创建 **ResourceSet** 类的实例时，它会枚举一个实现 **IResourceReader** 接口的类，并在散列表中存储所有的资源。
- **IResourceReader** 接口用于从 **ResourceSet** 中枚举资源。**ResourceReader** 类实现这个接口。
- **ResourceWriter** 类用于创建资源文件。**ResourceWriter** 类实现 **IResourceWriter** 接口。
- **ResXResourceSet**、**ResXResourceReader** 和 **ResXResourceWriter** 类分别类似于 **ResourceSet**、**ResourceReader** 和 **ResourceWriter** 类，但创建的是基于 XML 的资源文件.resX，而不是二进制文件。**ResXFileRef** 可以用于链接资源，而不是把资源嵌入到 XML 文件中。
- **System.Resources.Tools** 名称空间包含的 **StronglyTypedResourceBuilder** 类可以从资源中创建类。

22.3 使用 Visual Studio 的 Windows 窗体本地化

下面创建一个简单的 Windows 窗体应用程序，来说明如何使用 Visual Studio 2010 进行本地化。

这个应用程序没有使用复杂的 Windows 窗体，也没有任何实际的内部功能，因为这里主要是说明本地化功能。在自动生成的源代码中，把名称空间改为 Wrox.ProCSharp. Localization，把类名改为 BookOfTheDayForm。因为名称空间不仅在源文件 BookOfTheDayForm.cs 中作了修改，还在项目设置中进行了修改，所以所有生成的资源文件也都可以获得这个名称空间。为此，需要从 Project | Properties 菜单中选择 Common Properties 命令，为所有新建的条目修改该名称空间。

第 39 章将详细介绍 Windows 窗体应用程序。

为了说明本地化的一些问题，这个程序有一幅图片、一些文本，一个日期和一个数字。图片显示的是一幅已进行了本地化的国旗。图 22-8 在 Windows 窗体设计器中显示了该应用程序的这个窗体。

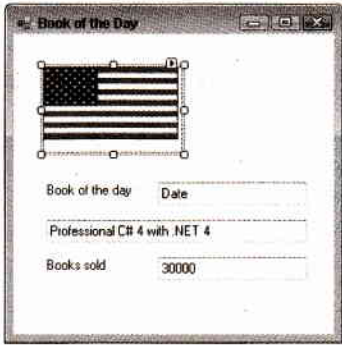


图 22-8

Windows 窗体元素的 Name 和 Text 属性值如表 22-2 所示。

表 22-2

名 称	文 本
labelBookOfTheDay	Book of the day
labelItemsSold	Books sold
textDate	Date
textTitle	Professional C#
textItemsSold	30000
pictureFlag	

除了这个窗体外，还需要一个消息框，根据当前时间显示不同的问候信息。该示例说明了动态创建的对话框在进行本地化时必须采用不同的方式。在 WelcomeMessage() 方法中，使用 MessageBox.Show() 方法显示一个消息框，在窗体类 BookOfTheDayForm 的构造函数中调用 WelcomeMessage() 方法，之后调用 InitializeComponent() 方法。

下面是 WelcomeMessage() 方法的代码：

```

public void WelcomeMessage()
{
    DateTime now = DateTime.Now;
    string message;
    if (now.Hour <= 12)
    {
        message = "Good Morning";
    }
    else if (now.Hour <= 19)
    {
        message = "Good Afternoon";
    }
    else
    {
        message = "Good Evening";
    }
    MessageBox.Show(String.Format("{0}\nThis is a localization sample",
        message));
}

```

窗体中的数字和日期应使用格式化选项来设置。我们添加一个新方法 `SetDateAndNumber()`，用格式选项来设置这些值。在实际应用程序中，这些值应从一个 Web 服务或数据库中得到，但本例把注意力集中在本地化上。日期使用 `D` 选项来格式化(以显示长日期名)。使用图片数字格式字符串 `###,###,###` 来显示该数字，其中 `#` 表示一个数字，`,` 是组分隔符：



可从
wrox.com
下载源代码

```

public void SetDateAndNumber()
{
    DateTime today = DateTime.Today;
    textDate.Text = today.ToString("D");
    int itemsSold = 327444;
    textItemsSold.Text = itemsSold.ToString("###,###,###");
}

```

代码段 `BookOfTheDay/Demo.BookOfTheDayForm.cs`

在 `BookOfTheDayForm` 类的构造函数中，调用了 `WelcomeMessage()` 和 `SetDateAndNumber()` 方法：

```

public BookOfTheDayForm()
{
    WelcomeMessage();

    InitializeComponent();

    SetDateAndNumber();
}

```

Windows 窗体设计器的一个强大功能是可以把窗体的 `Localizable` 属性从 `false` 改为 `true`。这个设置的结果是为对话框创建一个基于 XML 的资源文件，以存储所有资源字符串、属性(包括 Windows 窗体元素的位置和大小)，嵌入的图片等。另外，对 `InitializeComponent()` 方法的实现代码也进行了修改：创建 `System.Resources.ResourceManager` 类的一个实例，为了获取文本字段以及图片的值和位置，应使用 `GetObject()` 方法，而不是直接在代码中写入值。`GetObject()` 方法使用当前线程的 `CurrentUICulture` 属性来查找合适的本地化资源。

下面是从 `BookOfTheDayForm.Designer.cs` 文件中把 `Localizable` 属性设置为 `true` 之前

InitializeComponent()方法的部分代码, 其中设置了 textTitle 的所有属性。

```
private void InitializeComponent()
{
    //...
    this.textTitle = new System.Windows.Forms.TextBox();
    //
    // textTitle
    //
    this.textTitle.Anchor = ((System.Windows.Forms.AnchorStyles)
        (((System.Windows.Forms.AnchorStyles.Top
            | System.Windows.Forms.AnchorStyles.Left)
            | System.Windows.Forms.AnchorStyles.Right)));
    this.textTitle.Location = new System.Drawing.Point(29, 164);
    this.textTitle.Name = "textTitle";
    this.textTitle.Size = new System.Drawing.Size(231, 20);
    this.textTitle.TabIndex = 3;
```

把 Localizable 属性设置为 true 后, InitializeComponent()方法的代码会自动修改, 如下所示:



可从
wrox.com
下载源代码

```
private void InitializeComponent()
{
    System.ComponentModel.ComponentResourceManager resources =
        new System.ComponentModel.ComponentResourceManager(
            typeof(BookOfTheDayForm));
    //...
    this.textTitle = new System.Windows.Forms.TextBox();
    //
    // textTitle
    //
    resources.ApplyResources(this.textTitle, "textTitle");
    this.textTitle.Name = "textTitle";
```

代码段 BookOfTheDay/Demo.BookOfTheDayForm.Designer.cs

资源管理器从哪里获取的数据? 在把 Localizable 属性设置为 true 时, 就生成了一个资源文件 BookOfTheDay.resX。在这个文件中, 首先找到 XML 资源的架构, 接着找到窗体中的所有元素: Type、Text、Location 和 TabIndex 等。

ComponentResourceManager 类派生自 ResourceManager 类, 并提供了 ApplyResources()方法。在这个方法中, 使用第二个参数定义的资源应用于第一个参数中的对象。

下面的 XML 片段说明了 textBoxTitle 的几个属性: Location 属性的值是 “29, 164”, Size 属性的值是 “231, 20”, Text 属性设置为 Professional C# 4 with .NET 4 等。对于每个值, 还存储了值的类型。例如, Location 属性的类型是 System.Drawing.Point, 这个类可在程序集 System.Drawing 中找到。

为什么位置和大小也存储在这个 XML 文件中? 在转换时, 许多字符串都会有完全不同的大小, 且不再适合于原始位置。把位置和大小都存储在资源文件中后, 需要进行本地化的全部内容也都存储在这些文件中, 从而与 C#代码分开:



可从
wrox.com
下载源代码

```
<data name="textTitle.Anchor" type=
    "System.Windows.Forms.AnchorStyles, System.Windows.Forms">
    <value> Top, Left, Right</value>
</data>
```

```

<data name="textTitle.Location" type="System.Drawing.Point, System.Drawing">
  <value> 29, 164 </value>
</data>
<data name="textTitle.Size" type="System.Drawing.Size, System.Drawing">
  <value> 231, 20 </value>
</data>
<data name="textTitle.TabIndex" type="System.Int32, mscorlib">
  <value> 3 </value>
</data>
<data name="textTitle.Text" xml:space="preserve">
  <value> Professional C# 4 with .NET 4 </value>
</data>
<data name="&gt;& gt;textTitle.Name" xml:space="preserve">
  <value> textTitle </value>
</data>
<data name="&gt;& gt;textTitle.Type" xml:space="preserve">
  <value> System.Windows.Forms.TextBox, System.Windows.Forms, Version=4.0.0.0,
    Culture=neutral, PublicKeyToken=b77a5c561934e089 </value>
</data>
<data name="& gt;& gt;textTitle.Parent" xml:space="preserve">
  <value> $this </value>
</data>
  <data name="&gt;& gt;textTitle.ZOrder" xml:space="preserve">
    <value> 2 </value>
  </data>
</data>

```

代码段 BookOfTheDay/BookOfTheDayForm.resx

在修改其中一些资源值时，不需要直接使用 XML 代码来修改。我们可以在 Visual Studio 设计器中直接修改这些资源。无论何时修改窗体的 Language 属性和一些窗体元素的属性，都会为指定的语言生成一个新资源文件。把 Language 属性设置为 German，就会创建德国版本的窗体。把 Language 属性设置为 French，就会创建法国版本的窗体。对于每种语言，都会生成一个属性已被修改的资源文件：对于本例，即 BookOfTheDayForm.de.resX 和 BookOfTheDayForm.fr.resX。

表 22-3 列出了德国版本的窗体需要进行的修改。

表 22-3

德国名称	值
\$this.Text (窗体的标题)	Buch des Tages
labelItemsSold.Text	Bücher verkauft:
labelBookOfTheDay.Text	Buch des Tages

表 22-4 列出了法国版本的窗体应进行的修改。

表 22-4

法国名称	值
\$this.Text (窗体的标题)	Le livre du jour
labelItemsSold.Text	Des livres vendus
labelBookOfTheDay.Text	Le livre du jour

图像不再默认移动到附属程序集中。但在示例应用程序中,国旗应根据国家的不同而不同。为此,必须在 `Resources.resx` 文件中添加美国国旗的图像。这个文件在 Visual Studio 的 Solution Explorer 窗口的 Properties 部分中。使用资源编辑器选择 Images 类别,如图 22-9 所示,以添加 `americanflag.bmp` 文件。为了用图像进行本地化,图像必须在所有的语言中有相同的名称。在这里, `Resources.resx` 文件中的图像名是 `Flag`。可以在属性编辑器中给图像重命名。在属性编辑器中,还可以修改图像是链接还是嵌入。为了提高资源的性能,图像默认为链接。对于链接的图像,图像文件必须与应用程序一起发布。如果要在程序集中嵌入图像,就可以把 `Persistence` 属性改为 `Embedded`。

把 `Resource.resx` 文件复制到 `Resource.de.resx` 和 `Resource.fr.resx` 中,并用 `GermanFlag.bmp` 和 `FranceFlag.bmp` 替代国旗,就可以添加国旗的本地化版本。因为只有中立资源需要强类型化的资源类,所以可以给所有特定语言的资源文件清除 `CustomTool` 属性。

现在编译该项目,为每种语言创建一个附属程序集。在 `debug` 目录里(根据目前的配置,或者是 `release` 目录),创建语言子目录如 `de` 和 `fr`。在这个子目录下,保存了 `BookOfTheDay.resources.dll` 文件。这个文件就是只包含本地化资源的附属程序集。使用 `ildasm` 打开这个程序集,可以看到其已嵌入资源的程序集清单,以及一个定义好的地区。因为这个程序集在程序集属性中有一个地区 `de`,所以它在 `de` 子目录下。其中还有一个带有 `.mresource` 后缀的资源名:它的前缀是名称空间名 `Wrox.ProCSharp.Localization`,其后是类名 `BookOfTheDayForm` 和语言代码 `de`。

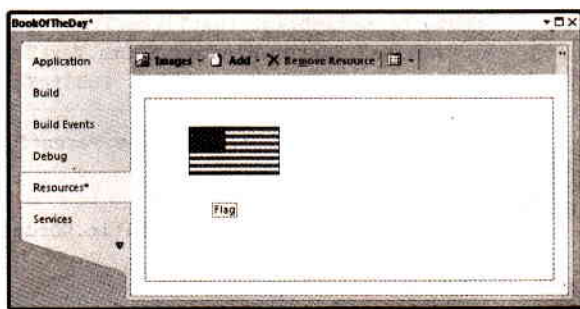


图 22-9

在翻译资源,构建附属程序集后,就可以根据为用户配置的区域性获得正确的译文。此时不翻译欢迎消息,它们需要以另一种方式翻译,稍后讨论。

22.3.1 通过编程方式修改区域性

除了系统配置外,还可以把语言代码当作命令行参数传递给应用程序,以便进行测试。修改 `BookOfTheDayForm` 构造函数,以传递区域性字符串,并根据这个字符串设置区域性。创建一个 `CultureInfo` 实例,把它传递给当前线程的 `CurrentCulture` 和 `CurrentUICulture` 属性。注意 `CurrentCulture` 属性用于格式化,而 `CurrentUICulture` 属性用于加载资源。



可从
wrox.com
下载源代码

```
public BookOfTheDayForm(string culture)
{
    if (!String.IsNullOrEmpty(culture))
    {
        CultureInfo ci = new CultureInfo(culture);
        // set culture for formatting
        Thread.CurrentThread.CurrentCulture = ci;
        // set culture for resources
        Thread.CurrentThread.CurrentUICulture = ci;
    }

    WelcomeMessage();

    InitializeComponent();
}
```

```
SetDateAndNumber();
```

```
}
```

代码段 BookOfTheDay/BookOfTheDayForm.cs

在 Program.cs 文件的 Main() 方法中实例化 BookOfTheDayForm()。在这个方法中, 把区域性字符串传送给 BookOfTheDayForm() 构造函数:



可从
wrox.com
下载源代码

```
[STAThread]
static void Main( string[] args )
{
    string culture = String.Empty;
    if (args.Length == 1)
    {
        culture = args[0];
    }

    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new BookOfTheDayForm(culture));
}
```

代码段 BookOfTheDay/Program.cs

现在可以使用命令行选项启动应用程序。通过正在运行的应用程序, 可以查看格式化选项和从 Windows 窗体设计器中生成的资源。图 22-10 和图 22-11 分别显示了使用 de-DE 和 fr-FR 命令行选项启动的应用程序。

欢迎信息框还有一个问题, 这些字符串在程序中都是硬编码的。因为这些字符串都不是窗体中元素的属性, 所以窗体设计器就不能像处理 Windows 控件的属性那样, 在改变窗体的 Localizable 属性时提取 XML 资源, 而必须自己修改代码。



图 22-10



图 22-11

22.3.2 使用自定义资源文件

对于欢迎消息, 必须翻译硬编码的字符串。英语文本翻译成德语和法语的译文如表 22-5 所示。可以在 Resources.resx 文件中直接编写自定义资源消息和与特定语言相关的派生文本, 当然, 也可以新建资源文件。

表 22-5

名 称	英 语	德 语	法 语
GoodMorning	Good Morning	Guten Morgen	Bonjour
GoodAfternoon	Good Afternoon	Guten Tag	Bonjour
GoodEvening	Good Evening	Guten Abend	Bonsoir
Message1	This is a localization sample.	Das ist ein Beispiel mit Lokalisierung.	C'est un exemple avec la localization.

WelcomeMessage()方法的源代码也必须改为使用资源。在强类型化的资源中，不需要实例化ResourceMessenger 类，而可以使用强类型化资源的属性：



```
public static void WelcomeMessage()
{
    DateTime now = DateTime.Now;
    string message;
    if (now.Hour <= 12)
    {
        message = Properties.Resources.GoodMorning;
    }
    else if (now.Hour <= 19)
    {
        message = Properties.Resources.GoodAfternoon;
    }
    else
    {
        message = Properties.Resources.GoodEvening;
    }
    MessageBox.Show(String.Format("{0}\n{1}", message,
        Properties.Resources.Message1));
}
```

代码段 BookOfTheDay/BookOfTheDayForm.cs

使用英语、德语或法语启动程序时，会得到对应语言的消息框。

22.3.3 资源的自动回退

对于法国和德国版本，我们已经在附属程序集中包含了所有资源。如果不使用这些版本，则所有标签和文本框的值都要修改，这根本没有问题。只需要把要修改的值放在附属程序集中，其他值放在父程序集中即可。例如，对于 de-at (奥地利)，可以把 GoodAfternoon 资源的值改为 Grüß Gott，而其他值不应修改。在运行期间，如果在 de-at 附属程序集中找不到 Good Morning 资源的值，就应搜索父程序集。de-at 附属程序集的父程序集是 de。如果 de 程序集中也没有这个资源，就应在 de 的父程序集(即中立程序集)中搜索该值。中立程序集不包含区域性代码。

在主程序集的区域性代码中，不应定义任何区域性！

22.3.4 外包翻译

使用资源文件很容易完成外包翻译(outsource translation)的任务。在翻译资源文件时,不需要安装 Visual Studio,一个简单的 XML 编辑器就足够了。使用 XML 编辑器的缺点是没有机会重新安排 Windows 窗体元素,如果翻译过来的文本不适合标签或按钮的原始边框,则其大小是不能改变的。使用 Windows 窗体设计器进行翻译是很自然的选择。

Microsoft 的 .NET Framework SDK 附带的一个工具可以满足所有这些需要: Windows 资源本地化编辑器 winres.exe(参见图 22-12)。使用该工具的用户无须访问 C#源文件,只需要翻译二进制文件或基于 XML 的资源文件。在完成这些翻译工作后,就可以把资源文件导入到 Visual Studio 项目中,构建附属程序集了。

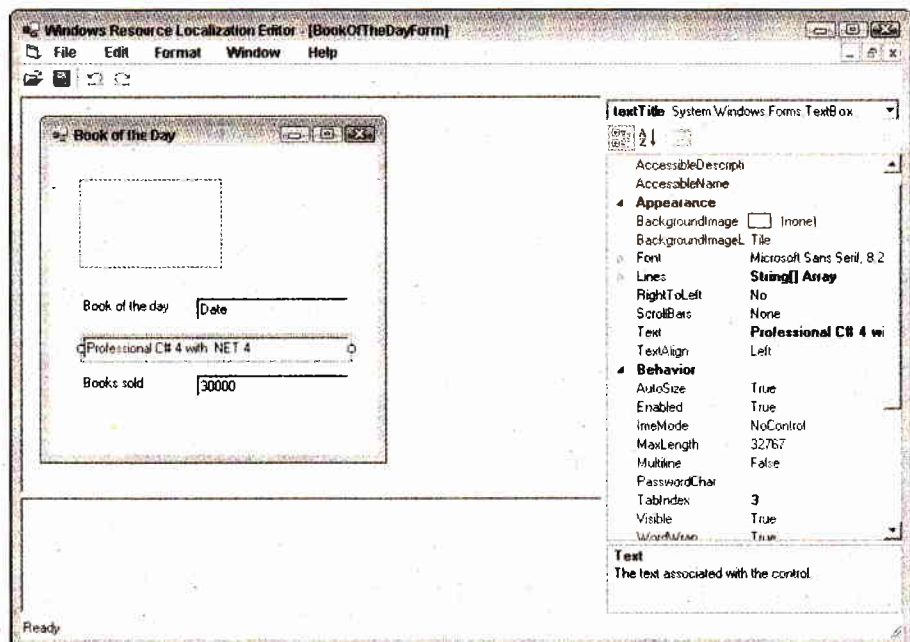


图 22-12

如果不希望翻译局改变标签和按钮的大小和位置,且不能处理 XML 文件,就可以发送一个基于文本的简单文件。使用命令行实用程序 `resgen.exe`,可以从 XML 文件中创建一个文本文件:

```
resgen myresource.resX myresource.txt
```

在收到翻译局完成的翻译文件后,就可以从返回的文本文件中创建一个 XML 文件,注意要给文件名添加区域性名:

```
resgen myresource.es.txt myresource.es.resX
```

22.4 用 ASP.NET 本地化

ASP.NET 应用程序的本地化与 Windows 应用程序的本地化类似。第 40 章将讨论 ASP.NET 应用

程序的功能, 本节只讨论 ASP.NET 应用程序的本地化问题。ASP.NET 4 和 Visual Studio 2010 有许多支持本地化的特性。本地化和全球化的基本概念与前面讨论的一样, 但 ASP.NET 的本地化有一些特殊的问题。

如前所述, 在 ASP.NET 中, 必须区分用户界面区域性和用于格式化的区域性。这两种区域性都可以在 Web 和页面级上定义, 也可以通过编程定义。

要独立于 Web 服务器的操作系统, 区域性和用户界面区域性可以用 web.config 配置文件中的 <globalization> 元素定义:



可从
wrox.com
下载源代码

```
<configuration>
  <system.web>
    <globalization culture="en-US" uiCulture="en-US" />
  </system.web>
</configuration>
```

代码段 WebApplication/Web.config

如果特定 Web 页面的配置不同, 则可以使用 Page 指令指定区域性:

```
<%Page Language="C#" Culture="en-US" UICulture="en-US" %>
```

用户可以用浏览器配置语言。在 Internet Explorer 中, 这个设置用 Language Preference 选项定义, 如图 22-13 所示。

如果页面的语言应根据客户的语言设置而不同, 就可以通过编程把线程的区域性设置从客户接收的语言设置。ASP.NET 的一个自动设置可以完成这个任务。把区域性设置为 Auto, 就可以根据客户的设置指定线程的区域性。

```
<%Page Language="C#" Culture="Auto"
UICulture="Auto" %>
```

在使用资源时, ASP.NET 会区分用于整个 Web 站点的资源和只用于一个页面的资源。

如果资源在一个页面中使用, 就可以在设计视图中选择 Visual Studio 的菜单 Tools | Generate Local Resource 命令, 为页面创建资源。这会创建一个子目录 App_LocalResources, 在这里存储每个页面的资源文件。

这些资源可以用与 Windows 应用程序相同的方式进行本地化。Web 控件和本地资源文件之间的关系用 meta:resourcekey 属性指定, 如下面的 ASP.NET 标签控件。LabelResource1 是资源名, 该名称可以在本地资源文件中修改。



可从
wrox.com
下载源代码

```
<asp:Label ID="Label1" Runat="server" Text="English Text"
meta:resourcekey="Label1Resource1"> </asp:Label>
```

代码段 WebApplication/MultiLanguage.aspx

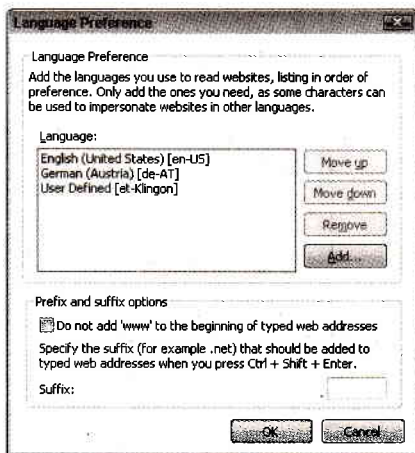


图 22-13

对于应该在多个页面之间共享的资源, 必须创建一个 ASP.NET 文件夹 Appl_GlobalResources。

在这个目录中,可以添加资源文件(如 Messages.resx)及其资源。为了把 Web 控件与这些资源关联起来,可以使用属性编辑器中的 Expressions。单击 Expressions 按钮,打开 Expressions 对话框,如图 22-14 所示。在该对话框中,可以选择表达式类型 Resources,设置 ClassKey 的名称(这是资源文件的名称,这里会生成一个强类型化的资源文件)、ResourceKey 的名称(资源的名称)。

在 ASPX 文件中,用绑定表达式语法<%%\$关联资源:

```
<asp:Label ID="Label2" Runat="server"
    Text=" <%%$ Resources:Messages,
String1 %> ">
</asp:Label>
```

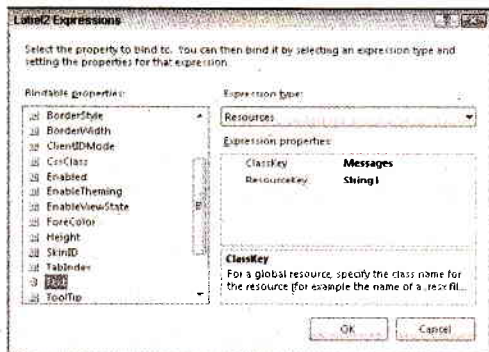


图 22-14

22.5 用 WPF 本地化

Visual Studio 2010 仍没有给 WPF 应用程序的本地化提供强大的支持,但不必等到推出下一个版本,就能本地化 WPF 应用程序。WPF 从一开始就内置了本地化支持,本地化应用程序有几个选项可供选择。可以使用与 Windows 窗体和 ASP.NET 应用程序类似的.NET 资源,也可以使用 XAML(XML for Application Markup Language)资源字典。

下面讨论这些选项。WPF 和 XAML 详见第 35 章和第 36 章。

为了说明如何对 WPF 应用程序使用资源,创建一个简单的 WPF 应用程序,它只包含一个按钮,如图 22-15 所示。

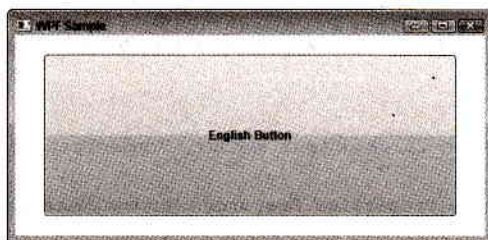


图 22-15

这个应用程序的 XAML 代码如下:



可从
wrox.com
下载源代码

```
<Window x:Class="Wrox.ProCSharp.Localization.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF Sample" Height="240" Width="500">
    <Grid>
        <Button Name="button1" Margin="30,20,30,20" Click="Button_Click"
            Content="English Button" />
    </Grid>
</Window>
```

代码段 WPFApplicationUsingResources/MainWindow.xaml

在按钮的 Click 事件的处理程序代码中，弹出一个包含示例消息的消息框：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("English Message");
}
```

22.5.1 用于 WPF 的 .NET 资源

可以用处理其他应用程序的方式把 .NET 资源添加到 WPF 应用程序中。在 Resources.resx 文件中定义资源 Button1Text 和 Button1Message。这个资源文件默认使用 Internal 访问修饰符来创建 Resources 类。为了在 XAML 中使用它，需要在托管资源编辑器中把它改为 Public。

要使用生成的资源类，需要修改 XAML 代码。添加一个 XML 名称空间别名，以引用 .NET 名称空间 Wrox.ProCSharp.Localization.Properties，如下所示。这里，把别名设置为 props。在 XAML 元素中，这个类的属性可以用于 x:Static 标记扩展。把 Button 的 Content 属性设置为 Resources 类的 Button1Text 属性。



```
<Window x:Class="Wrox.ProCSharp.Localization.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:props="clr-namespace:Wrox.ProCSharp.Localization.Properties"
        Title="WPF Sample" Height="300" Width="300">
    <Grid>
        <Button Name="button1" Margin="30,20,30,20" Click="Button_Click"
                Content="{x:Static props:Resources.Button1Text}"/>
    </Grid>
</Window>
```

代码段 WPFApplicationUsingResources/MainWindow.xaml

要在代码隐藏中使用 .NET 资源，可以直接访问 Button1Message 属性，这与 Windows 窗体应用程序的使用方式相同：

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(Properties.Resources.Button1Message);
}
```

现在资源可以以前面介绍的方式本地化了。

使用 .NET 资源本地化 WPF 应用程序的优点是：

- .NET 资源很容易管理。
- x:Static 绑定由编译器检查。

当然，.NET 资源也有缺点：

- 需要在 XAML 文件中添加 x:Static 绑定，它没有设计器支持。
- 绑定到使用 ResourceManager 生成的资源类。需要执行一些额外的工作，才能支持其他资源管理器，如本章后面讨论的 DatabaseResourceManager。
- 其他 XAML 元素没有类型转换器的支持。

22.5.2 XAML 资源字典

除了使用 .NET 资源本地化 WPF 应用程序之外，还可以直接使用 XAML 创建本地化内容。它也有自己的优缺点。本地化过程的步骤如下：

- (1) 从主要内容中创建一个附属程序集。
- (2) 对可以本地化的内容使用资源字典。
- (3) 给应本地化的元素添加 `x:Uid` 属性。
- (4) 从程序集中提取本地化内容。
- (5) 翻译相应内容。
- (6) 为每种语言创建附属程序集。

下面描述这些步骤。

1. 创建附属程序集

在编译 WPF 应用程序时，XAML 代码编译为二进制格式 BAML，BAML 存储在一个程序集中。要把 BAML 代码从主程序集移动到一个独立的附属程序集中，可以修改 `.csproj` 构建文件，添加如下的 `<UICulture>` 元素，作为 `<propertyGroup>` 元素的一个子元素。这里的区域性是 `en-US`，它定义了项目的默认区域性。用这个构建设置构建项目，会创建一个子目录 `en-US`，并创建一个附属程序集，其中包含了默认语言的 BAML 代码。



可从
wrox.com
下载源代码

```
<UICulture>en-US</UICulture>
```

代码段 WPFApplicationUsingXAMLDictionary/WPFApplicationUsingXAMLDictionary.csproj



修改在 UI 中不可用的项目设置的最简单方法是在 Solution Explorer 窗口中选择该项目，从弹出的上下文菜单中选择 `Unload Project` 命令，以卸载项目，然后从弹出的上下文菜单中选择 `Edit project-file` 命令。修改了项目文件后，就可以再次加载项目。

把 BAML 分离到一个附属程序集中，还应应用 `NeutralResourcesLanguage` 属性，给附属程序集提供资源回退位置。如果决定把 BAML 保存在主程序集中(不给 `.csproj` 文件定义 `<UICulture>`)，`UltimateResourceFallbackLocation` 属性就应设置为 `MainAssembly`。



可从
wrox.com
下载源代码

```
[assembly: NeutralResourcesLanguage("en-US",  
UltimateResourceFallbackLocation.Satellite)]
```

代码段 WPFApplicationUsingXAMLDictionary/AssemblyInfo.cs

2. 添加资源字典

对于需要本地化的代码隐藏内容，可以添加一个资源字典。使用 XAML 可以在 `<ResourceDictionary>` 元素中定义资源，如下所示。在 Visual Studio 中，可以添加一个新的资源字典项，并定义文件名，以新建资源字典。在这里的例子中，资源字典包含一个字符串项。要访问 `System`

名称空间中的 `String` 类型, 需要定义一个 XML 名称空间别名。这里把别名 `system` 设置为程序集 `microsoftlib` 中的 `clr` 名称空间 `System`。所定义的字符串可以用 `message1` 键访问。这个资源字典在 `LocalizedStrings.xaml` 文件中定义。



可从
wrox.com
下载源代码

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:system="clr - namespace:System;assembly=microsoftlib">
    <system:String x:Key="Message1"> English Message </system:String>
</ResourceDictionary>
```

代码段 WPFApplicationUsingXAMLDictionary/LocalizationStrings.xaml

为了使资源字典可用于应用程序, 必须把它添加到资源中。如果资源字典只需在一个窗口或一个特定的 WPF 元素中使用, 就可以把它添加到该窗口或该 WPF 元素的资源集合中。如果多个窗体需要同一个资源字典, 就可以把该资源字典添加到 `<Application>` 元素的 `App.xaml` 文件中, 因此它可用于整个应用程序。这里把资源字典添加到主窗口的资源中:



可从
wrox.com
下载源代码

```
<Window.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="LocalizationStrings.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Window.Resources>
```

代码段 WPFApplicationUsingXAMLDictionary/MainWindow.xaml

要从代码隐藏中使用 XML 资源字典, 可以使用 `Resources` 属性的索引器、`FindResource()` 方法或者 `TryFindResource()` 方法。因为资源是用窗口定义的, 所以使用 `Windows` 类的 `Resources` 属性的索引器访问该资源。`FindResource()` 方法以层次结构的方式来搜索资源。在这个简单的应用程序中, 如果使用 `Button` 的 `FindResource()` 方法, 且通过 `Button` 资源没有找到它, 就可以在 `Grid` 中搜索资源。如果还没有找到资源, 就查找 `Window` 资源, 然后查找 `Application` 资源。



可从
wrox.com
下载源代码

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(this.Resources["Message1"] as string);
    MessageBox.Show(this.FindResource("Message1") as string);
}
```

代码段 WPFApplicationUsingXAMLDictionary/MainWindow.xaml.cs

3. 用于本地化的 Uid 特性

对于自定义资源字典文件, 可以引用应从代码中本地化的文本。为了本地化 XAML 代码和 WPF 元素, `x:Uid` 特性用作需要本地化的元素的唯一标识符。不必把这个特性手动应用于 XAML 内容, 而可以使用 `msbuild` 命令和如下选项:

```
msbuild /t:updateuid
```

在项目文件所在的目录下调用这条命令时, 项目的 XAML 文件会修改, 以给每个元素添加一个 `x:Uid` 属性和一个唯一标识符。如果控件已经有 `Name` 或应用了 `x:Name` 特性, `x:Uid` 就有相同的值; 否则就生成一个新值。前面的相同 XAML 现在就应用了新特性:

```
<Window x:Uid="Window_1"
        x:Class="WPFApplicationUsingXAMLictionaries.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Main Window" Height="240" Width="500">
    <Window.Resources>
        <ResourceDictionary x:Uid="ResourceDictionary_1">
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary x:Uid="ResourceDictionary_2"
                                    Source="LocalizationStrings.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Window.Resources>
    <Grid x:Uid="Grid_1">
        <Button x:Uid="button1" Name="button1" Margin="30,20,30,20" Click="Button_Click"
                Content="English Button" />
    </Grid>
</Window>
```

如果在添加了 `x:Uid` 特性后修改 XAML 文件, 就可以用选项 `/t:checkuid` 验证 `x:Uid` 特性的正确性。

现在可以编译项目, 以创建包含 `x:Uid` 特性的 BAML 代码, 并使用一个工具来提取这个信息。

4. 用于本地化的 LocBaml 工具

编译项目, 会创建一个包含 BAML 代码的附属程序集。在这个附属程序集中, 可以用 `System.Windows.Markup.Localizer` 名称空间中的类提取需要本地化的内容。在 Windows SDK 中包含示例程序 `LocBaml`。这个程序可以用于从 BAML 中提取本地化内容。我们需要把这个可执行的、包含默认内容的附属程序集和 `LocBaml.exe` 复制到一个目录下, 并启动示例程序, 用本地化内容生成一个.csv 文件。

```
LocBaml /parse WPFApplicationUsingXAMLictionaries.resources.dll /out: trans.csv
```



要给通过 .NET 4 构建的 WPF 应用程序使用 `LocBaml`, 还必须用 .NET 4 或较新的版本构建这个工具。如果 `LocBaml` 工具是用 .NET 2.0 构建的旧版本, 它就不能加载 .NET 4 程序集。因为 Windows SDK 包含这个工具的源代码, 所以可以使用 .NET 的最新版本重建它。

可以使用 Microsoft Excel 打开.csv 文件, 并翻译其内容。从.csv 文件中提取的内容包含了按钮的内容和资源字典中的消息, 如下所示:

```
WPFandXAMLResources.g.en-US.resources:localizationstrings.baml,
```

```

system:String_1:System.String.$Content,None,True,True,,English Message
WPFandXAMLResources.g.en-US.resources>window1.baml,
button1:System.Windows.Controls.ContentControl.Content,Button,True,True,,
English Button

```

这个文件包含如下字段:

- BAML 的名称
- 资源的标识符
- 提供内容类型的资源类别
- 资源对于翻译是否可见(可读)的布尔值
- 资源对于翻译是否可修改的布尔值
- 本地化注释
- 资源的值

本地化资源后,就可以为新语言创建一个新目录(如用于德语的 `de` 目录)。目录结构与本章前面的附属程序集相同。使用 `LocBaml` 工具,可以用翻译过来的内容创建附属程序集:

```

LocBaml /generate WPFandXAMLResources.resources.dll /trans:trans_de.csv
/out: ./de /cul:de-DE

```

现在,这里应用了用来设置线程区域性和查找附属程序集相同规则,这以前通过 `Windows` 窗体应用程序来说明。

如前所述,使用 `XAML` 字典进行本地化有许多工作要做。这是一个缺点。幸好不必每天都进行本地化。那么其优点是什么?

- 可以把 `XAML` 文件中的本地化过程延迟到应用程序完成之后。不需要特别的标记或资源映射语法。本地化过程可以与开发过程分开。
- 使用 `XAML` 资源字典在运行时的效率很高。
- 本地化很容易使用 `CSV` 编辑器完成。

缺点是:

- 在 `SDK` 的示例中不支持 `LocBaml` 工具。
- 本地化是一次性过程,很难修改本地化的结果。

22.6 自定义资源读取器

资源读取器是 `.NET Framework 4` 的一部分,利用资源读取器,可以从资源文件和附属程序集中读取资源。如果要把资源放在另一个存储器(如数据库)中,就可以创建一个自定义资源读取器来读取这些资源。

要使用自定义资源读取器,还必须创建自定义资源集和自定义资源管理器。但是,这些都不难,因为可以从已有的类中派生自定义类。

对于该示例应用程序,需要创建一个简单的数据库,其中只有一个表,用于存储消息,该表对于每种支持的语言有一列。表 22-6 列出了这些列和它们相应的值。

```

system:String_1:System.String.$Content,None,True,True,,English Message
WPFandXAMLResources.g.en-US.resources>window1.baml,
button1:System.Windows.Controls.ContentControl.Content,Button,True,True,,
English Button

```

这个文件包含如下字段:

- BAML 的名称
- 资源的标识符
- 提供内容类型的资源类别
- 资源对于翻译是否可见(可读)的布尔值
- 资源对于翻译是否可修改的布尔值
- 本地化注释
- 资源的值

本地化资源后,就可以为新语言创建一个新目录(如用于德语的 de 目录)。目录结构与本章前面的附属程序集相同。使用 LocBaml 工具,可以用翻译过来的内容创建附属程序集:

```

LocBaml /generate WPFandXAMLResources.resources.dll /trans:trans_de.csv
/out: ./de /cul:de-DE

```

现在,这里应用了用来设置线程区域性和查找附属程序集的相同规则,这以前通过 Windows 窗体应用程序来说明。

如前所述,使用 XAML 字典进行本地化有许多工作要做。这是一个缺点。幸好不必每天都进行本地化。那么其优点是什么?

- 可以把 XAML 文件中的本地化过程延迟到应用程序完成之后。不需要特别的标记或资源映射语法。本地化过程可以与开发过程分开。
- 使用 XAML 资源字典在运行时的效率很高。
- 本地化很容易使用 CSV 编辑器完成。

缺点是:

- 在 SDK 的示例中不支持 LocBaml 工具。
- 本地化是一次性过程,很难修改本地化的结果。

22.6 自定义资源读取器

资源读取器是 .NET Framework 4 的一部分,利用资源读取器,可以从资源文件和附属程序集中读取资源。如果要把资源放在另一个存储器(如数据库)中,就可以创建一个自定义资源读取器来读取这些资源。

要使用自定义资源读取器,还必须创建自定义资源集和自定义资源管理器。但是,这些都不难,因为可以从已有的类中派生自定义类。

对于该示例应用程序,需要创建一个简单的数据库,其中只有一个表,用于存储消息,该表对于每种支持的语言有一列。表 22-6 列出了这些列和它们相应的值。

表 22-6

键	默 认	DE	ES	FR	IT
Welcome	Welcome	Willkommen	Recepcion	Bienvenue	Benvenuto
Good Morning	Good Morning	Guten Morgen	Buonas diaz	Bonjour	Buona mattina
Good Evening	Good Evening	Guten Abend	Buonas noches	Bonsoir	Buona sera
Thank you	Thank you	Danke	Gracias	Merci	Grazie
Goodbye	Goodbye	Auf Wiedersehen	Adiós	Au revoir	Arrivederci

对于自定义资源读取器，用 3 个类创建一个组件库，这些类分别是 DatabaseResourceReader、DatabaseResourceSet 和 DatabaseResourceManager。

22.6.1 创建 DatabaseResourceReader 类

DatabaseResourceReader 类定义了两个字段，即访问数据库所需要的连接字符串和读取器应返回的语言。这些字段在这个类的构造函数中填充。language 字段设置为区域性名，这个区域性名将与 CultureInfo 对象一起传递给构造函数。



可从
wrox.com
下载源代码

```
public class DatabaseResourceReader: IResourceReader
{
    private string connectionString;
    private string language;

    public DatabaseResourceReader(string connectionString,
        CultureInfo culture)
    {
        this.connectionString = connectionString;
        this.language = culture.Name;
    }
}
```

代码段 DatabaseResourceReader/DatabaseResourceReader.cs

资源读取器必须实现 IResourceReader 接口，这个接口定义了 Close() 和 GetEnumerator() 方法，GetEnumerator() 方法返回一个 IDictionaryEnumerator，它为资源返回键和值。在 GetEnumerator() 方法的实现代码中，将创建一个散列表，其中存储了特定语言的所有键和值。接着，使用 System.Data.SqlClient 名称空间中的 SqlConnection 类在 SQL Server 中访问数据库。Connection.CreateCommand() 方法创建一个 SqlCommand() 对象，该对象用于指定 SQL 的 SELECT 语句，以访问数据库中的数据。如果把语言设置为 de，Select 语句就是 SELECT [key], [de] FROM Messages。接着，使用 SqlDataReader 对象从数据库中读取所有的值，并把它们放在一个散列表中。最后，返回散列表的枚举器。



有关 ADO.NET 数据访问的更多信息请参阅第 30 章。

```
public System.Collections.IDictionaryEnumerator GetEnumerator()
```

```

    Dictionary<string, string> dict = new Dictionary<string, string>();

    SqlConnection connection = new SqlConnection(connectionString);
    SqlCommand command = connection.CreateCommand();
    if (String.IsNullOrEmpty(language))
        language = "Default";

    command.CommandText = "SELECT [key], [" + language + "] " +
        "FROM Messages";

    try
    {
        connection.Open();

        SqlDataReader reader = command.ExecuteReader();
        while (reader.Read())
        {
            if (reader.GetValue(1) != System.DBNull.Value)
            {
                dict.Add(reader.GetString(0).Trim(), reader.GetString(1));
            }
        }

        reader.Close();
    }
    catch (SqlException ex)
    {
        if (ex.Number != 207) // ignore missing columns in the database
            throw;          // rethrow all other exceptions
    }
    finally
    {
        connection.Close();
    }
    return dict.GetEnumerator();
}

public void Close()
{
}

```

因为 `IResourceReader` 接口派生自 `IEnumerable` 和 `IDisposable` 接口, 所以必须实现 `GetEnumerator()` 方法, 该方法返回 `IEnumerator` 接口, 同时也必须实现和 `Dispose()` 方法。

```

IEnumerator IEnumerable.GetEnumerator()
{
    return this.GetEnumerator();
}

void IDisposable.Dispose()
{
}

```

```

        if (resourceSets.ContainsKey(culture.Name))
        {
            rs = resourceSets[culture.Name];
        }
        else
        {
            rs = new DatabaseResourceSet(connectionString, culture);
            resourceSets.Add(culture.Name, rs);
        }
        return rs;
    }
}

```

22.6.4 DatabaseResourceReader 的客户端应用程序

在客户端应用程序中使用 `ResourceManager` 类的方式与该类在前面的使用方式没有太大的区别。唯一的区别是要使用自定义类 `DatabaseResourceManager` 代替 `ResourceManager` 类。下面的代码段说明了如何使用自己的资源管理器。

通过把数据库连接字符串传递给构造函数,可以新建一个 `DatabaseResourceManager` 对象。接着,像以前一样,调用在基类中实现的 `GetString()` 方法,给它传递键和一个 `CultureInfo` 类型的可选对象,以指定某种区域性。然后,从数据库中获取一个资源值,因为这个资源管理器正在使用 `DatabaseResourceSet` 和 `DatabaseResourceReader` 类。



可从
wrox.com
下载源代码

```

var rm = new DatabaseResourceManager(
    @"server=(local)\sqlexpress;database=LocalizationDemo;" +
    "trusted_connection=true");
string spanishWelcome = rm.GetString("Welcome", new CultureInfo("es-ES"));
string italianThankyou = rm.GetString("ThankYou", new CultureInfo("it"));
string threadDefaultGoodMorning = rm.GetString("GoodMorning");

```

代码段 DatabaseResourceReaderClient/Program.cs

22.7 创建自定义区域性

随着时间的推移, .NET Framework 支持的语言越来越多。但并不是所有语言都可用于 .NET。可以创建自定义区域性。例如,为了给一个区域的少数民族创建自定义区域性,或者给不同的方言创建子区域性,创建自定义区域就很有用。

自定义区域性和区域可以用 `System.Globalization` 名称空间中的 `CultureAndRegionInfoBuilder` 类创建。这个类在 `sysglobl` 程序集中。



因为 `sysglobl` 程序集不在 .NET Framework 4 Client Profile 中,所以必须使用项目设置给 .NET Framework 4 设置 Target Framework, 以引用该程序集。

在 `CultureAndRegionInfoBuilder` 类的构造函数中,可以传递区域性名。该构造函数的第二个参数需要 `CultureAndRegionModifiers` 类型的一个枚举。这个枚举有 3 个值: `Neutral` 表示中立区域性,

如果应替换已有的 Framework 区域性, 就使用 Replacement, 第3个值是 None。

在实例化 CultureAndRegionInfoBuilder 对象后, 就可以设置属性, 来配置区域性。使用这个类的一些属性, 可以定义所有的区域性信息和区域信息, 如名称、日历、数字格式、米制信息等。如果区域性应基于已有的区域性和区域, 就可以使用 LoadDataFromCultureInfo() 和 LoadDataFromRegionInfo() 方法设置实例的属性, 之后通过设置属性来修改不同的值。

调用 Register() 方法, 给操作系统注册新区域性。描述区域性的区域位于 <windows>\Globalization 目录中, 其扩展名是 .nlp。



可从
wrox.com
下载源代码

```
using System;
using System.Globalization;

namespace CustomCultures
{
    class Program
    {
        static void Main()
        {
            try
            {
                // Create a Styria culture
                var styria = new CultureAndRegionInfoBuilder("de-AT-ST",
                                                            CultureAndRegionModifiers.None);

                var cultureParent = new CultureInfo("de-AT");
                styria.LoadDataFromCultureInfo(cultureParent);
                styria.LoadDataFromRegionInfo(new RegionInfo("AT"));
                styria.Parent = cultureParent;
                styria.RegionNativeName = "Steiermark";
                styria.RegionEnglishName = "Styria";
                styria.CultureEnglishName = "Styria (Austria)";
                styria.CultureNativeName = "Steirisch";

                styria.Register();
            }
            catch (UnauthorizedAccessException ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

代码段 CustomCultures/Program.cs

因为在系统上注册自定义语言需要管理员权限, 所以需要使用应用程序清单文件, 来指定需要的执行权限。在项目属性中, 清单文件必须在 Application 设置中指定:



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1"
xmlns:asmv1="urn:schemas-microsoft-com:asm.v1" xmlns:asmv2="urn:schemas-microsoft-com:
asm.v2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
```

```
<requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
  <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
</requestedPrivileges>
</security>
</trustInfo>
</asmv1:assembly>
```

代码段 CustomCultures/app.manifest

现在新建的区域性就可以像其他区域性那样使用了：

```
var ci = new CultureInfo("de-AT-ST");
Thread.CurrentThread.CurrentCulture = ci;
Thread.CurrentThread.CurrentUICulture = ci;
```

区域性可以用于格式化和资源。如果再次启动本章前面编写的 **Cultures In Action** 应用程序，就可以看到自定义区域性。

22.8 小结

本章讨论了.NET 应用程序的全球化和本地化。

在应用程序的全球化环境下，我们讨论了 **System.Globalization** 名称空间，用于格式化依赖于区域性的数字和日期。而且说明了在默认情况下，字符串的排序取决于区域性。我们使用不变的区域性进行独立于区域性的排序。并学习了如何使用 **CultureAndRegionInfoBuilder** 类可以创建自定义文件。

应用程序的本地化使用资源来实现。资源可以放在文件、附属程序集或自定义存储器(如数据库)中。本地化所使用的类在 **System.Resources** 名称空间中。要从其他地方读取资源，如附属程序集或资源文件，可以创建自定义资源读取器。

我们还学习了如何本地化 **Windows** 窗体应用程序、**WPF** 应用程序和 **ASP.NET** 应用程序。除此之外，还了解了不同语言的一些重要词汇。

下一章介绍另一个完全不同的主题——事务。事务处理不仅仅可用于数据库，除了介绍数据库事务之外，下一章还会讨论基于内存的事务资源和事务文件系统。

第 23 章

System.Transactions

本章内容:

- 事务处理阶段和 ACID 属性
- 传统的事务
- 可提交的事务
- 事务的升级
- 依赖的事务
- 环境事务
- 事务的孤立级别
- 自定义资源管理器
- Windows 7 和 Windows Server 2008 的事务

事务的主要特征是,任务要么全部完成,要么都不完成。在写入一些记录时,要么写入所有记录,要么什么都不写入。如果在写入一个记录时即使出现了一次失败,那么在事务中已写入的所有其他数据也会回滚。

事务常用于数据库,但利用 `System.Transactions` 名称空间中的类,还可以对不稳定的、基于内存的对象执行事务处理,如对象列表。对于支持事务的对象列表,如果添加或删除了一个对象时事务处理失败,这个列表的操作会自动撤销。写入一个基于内存的列表与写入数据库一样,也可以在事务中完成。

自从 Windows Vista 之后,文件系统和注册表也支持事务。在注册表中写入一个文件,并做出一些修改的操作可以通过事务来完成。

23.1 概述

什么是事务?考虑一下在 Web 站点上订购图书。图书订购进程会把客户要购买的图书从仓库中取出,并把它放在客户的订购框中,再从客户的信用卡收取购买图书的费用。这两个动作要么都成功完成,要么都不完成。如果从仓库中取出图书时出现错误,就不应从信用卡中收取费用。这个工作可以用事务来完成。

事务最常见的用途是写入或更新数据库中的数据。在消息队列中写入消息,或将数据写入文件或注册表时,也可以使用事务完成。一个事务可以包含多个操作。



System.Messaging 名称空间参见第 46 章。

图 23-1 显示了事务中的主要元素。事务由事务管理器来管理和协调。每个影响事务的结果的资源都由一个资源管理器来管理。事务管理器与资源管理器通信，以定义事务的结果。

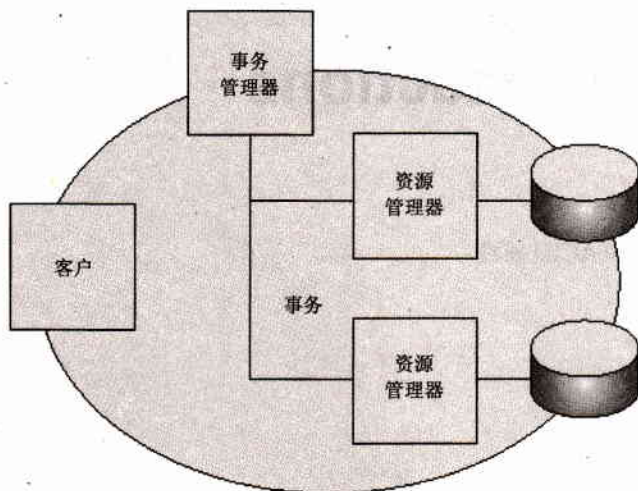


图 23-1

23.1.1 事务处理阶段

事务处理分为激活、准备和提交 3 个阶段。

- **激活阶段：**在这个阶段创建事务。为资源管理事务处理的资源管理器可以用事务进行登记。
- **准备阶段：**在这个阶段，每个资源管理器都可以定义事务的结果。事务的创建者发出结束事务的指令时，就启动这个阶段。事务管理器给所有的资源管理器发出一条“准备”消息。如果资源管理器可以成功生成事务的结果，就向事务管理器发出一条“已准备好”消息。如果资源管理器未能准备好，就可以终止事务处理，发出一条“回滚”消息，强制事务管理器执行回滚操作。在发出“已准备好”消息后，资源管理器必须保证在提交阶段能成功完成工作。为此，稳定的资源管理器必须将准备状态的信息写入一个日志中，这样，如果在准备和提交过程中出现停电等故障时，就可以从该状态继续执行。
- **提交阶段：**当所有的资源管理器都成功准备好了，就开始这个阶段。即所有资源管理器都发出了“已准备好”消息。接着，事务管理器就可以给所有的参与者发送一条“提交”消息，以完成工作。资源管理器现在可以完成事务中的工作，并返回一条“已提交”消息。

23.1.2 ACID 属性

事务有一些特殊的要求，例如，事务的结果必须处于有效的状态。即使服务器断电了，也需要有有效状态。事务的特征可以用术语 ACID 来定义，ACID 是 Atomicity、Consistency、Isolation 和 Durability 的首字母缩写。

- **Atomicity(原子性)**: 表示一个工作单元。在事务中, 要么整个工作单元都成功完成, 要么都不完成。
- **Consistency(一致性)**: 事务开始前的状态和事务完成后的状态必须有效。在执行事务的过程中, 状态可以有临时值。
- **Isolation(隔离性)**: 表示并发进行的事务独立于状态, 而状态在事务处理过程中可能发生变化。在事务未完成时, 事务 A 看不到事务 B 中的临时状态。
- **Durability(持久性)**: 在事务完成后, 它必须以可持久的方式存储起来。如果关闭电源或服务器崩溃, 该状态在重新启动时必须恢复。

并不是每个事务都需要这 4 个属性。例如, 基于内存的事务(如将一项写入列表中)就不需要持久性。事务也不总是需要与外界隔离, 如后面的事务隔离级别所述。

23.2 数据库和实体类

本章的事务使用样本数据库 CourseManagement, 它由图 23-2 所示的结构定义。Courses 表包含课程信息: 课程编号和名称; CourseDates 表包含指定课程的日期, 它链接到表 Courses 上。Students 表包含选修某门课程的人。CourseAttendees 表是 Students 表和 CourseDates 表之间的链接, 它定义了哪些学生选修了什么课程。

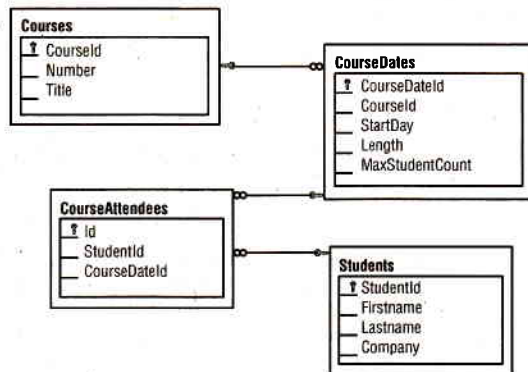


图 23-2



可以从 Wrox 网站或本书附赠光盘上找到本章的数据库和源代码。

在本章的示例应用程序中, 使用了一个包含实体类和数据访问类的库。这个 Student 类包含的属性定义了一个学生, 如 Firstname、Lastname 和 Company:



可从
wrox.com
下载源代码

```

using System;

namespace Wrox.ProCSharp.Transactions
{
    [Serializable]
    public class Student
    {
        public string FirstName { get; set; }
    }
  
```

```

public string LastName { get; set; }
public string Company { get; set; }
public int Id { get; set; }

public override string ToString()
{
    return String.Format("{0} {1}", FirstName, LastName);
}
}
}

```

代码段 DataLib/Student.cs

将学生信息添加到数据库中是在 `StudentData` 类的 `AddStudent()` 方法中完成的。这里创建一个 ADO.NET 连接，来连接 SQL Server 数据库，用 `SqlCommand` 对象定义 SQL 语句，并调用 `ExecuteNonQuery()` 方法来执行该命令：



可从
wrox.com
下载源代码

```

using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    public class StudentData
    {
        public void AddStudent(Student student)
        {
            var connection = new SqlConnection(
                Properties.Settings.Default.CourseManagementConnectionString);
            connection.Open();
            try
            {
                SqlCommand command = connection.CreateCommand();

                command.CommandText = "INSERT INTO Students " +
                    "(FirstName, LastName, Company) VALUES " +
                    "(@FirstName, @LastName, @Company)";
                command.Parameters.AddWithValue("@FirstName", student.FirstName);
                command.Parameters.AddWithValue("@LastName", student.LastName);
                command.Parameters.AddWithValue("@Company", student.Company);

                command.ExecuteNonQuery();
            }
            finally
            {
                connection.Close();
            }
        }
    }
}

```

代码段 DataLib/StudentData.cs



ADO.NET 详见第 30 章。

23.3 传统的事务

在发布 System.Transaction 名称空间之前, 可以直接用 ADO.NET 创建事务, 也可以通过组件、特性和 COM+ 运行库(位于 System.EnterpriseServices 名称空间中)进行事务处理。所以, 本章将比较新的事务模型与传统的事务处理方式, 简要论述如何处理 ADO.NET 事务和如何用 Enterprise Services 进行事务处理。

23.3.1 ADO.NET 事务

首先看看传统的 ADO.NET 事务。如果没有手动创建事务, 每条 SQL 语句就都有一个事务。如果多条语句应参与到同一个事务处理中, 就必须手动创建一个事务。

下面的代码段说明了如何使用 ADO.NET 事务。SqlConnection 类定义了 BeginTransaction() 方法, 它返回一个 SqlTransaction 类型的对象。这个事务对象必须与要参与事务处理的每条命令关联起来。要把命令关联到事务处理上, 可将 SqlCommand 类的 Transaction 属性设置为 SqlTransaction 实例。为了使事务成功完成, 必须调用 SqlTransaction 对象的 Commit() 方法。如果有错误, 就必须调用 Rollback() 方法, 并撤销每个修改。使用 try/catch 有助于检查错误, 并在 catch 块中执行回滚。

```
using System;
using System.Data.SqlClient;
using System.Diagnostics;

namespace Wrox.ProCSharp.Transactions
{
    public class CourseData
    {
        public void AddCourse(Course course)
        {
            var connection = new SqlConnection(
                Properties.Settings.Default.CourseManagementConnectionString);
            SqlCommand courseCommand = connection.CreateCommand();
            courseCommand.CommandText =
                "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
            connection.Open();
            SqlTransaction tx = connection.BeginTransaction();

            try
            {
                courseCommand.Transaction = tx;

                courseCommand.Parameters.AddWithValue("@Number", course.Number);
                courseCommand.Parameters.AddWithValue("@Title", course.Title);
                courseCommand.ExecuteNonQuery();

                tx.Commit();
            }
            catch (Exception ex)
            {
                Trace.WriteLine("Error: " + ex.Message);
                tx.Rollback();
            }
            finally
            {
                connection.Close();
            }
        }
    }
}
```

如果有多条命令要运行在一个事务中，每条命令都必须与该事务关联起来。因为事务还与一个连接关联起来，所以这些命令也必须关联到同一个连接实例上。ADO.NET 事务不支持跨多个连接的事务；它总是关联到一个连接上的本地事务上。

如果使用多个对象创建了一个对象持久性模型，例如，Course 和 CourseDate 类应在一个事务处理中持续使用，就很难使用 ADO.NET 事务来实现。这需把事务传递给参与同一个事务的所有对象。



ADO.NET 事务不是分布式事务。在 ADO.NET 事务中，很难使多个对象参与同一个事务。

23.3.2 System.EnterpriseServices

利用 Enterprise Services，可以免费获得许多服务，其中之一是自动事务处理。通过 System.EnterpriseServices 使用事务的优点是，不需要显式地进行事务处理，运行库会自动创建事务，只需给有事务处理要求的类添加[Transaction]特性即可。[AutoComplete]特性把方法标记为自动设置事务的状态位：如果该方法成功，就设置成功位，因此可以提交事务。如果发生异常，就终止事务。

```
using System;
using System.Data.SqlClient;
using System.EnterpriseServices;
using System.Diagnostics;

namespace Wrox.ProCSharp.Transactions
{
    [Transaction(TransactionOption.Required)]
    public class CourseData: ServicedComponent
    {
        [AutoComplete]
        public void AddCourse(Course course)
        {
            var connection = new SqlConnection(
                Properties.Settings.Default.CourseManagementConnectionString);
            SqlCommand courseCommand = connection.CreateCommand();
            courseCommand.CommandText =
                "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
            connection.Open();
            try
            {
                courseCommand.Parameters.AddWithValue("@Number", course.Number);
                courseCommand.Parameters.AddWithValue("@Title", course.Title);
                courseCommand.ExecuteNonQuery();
            }
            finally
            {
                connection.Close();
            }
        }
    }
}
```

用 `System.EnterpriseServices` 创建事务的一大优点是，多个对象能轻松地运行在同一个事务中，事务还可以自动登记。缺点是它需要 COM+ 主机模型，使用这个技术的类必须派生自基类 `ServiceComponent`。



关于 Enterprise Services 和 COM+ 事务服务的内容见第 51 章，可从 Wrox 网站上下载或直接在随书附赠光盘中找到第 51 章。

23.4 System.Transactions

自 .NET 2.0 以来增加了 `System.Transactions` 名称空间，为 .NET 应用程序带来了一个新的事务编程模型。

这个名称空间提供了几个依赖的 `TransactionXXX` 类。`Transaction` 是所有事务处理类的基类，并定义了所有事务类都可以使用的属性、方法和事件。`CommittableTransaction` 是唯一一个支持提交的事务类。这个类有一个 `Commit()` 方法，所有其他事务类都只能执行回滚。`DependentTransaction` 类用于依赖于其他事务的事务。依赖的事务可以依赖从可提交的事务中创建的事务。不管事务处理是否成功，都把依赖的事务添加到可提交的事务的结果中。`SubordinateTransaction` 类和分布式事务协调器(DTC)一起使用。这个类表示非根事务，但可以由 DTC 管理。

表 23-1 列出了 `Transaction` 类的属性和方法。

表 23-1

Transaction 类的成员	说 明
<code>Current</code>	<code>Current</code> 属性是一个静态属性，不需要有实例。 <code>Transaction.Current</code> 返回一个环境事务处理(如果存在)。环境事务处理在后面讨论
<code>IsolationLevel</code>	<code>IsolationLevel</code> 属性返回一个 <code>IsolationLevel</code> 类型的对象。 <code>IsolationLevel</code> 是一个枚举，它定义了其他事务必须有什么访问权限才能访问事务的临时结果。它会影响到 ACID 中的 “I”；并不是所有的事务处理都是隔离的
<code>TransactionInformation</code>	<code>TransactionInformation</code> 属性返回一个 <code>TransactionInformation</code> 对象。该对象提供了事务的当前状态信息、事务的创建时间和事务标识符
<code>EnlistVolatile()</code> <code>EnlistDurable()</code> <code>EnlistPromotableSinglePhase()</code>	使用登记方法 <code>EnlistVolatile()</code> 、 <code>EnlistDurable()</code> 和 <code>EnlistPromotableSinglePhase()</code> ，可以登记参与事务处理的自定义资源管理器
<code>Rollback()</code>	使用 <code>Rollback()</code> 方法，可以终止一个事务，撤销所有的改变，把所有的结果设置为事务处理之前的状态
<code>DependentClone()</code>	使用 <code>DependentClone()</code> 方法，可以创建一个依赖于当前事务的事务
<code>TransactionCompleted</code>	<code>TransactionCompleted</code> 是一个事件，在事务完成时触发——事务可能成功，也可能失败。在 <code>TransactionCompletedEvent Handler</code> 类型的事件处理程序中，可以访问 <code>Transaction</code> 对象，并读取其状态

为了说明 `System.Transaction` 名称空间的特性, 下面在一个独立程序集中的 `Utilities` 类提供了一些静态方法。`AbortTx()`方法根据用户的输入返回 `true` 或 `false`。`DisplayTransactionInformation()`方法将一个 `TransactionInformation` 对象作为参数, 显示事务中的所有信息: 创建时间、状态、本地和分布式标识符。



可从
wrox.com
下载源代码

```
public static class Utilities
{
    public static bool AbortTx()
    {
        Console.WriteLine("Abort the Transaction (y/n)?");
        return Console.ReadLine() == "y";
    }

    public static void DisplayTransactionInformation(string title,
        TransactionInformation ti)
    {
        if (ti != null)
        {
            Console.WriteLine(title);

            Console.WriteLine("Creation Time: {0:T}", ti.CreationTime);
            Console.WriteLine("Status: {0}", ti.Status);
            Console.WriteLine("Local ID: {0}", ti.LocalIdentifier);
            Console.WriteLine("Distributed ID: {0}", ti.DistributedIdentifier);
            Console.WriteLine();
        }
    }
}
```

代码段 Utilities/Utilities.cs

23.4.1 可提交的事务

`Transaction` 类不能以编程方式提交, 它没有提交事务的方法。基类 `Transaction` 只支持事务处理的终止。唯一支持提交的事务类是 `CommittableTransaction` 类。

在 ADO.NET 中, 事务可以用连接方式登记。为此, 在 `StudentData` 类中添加 `AddStudent()`方法, 该方法将一个 `System.Transactions.Transaction` 对象作为第二个参数。调用 `SqlConnection` 类的 `EnlistTransaction()`方法, 以便使用连接登记 `tx` 对象。这样, ADO.NET 连接就关联到事务上。



可从
wrox.com
下载源代码

```
public void AddStudent(Student student, Transaction tx)
{
    var connection = new SqlConnection(
        Properties.Settings.Default.CourseManagementConnectionString);
    connection.Open();
    try
    {
        if (tx != null)
            connection.EnlistTransaction(tx);
        SqlCommand command = connection.CreateCommand();

        command.CommandText = "INSERT INTO Students (FirstName, " +
            "LastName, Company)" +
            "VALUES (@FirstName, @LastName, @Company)";
    }
}
```

```

        command.Parameters.AddWithValue("@FirstName", student.FirstName);
        command.Parameters.AddWithValue("@LastName", student.LastName);
        command.Parameters.AddWithValue("@Company", student.Company);

        command.ExecuteNonQuery();
    }
    finally
    {
        connection.Close();
    }
}

```

代码段 DataLib/StudentData.cs

在控制台应用程序 `CommittableSamples` 的 `CommittableTransaction()` 方法中, 先创建一个 `CommittableTransaction` 类型的事务。之后, 将信息显示在控制台上。接着, 创建一个 `Student` 对象, 把这个对象从 `AddStudent()` 方法写入数据库中。如果在事务的外部验证数据库中的记录, 就看不到刚才添加的学生, 除非事务已完成。如果事务失败, 就执行回滚, 不把学生写入数据库中。

在调用 `AddStudent()` 方法之后, 就调用帮助方法 `Utilities.AbortTx()`, 确定事务处理是否要终止。如果用户要终止, 就抛出一个 `ApplicationException` 类型的异常, 在 `catch` 块中, 调用 `Transaction` 类的 `Rollback()` 方法, 回滚事务处理。不把记录写入数据库中。如果用户不终止, `Commit()` 方法就提交事务, 并提交事务的最终状态。



可从
wrox.com
下载源代码

```

static void CommittableTransaction()
{
    var tx = new CommittableTransaction();
    Utilities.DisplayTransactionInformation("TX created",
        tx.TransactionInformation);

    try
    {
        var s1 = new Student
        {
            FirstName = "Jörg",
            LastName = "Neumann",
            Company = "thinkecture"
        };
        var db = new StudentData();
        db.AddStudent(s1, tx);

        if (Utilities.AbortTx())
        {
            throw new ApplicationException("transaction abort");
        }

        tx.Commit();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine();
        tx.Rollback();
    }

    Utilities.DisplayTransactionInformation("TX completed",

```

```
tx.TransactionInformation);
```

```
}
```

代码段 TransactionSamples/Program.cs

这里，在应用程序的输出中，事务是激活的，且有一个本地标识符。以下应用程序的输出结果显示了用户选择终止事务。在完成事务后，会看到终止状态。

```
TX created
Creation Time: 7:30:49 PM
Status: Active
Local ID: bdcflcdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Abort the Transaction (y/n)? y
Transaction abort

TX completed
Creation Time: 7:30:49 PM
Status: Aborted
Local ID: bdcflcdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
Distributed ID: 00000000-0000-0000-0000-000000000000
```

从应用程序第2次的输出结果可以看出，用户没有终止事务。事务的状态是已提交，数据写入数据库中。

```
TX Created
Creation Time: 7:33:04 PM
Status: Active
Local ID: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Abort the Transaction (y/n)? n

TX completed
Creation Time: 7:33:04 PM
Status: Committed
Local ID: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Distributed ID: 00000000-0000-0000-0000-000000000000
```

23.4.2 事务处理的升级

`System.Transactions` 支持可升级的事务处理。根据参与事务处理的资源，或者创建本地事务或者创建分布式事务。SQL Server 2005 和 2008 支持可升级的事务，但目前我们只看到本地事务。在前面的例子中，分布式事务 ID 总是设置为 0，且只赋予了本地 ID。对于不支持可升级的事务的资源，会创建分布式事务。如果把多个资源添加到事务中，事务就可以先设置为本地事务，再根据需要升级为分布式事务。当多个 SQL Server 数据库连接添加到事务中时，就会进行这种升级。事务开始时是一个本地事务，之后升级为分布式事务。

现在修改控制台应用程序，使用同一个事务对象 `tx` 添加第二个学生。因为每个 `AddStudent()` 方法都会打开一个新连接，所以在添加第二个学生后，把两个连接关联到该事务上。



可从
wrox.com
下载源代码

```
static void TransactionPromotion()
{
    var tx = new CommittableTransaction();
    Utilities.DisplayTransactionInformation("TX created",
```

```

        tx.TransactionInformation);

    try
    {
        var s1 = new Student
        {
            FirstName = "Jörg",
            LastName = "Neumann",
            Company = "thinktecture"
        };
        var db = new StudentData();
        db.AddStudent(s1, tx);

        var s2 = new Student
        {
            FirstName = "Richard",
            LastName = "Blewett",
            Company = "thinktecture"
        };
        db.AddStudent(s2, tx);

        Utilities.DisplayTransactionInformation("2nd connection enlisted",
            tx.TransactionInformation);

        if (Utilities.AbortTx())
        {
            throw new ApplicationException("transaction abort");
        }

        tx.Commit();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine();
        tx.Rollback();
    }

    Utilities.DisplayTransactionInformation("TX finished",
        tx.TransactionInformation);
}

```

代码段 TransactionSamples/Program.cs

现在运行应用程序，就会看到，对于添加的第一个学生，其分布式标识符是 0，但对于添加的第二个学生，升级了事务，所以分布式标识符与该事务关联起来。

```

TX created
Creation Time: 7:56:24 PM
Status: Active
Local ID: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Distributed ID: 00000000-0000-0000-0000-000000000000

```

```

2nd connection enlisted
Creation Time: 7:56:24 PM
Status: Active
Local ID: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Distributed ID: 501abd91-e512-47f3-95d5-f0488743293d

```

Abort the Transaction (y/n)?

事务的升级需要启动分布式事务协调器(DTC)。如果在系统中升级事务时失败,就应验证 DTC 服务是否启动。启动 Component Services MMC 插件,就可以看到运行在系统上的所有 DTC 事务的实际状态。

在树型视图中选择 Transaction List 节点,就可以看到所有激活的事务。在图 23-3 中,有一个激活的事务,其分布式标识符与前面控制台的输出相同。如果验证系统上的输出,就应确保该事务设置了超时时间,以防超过指定时间后,事务终止。在超过该时间后,在事务列表中就看不到该事务了。还可以用这个工具验证事务的统计信息。Transaction Statistics 显示了提交和终止的事务个数。

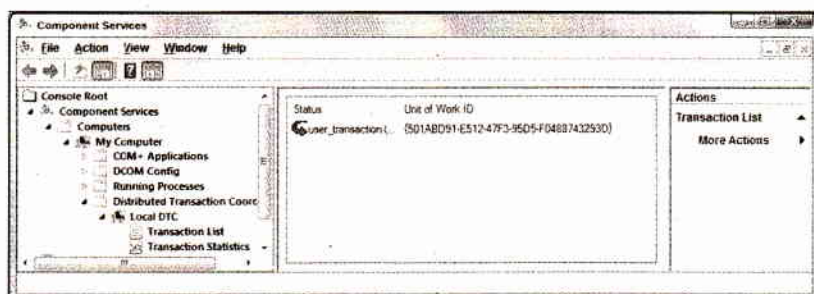


图 23-3

23.4.3 依赖事务

使用依赖事务,可以影响来自多个线程的某个事务。依赖事务会依靠另一个事务,并影响事务的结果。

示例应用程序 `DependentTransactions` 为一个新线程创建了一个依赖的事务。`TxThread()`方法是新线程的一个方法,它将一个 `DependentTransaction` 对象作为参数传递。依赖事务的相关信息用帮助方法 `DisplayTransactionInformation()`来显示。在线程退出之前,调用依赖事务的 `Complete()`方法来定义事务的结果。依赖事务可以调用 `Complete()`或 `Rollback()`方法来定义事务的结果。`Complete()`方法设置成功位。如果根事务结束,且所有依赖事务都把成功位设置为 `true`,就提交事务。如果某个依赖事务调用 `Rollback()`方法来设置终止位,整个事务就会终止。



可从
wrox.com
下载源代码

```
static void TxThread(object obj)
{
    DependentTransaction tx = obj as DependentTransaction;
    Utilities.DisplayTransactionInformation("Dependent Transaction",
        tx.TransactionInformation);

    Thread.Sleep(3000);
    tx.Complete();

    Utilities.DisplayTransactionInformation("Dependent TX Complete",
        tx.TransactionInformation);
}
```

代码段 TransactionSamples/Program.cs

在 `DependentTransaction()`方法中,先实例化 `CommittableTransaction` 类,从创建一个根事务,显示事务的信息。接着, `tx.DependentClone()`方法创建一个依赖的事务。把这个依赖事务传递给 `TxThread()`方法,它定义为新线程的入口点。

`DependentClone()`方法需要一个 `DependentCloneOption` 类型的参数。`DependentCloneOption` 是一个枚举，其值是 `BlockCommitUntilComplete` 和 `RollbackIfNotComplete`。如果根事务在依赖事务之前完成，这个选项就很重要。把该选项设置为 `RollbackIfNotComplete`，如果依赖事务没有在根事务的 `Commit()`方法之前调用 `Complete()`方法，事务就终止。把该选项设置为 `BlockCommitUntilComplete`，`Commit()`方法就等待由所有依赖事务定义的结果。

接着，如果用户没有终止事务，就调用 `CommittableTransaction` 类的 `Commit()`方法。



第 20 章介绍了线程。



可从
wrox.com
下载源代码

```
static void DependentTransaction()
{
    var tx = new CommittableTransaction();
    Utilities.DisplayTransactionInformation("Root TX created",
        tx.TransactionInformation);

    try
    {
        new Thread(TxThread).Start(tx.DependentClone(
            DependentCloneOption.BlockCommitUntilComplete));
        if (Utilities.AbortTx())
        {
            throw new ApplicationException("transaction abort");
        }

        tx.Commit();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        tx.Rollback();
    }

    Utilities.DisplayTransactionInformation("TX finished",
        tx.TransactionInformation);
}
```

代码段 TransactionSamples/Program.cs

在应用程序的输出中，可以看到根事务及其标识符。由于使用了选项 `DependentCloneOption.BlockCommitUntilComplete`，根事务在 `Commit()`方法中等待，直到定义了依赖事务的结果。完成依赖的事务后，就提交事务。

```
Root TX created
Creation Time: 8:35:25 PM
Status: Active
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Abort the Transaction (y/n)? n

Dependent Transaction
Creation Time: 8:35:25 PM
```

```

Status: Active
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Dependent TX Complete
Root TX finished
Creation Time: 8:35:25 PM
Status: Committed
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Creation Time: 8:35:25 PM
Status: Committed
Local ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Distributed ID: 00000000-0000-0000-0000-000000000000

```

23.4.4 环境事务

System.Transactions 的一大优点是环境事务功能。使用环境事务,就不需要手动用连接登记事务;在支持环境事务的资源中,这是自动实现的。

环境事务与当前的线程关联起来。可以使用静态属性 Transaction.Current 获取和设置环境事务。支持环境事务的 API 会检查这个属性,以获得环境事务,用该事务登记。ADO.NET 连接支持环境事务。

可以创建一个 CommittableTransaction 对象,把它赋予 Transaction.Current 属性,以初始化环境事务。创建环境事务的另一种方式是使用 TransactionScope 类。TransactionScope 类的构造函数会创建一个环境事务。

TransactionScope 类的重要方法有 Complete()和 Dispose()方法。Complete()方法可以设置事务的作用域的成功位。Dispose()方法完成该作用域,如果该作用域与根作用域相关,就提交或回滚事务。

因为 TransactionScope 类实现 IDisposable 接口,所以可以用 using 语句定义事务的作用域。默认构造函数创建了一个新的事务,之后创建 TransactionScope 实例,用 Transaction.Current 属性的 get 访问器访问事务,以在控制台上显示事务信息。

要获得事务何时完成的信息,就应为环境事务的 TransactionCompleted 事件设置 OnTransactionCompleted()方法。

然后,调用 StudentData.AddStudent()方法,以新建一个 Student 对象,并把它写入数据库中。对于环境事务,不再需要给这个方法传递 Transaction 对象,因为 SqlConnection 类支持环境事务,并且会自动通过连接登记它。接着 TransactionScope 类的 Complete()方法设置成功位,在 using 语句的最后删除 TransactionScope 类,完成提交。如果没有调用 Complete()方法,Dispose()就终止事务。



如果 ADO.NET 连接不应使用环境事务来登记,就可以用连接字符串设置值 Enlist=false。



可从
wrox.com
下载源代码

```

static void TransactionScope()
{
    using (var scope = new TransactionScope())
    {
        Transaction.Current.TransactionCompleted +=

```

```

        OnTransactionCompleted;

        Utilities.DisplayTransactionInformation("Ambient TX created",
            Transaction.Current.TransactionInformation);

        var s1 = new Student
        {
            FirstName = "Ingo",
            LastName = "Rammer",
            Company = "thinktecture"
        };
        var db = new StudentData();
        db.AddStudent(s1);

        if (!Utilities.AbortTx())
            scope.Complete();
        else
            Console.WriteLine("transaction will be aborted");
    } // scope.Dispose()
}

static void OnTransactionCompleted(object sender, TransactionEventArgs e)
{
    Utilities.DisplayTransactionInformation("TX completed",
        e.Transaction.TransactionInformation);
}

```

代码段 TransactionSamples/Program.cs

运行应用程序，可以看到在创建 `TransactionScope` 类的一个实例后，有一个激活的环境事务。应用程序的最终结果是来自 `TransactionCompleted` 事件处理程序的输出，以显示最终的事务状态。

```

Ambient TX created
Creation Time: 9:55:40 PM
Status: Active
Local ID: a06df6fb-7266-435e-b90e-f024fld6966e:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Abort the Transaction (y/n)? n

TX completed
Creation Time: 9:55:40 PM
Status: Committed
Local ID: a06df6fb-7266-435e-b90e-f024fld6966e:1
Distributed ID: 00000000-0000-0000-0000-000000000000

```

1. 嵌套的作用域和环境事务

使用 `TransactionScope` 类，还可以嵌套作用域。嵌套的作用域可以直接位于原来的作用域中，或位于从作用域中调用的方法里。嵌套的作用域可以使用与外层作用域相同的事务，抑制事务，或创建独立于外层作用域的新事务。作用域的要求通过 `TransactionScopeOption` 枚举定义，把该枚举传递给 `TransactionScope` 类的构造函数。

`TransactionScopeOption` 枚举的值及其作用如表 23-2 所示。

表 23-2

TransactionScopeOption 的 成员	说 明
Required	Required 指定，作用域需要一个事务。如果外层的作用域已经包含了一个环境事务，内层的作用域就使用已有的事务。如果环境事务不存在，就新建一个事务 如果两个作用域共享同一个事务，这两个作用域都会影响事务的结果。只有所有作用域都设置了成功位，事务才能提交。如果在根作用域被删除之前一个作用域没有调用 Complete()方法，事务就终止
RequiresNew	RequiresNew 总是创建一个新的事务。如果外层的作用域已定义了一个事务，内层作用域中的事务就是完全独立的。两个事务都可以独立地提交或终止
Suppress	使用 Suppress，无论外层的作用域是否包含一个事务，作用域都不会包含环境事务

下面的例子定义了两个作用域，其中使用 TransactionScopeOption.RequiresNew 选项，把内层作用域配置为需要一个新的事务：



可从
wrox.com
下载源代码

```
using (var scope = new TransactionScope())
{
    Transaction.Current.TransactionCompleted +=
        OnTransactionCompleted;

    Utilities.DisplayTransactionInformation("Ambient TX created",
        Transaction.Current.TransactionInformation);

    using (var scope2 =
        new TransactionScope(TransactionScopeOption.RequiresNew))
    {
        Transaction.Current.TransactionCompleted +=
            OnTransactionCompleted;

        Utilities.DisplayTransactionInformation(
            "Inner Transaction Scope",
            Transaction.Current.TransactionInformation);

        scope2.Complete();
    }
    scope.Complete();
}
```

代码段 TransactionSamples/Program.cs

运行应用程序，尽管使用相同的线程，但会看到两个作用域有不同的事务标识符。让一个线程因为作用域不同而有不同的环境事务，事务标识符在 GUID 后面的最后一位数字上不同。

 GUID 是包含 128 位唯一值的全局唯一标识符。

Ambient TX created
Creation Time: 11:01:09 PM
Status: Active
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:1
Distributed ID: 00000000-0000-0000-0000-0000000000

```

Inner Transaction Scope
Creation Time: 11:01:09 PM
Status: Active
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Distributed ID: 00000000-0000-0000-0000-000000000000

```

```

TX completed
Creation Time: 11:01:09 PM
Status: Committed
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Distributed ID: 00000000-0000-0000-0000-000000000000

```

```

TX completed
Creation Time: 11:01:09 PM
Status: Committed
Local ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:1
Distributed ID: 00000000-0000-0000-0000-000000000000

```

如果把内层作用域的设置改为 `TransactionScopeOption.Required`，就会发现两个作用域使用同一个事务，且都影响事务的结果。

2. 多线程和环境事务

如果多个线程使用同一个环境事务，就需要多做一些工作。因为一个环境事务绑定到一个线程上，所以如果新建了一个线程，它就不会有起始线程中的环境事务。

这个行为在下一个例子中说明。在 `Main()` 方法中，创建了一个 `TransactionScope`。在这个事务的作用域中，启动一个新线程。新线程的主要方法 `ThreadMethod()` 新建了一个事务的作用域。在创建该作用域的过程中，没有传递任何参数，因此使用默认选项 `TransactionScopeOption.Required`。如果存在一个环境事务，就使用已有的事务。如果没有环境事务，就新建一个事务。



可从
wrox.com
下载源代码

```

using System;
using System.Threading;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    class Program
    {
        static void Main()
        {
            try
            {
                using (var scope = new TransactionScope())
                {
                    Transaction.Current.TransactionCompleted +=
                        TransactionCompleted;

                    Utilities.DisplayTransactionInformation("Main thread TX",
                        Transaction.Current.TransactionInformation);

                    new Thread(ThreadMethod).Start();

                    scope.Complete();
                }
            }
        }
    }
}

```

```

        catch (TransactionAbortedException ex)
        {
            Console.WriteLine("Main-Transaction was aborted, {0}",
                               ex.Message);
        }
    }

    static void TransactionCompleted(object sender, TransactionEventArgs e)
    {
        Utilities.DisplayTransactionInformation("TX completed",
                                                e.Transaction.TransactionInformation);
    }

    static void ThreadMethod(object dependentTx)
    {
        try
        {
            using (var scope = new TransactionScope())
            {
                Transaction.Current.TransactionCompleted +=
                    TransactionCompleted;
                Utilities.DisplayTransactionInformation("Thread TX",
                                                        Transaction.Current.TransactionInformation);
                scope.Complete();
            }
        }
        catch (TransactionAbortedException ex)
        {
            Console.WriteLine("ThreadMethod-Transaction was aborted, {0}",
                               ex.Message);
        }
    }
}

```

代码段 MultithreadingAmbientTx/Program.cs

启动应用程序，就会看到两个线程中的事务是完全独立的。新线程中的事务有一个不同的事务ID。事务ID的区别是GUID后面的最后一个数字不同，这与嵌套的作用域相同(当嵌套的作用域也需要一个新的事务时)。

```

Main thread TX
Creation Time: 21:41:25
Status: Active
Local ID: f1e736ae-84ab-4540-b71e-3de272ffc476:1
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 21:41:25
Status: Committed
Local ID: f1e736ae-84ab-4540-b71e-3de272ffc476:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Thread TX
Creation Time: 21:41:25
Status: Active

```

```

Local ID: f1e736ae-84ab-4540-b71e-3de272ffc476:2
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 21:41:25
Status: Committed
Local ID: f1e736ae-84ab-4540-b71e-3de272ffc476:2
Distributed ID: 00000000-0000-0000-0000-000000000000

```

要在另一个线程中使用同一个环境事务，需要使用依赖事务。现在修改示例，给新线程传递一个依赖事务。调用环境事务上的 `DependentClone()` 方法，以从环境事务中创建依赖事务。在这个方法中，设置了 `DependentCloneOption.BlockCommitUntilComplete`，以便主调线程会等到新线程完成后，才提交事务。

```

class Program
{
    static void Main()
    {
        try
        {
            using (var scope = new TransactionScope())
            {
                Transaction.Current.TransactionCompleted +=
                    TransactionCompleted;

                Utilities.DisplayTransactionInformation("Main thread TX",
                    Transaction.Current.TransactionInformation);

                new Thread(ThreadMethod).Start(
                    Transaction.Current.DependentClone(
                        DependentCloneOption.BlockCommitUntilComplete));

                scope.Complete();
            }
        }
        catch (TransactionAbortedException ex)
        {
            Console.WriteLine("Main-Transaction was aborted, {0}",
                ex.Message);
        }
    }
}

```

在线程的方法中，使用 `Transaction.Current` 属性的 `set` 访问器把所传递的依赖事务赋予环境事务。现在事务的作用域通过依赖事务来使用同一个事务。使用完依赖事务时，需要调用 `DependentTransaction` 对象的 `Complete()` 方法。

```

Static void ThreadMethod(object dependentTx)
{
    DependentTransaction dTx = dependentTx as DependentTransaction;

    try
    {
        Transaction.Current = dTx;

        using (var scope = new TransactionScope())
        {
            Transaction.Current.TransactionCompleted +=

```

```

        TransactionCompleted;

        Utilities.DisplayTransactionInformation("Thread TX",
            Transaction.Current.TransactionInformation);
        scope.Complete();
    }
}
catch (TransactionAbortedException ex)
{
    Console.WriteLine("ThreadMethod-Transaction was aborted, {0}",
        ex.Message);
}
finally
{
    if (dTx != null)
    {
        dTx.Complete();
    }
}
}

static void TransactionCompleted(object sender, TransactionEventArgs e)
{
    Utilities.DisplayTransactionInformation("TX completed",
        e.Transaction.TransactionInformation);
}
}

```

现在运行应用程序，主线程和新建的线程都正在使用同一个事务，并正在影响该事务。线程列出的事务有相同的标识符。如果一个线程没有调用 `Complete()` 方法设置成功位，就终止整个事务。

```

Main thread TX
Creation Time: 23:00:57
Status: Active
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

Thread TX
Creation Time: 23:00:57
Status: Active
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 23:00:57
Status: Committed
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

TX completed
Creation Time: 23:00:57
Status: Committed
Local ID: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Distributed ID: 00000000-0000-0000-0000-000000000000

```

23.5 隔离级别

本章的开头介绍了用于描述事务的 ACID 属性。ACID 中的字母 I(Isolation, 隔离)并不总是完全需要。出于性能原因,可以降低隔离要求,但必须了解改变隔离级别带来的问题。

如果不完全隔离事务外部的作用域,就可能出问题,这些问题有 3 类。

- **脏读**——在脏读操作中,另一个事务可以读取在一个事务中改变的记录。因为在一个事务中改变的记录可能回滚到最初的状态,所以从另一个事务中读取这个临时状态就称为“脏读”——数据并没有提交。通过锁定要改变的记录,就可以避免这个问题。
- **不可重复读**——当数据在事务中读取,而该事务运行的同时,另一个事务修改了相同的记录,此时,就会出现不可重复读操作。如果该记录在事务中读取多次,结果就会不同——不可重复。锁定读取的记录,即可避免这个问题。
- **幻读**——当读取一个范围内的数据,例如,使用 WHERE 子句读取时,就会出现幻读问题。在一个事务中读取这些记录时,另一个事务可以添加一个属于该范围的新记录。用相同的 WHERE 子句再次读取这些记录,会返回数量不同的记录。在更新一个范围的记录时,幻读是一个特殊的问题。例如,UPDATE Addresses SET Zip=4711 WHERE (Zip=2315)会把所有记录的邮政编码从 2315 更新为 4711。如果在更新过程中,另一个用户添加了一个邮政编码为 2315 的新记录,那么完成更新后,数据库将仍包含邮政编码为 2315 的记录。这个问题可以通过范围锁定来避免。

在定义隔离要求时,可以设置隔离级别。隔离级别用 IsolationLevel 枚举定义,在创建事务时,会配置该枚举(或者使用 CommittableTransaction 类的构造函数或者使用 TransactionScope 类的构造函数)。IsolationLevel 枚举定义了锁定操作。表 23-3 列出了 IsolationLevel 枚举的值。

表 23-3

隔离级别	说 明
ReadUncommitted	使用 ReadUncommitted, 事务不会相互隔离。使用这个级别,不等待其他事务释放锁定的记录。这样,就可以从其他事务中读取未提交的数据——脏读。这个级别通常仅用于读取不管是否读取临时修改都无关紧要的记录,如报表
ReadCommitted	ReadCommitted 等待其他事务释放对记录的写入锁定。这样,就不会出现脏读操作。这个级别为读取当前的记录设置读取锁定,为要写入的记录设置写入锁定,直到事务完成为止。对于要读取的一系列记录,在移动到下一个记录上时,前一个记录都是未锁定的,所以可能出现不可重复的读操作
RepeatableRead	RepeatableRead 为读取的记录设置锁定,直到事务完成为止。这样,就避免了不可重复读的问题。但幻读仍可能发生
Serializable	Serializable 设置范围锁定。在运行事务时,不可能添加与所读取的数据位于同一个范围的新记录
Snapshot	Snapshot 只能用于 SQL Server 2005 及其以后的版本。在复制修改的记录时,这个级别会减少锁定。这样,其他事务仍可以读取旧数据,而无需等待解锁
Unspecified	Unspecified 表示,提供程序使用另一个隔离级别值,该值不同于 IsolationLevel 枚举定义的值
Chaos	Chaos 类似于 ReadUncommitted, 但除了执行 ReadUncommitted 值的操作之外,它不能锁定更新的记录

表 23-4 总结了设置最常用的事务隔离级别可能导致的问题。

表 23-4			
隔 离 级 别	脏 读	不可重复读	幻 读
Read Uncommitted	Y	Y	Y
Read Committed	N	Y	Y
Repeatable Read	N	N	Y
Serializable	N	N	N

下面的代码段说明了如何使用 TransactionScope 类设置隔离级别。在 TransactionScope 类的构造函数中,可以设置前面讨论的 TransactionScopeOption 枚举和 TransactionOptions 类.TransactionOptions 类允许定义 IsolationLevel 和 Timeout 属性。

```
var options = new TransactionOptions();
options.IsolationLevel = IsolationLevel.ReadUncommitted;
options.Timeout = TimeSpan.FromSeconds(90);
using (TransactionScope scope =
    new TransactionScope(TransactionScopeOption.Required,
        options))
{
    // Read data without waiting for locks from other transactions,
    // dirty reads are possible.
}
```

23.6 自定义资源管理器

新事务模型的一个最大的优点是,很容易创建参与事务处理的自定义资源管理器。资源管理器不仅管理稳定的资源,还管理不稳定或内存中的资源,例如,简单的 int 数和泛型列表。

图 23-4 显示了资源管理器和事务类之间的关系。资源管理器实现了 IEnlistmentNotification 接口,该接口定义了 Prepare()、InDoubt()、Commit()和 Rollback()方法。资源管理器实现这个接口,是为了管理资源的事务。作为事务的一部分,资源管理器必须用 Transaction 类登记。不稳定的资源管理器调用 EnlistVolatile()方法,稳定的资源管理器调用 EnlistDurable()方法。根据事务的结果,事务管理器通过资源管理器从 IEnlistmentNotification 接口中调用不同的方法。

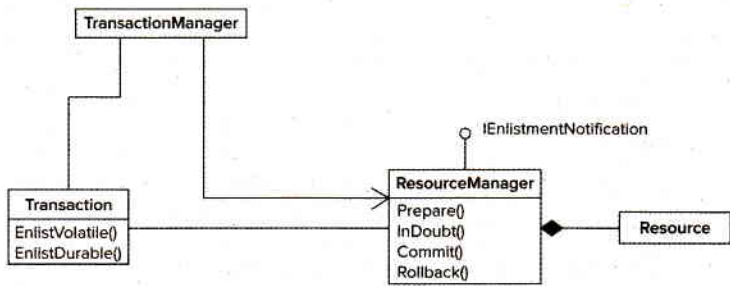


图 23-4

表 23-5 介绍了 `IEnlistmentNotification` 接口中必须通过资源管理器实现的方法。23.1.1 节解释了激活、准备和提交阶段。

表 23-5

<code>IEnlistmentNotification</code> 接口的成员	说 明
<code>Prepare()</code>	事务管理器调用 <code>Prepare()</code> 方法准备事务。资源管理器调用 <code>PreparingEnlistment</code> 参数(它传递给 <code>Prepare()</code> 方法)的 <code>Prepared()</code> 方法, 来完成准备阶段。如果工作没有成功完成, 资源管理器就调用 <code>ForceRollback()</code> 方法, 通知事务管理器。稳定的资源管理器必须编写一个日志, 以便它在准备阶段之后成功完成事务
<code>Commit()</code>	当所有资源管理器都成功准备好事务时, 事务管理器就调用 <code>Commit()</code> 方法。资源管理器现在可以完成工作, 使之可在事务的外部可见, 并调用 <code>Enlistment</code> 参数的 <code>Done()</code> 方法
<code>Rollback()</code>	如果一个资源管理器没有成功地准备好事务, 事务管理器就调用所有资源管理器的 <code>Rollback()</code> 方法。在状态返回为该事务之前的状态后, 资源管理器就调用 <code>Enlistment</code> 参数的 <code>Done()</code> 方法
<code>InDoubt()</code>	如果事务管理器调用 <code>Commit()</code> 方法后出现了一个问题(资源管理器没有用 <code>Done()</code> 方法返回完成信息), 事务管理器就调用 <code>InDoubt()</code> 方法

事务资源

事务资源必须保存实时值和临时值。实时值从事务的外部读取, 并定义了事务回滚时的有效状态。临时值定义了事务提交时事务的有效状态。

为了使非事务类型变成事务类型, 泛型类 `Transactional<T>` 封装了一个非泛型类型, 从而使它的用法如下:

```
Transactional<int>txInt = new Transactional<int>();
Transactional<string>txString = new Transactional<string>();
```

下面看看 `Transactional<T>` 类的实现代码。托管资源的实时值包含在变量 `liveValue` 中, 与事务相关的临时值存储在 `ResourceManager<T>` 中。变量 `enlistedTransaction` 与环境事务(假定存在环境事务)关联起来。



可从
wrox.com
下载源代码

```
using System.Diagnostics;
Insert IconMargin [FILENAME]
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    public partial class Transactional<T>
    {
        private T liveValue;
        private ResourceManager<T> enlistment;
        private Transaction enlistedTransaction;
```

代码段 CustomResource/Transactional.cs

在 `Transactional` 类的构造函数中，实时值设置为变量 `liveValue`。如果在环境事务中调用该构造函数，就调用帮助方法 `GetEnlistment()`。`GetEnlistment()`方法先检查是否有一个环境事务，并断言是否没有一个环境事务。如果没有登记事务，就实例化 `ResourceManager<T>` 辅助类，并调用 `EnlistVolatile()`方法，用该事务登记资源管理器。另外，把变量 `enlistedTransaction` 设置为该环境事务。

如果环境事务不同于已登记的事务，就抛出一个异常。该实现代码不支持在两个不同的事务中修改相同的值。如果有这个要求，就可以创建一把锁，等待一个事务释放该锁，之后在另一个事务中修改它。

```
public Transactional(T value)
{
    if (Transaction.Current == null)
    {
        this.liveValue = value;
    }
    else
    {
        this.liveValue = default(T);
        GetEnlistment().Value = value;
    }
}

public Transactional()
: this(default(T)) {}

private ResourceManager<T>GetEnlistment()
{
    Transaction tx = Transaction.Current;
    Trace.Assert(tx != null, "Must be invoked with ambient transaction");

    if (enlistedTransaction == null)
    {
        enlistment = new ResourceManager<T>(this, tx);
        tx.EnlistVolatile(enlistment, EnlistmentOptions.None);
        enlistedTransaction = tx;
        return enlistment;
    }
    else if (enlistedTransaction == Transaction.Current)
    {
        return enlistment;
    }
    else
    {
        throw new TransactionException(
            "This class only supports enlisting with one transaction");
    }
}
```

`Value` 属性返回所包含的类的值，并设置该值。但是，通过事务，不能只设置和返回 `liveValue` 变量。只有对象在事务的外部，才能设置和返回它。为了使代码可读性更强，`Value` 属性在其实现代码中使用 `GetValue()`和 `SetValue()`方法。

```
public T Value
{

```



可从
wrox.com
下载源代码

```
using System;
using System.Diagnostics;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    public partial class Transactional<T>
    {
        internal class ResourceManager<T1>: IEnlistmentNotification
        {
            private Transactional<T1>parent;
            private Transaction currentTransaction;

            internal ResourceManager(Transactional<T1>parent, Transaction tx)
            {
                this.parent = parent;
                Value = DeepCopy(parent.liveValue);
                currentTransaction = tx;
            }

            public T1 Value { get; set; }
        }
    }
}
```

代码段 CustomResource/ResourceManager.cs

因为临时值可能在事务中变化，所以包装类的实时值可能不会在事务中发生变化。在创建一些类的副本时，可以调用在 `ICloneable` 接口中定义的 `Clone()` 方法。但是，在定义 `Clone()` 方法时，允许实现代码创建浅表副本或深层副本。如果类型 `T` 包含引用类型，并实现了浅表副本，改变临时值也会改变初始值。这会与事务的隔离特性和一致特性冲突。这里需要一个深层副本。

为了进行深层复制，`DeepCopy()` 方法可把对象序列化到流中，并从流中反序列化对象。因为在 C# 4 中，不能定义对类型 `T` 的限制，即指定需要序列化，所以 `Transactional<T>` 类的静态构造函数会检查 `Type` 对象的 `IsSerializable` 属性，以确定类型是否可以序列化。

```
static ResourceManager()
{
    Type t = typeof(T1);
    Trace.Assert(t.IsSerializable, "Type " + t.Name +
        " is not serializable");
}

private T1 DeepCopy(T1 value)
{
    using (MemoryStream stream = new MemoryStream())
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(stream, value);
        stream.Flush();
        stream.Seek(0, SeekOrigin.Begin);

        return (T1)formatter.Deserialize(stream);
    }
}
```

`IEnlistmentNotification` 接口由 `ResourceManager<T>` 类实现。这是用事务进行登记的要求。

只有用 `preparingEnlistment` 调用 `Prepared()` 方法, `Prepare()` 的实现代码才会响应。因为在将临时值赋予实时值时, 不应有问题, 所以 `Prepare()` 方法会成功。在 `Commit()` 方法的实现代码中, 调用父对象的 `Commit()` 方法。其中, 把变量 `liveValue` 设置为在事务中使用的 `ResourceManager` 的值。`Rollback()` 方法仅完成工作, 不改变实时值。对于不稳定的资源, 在 `InDoubt()` 方法中不执行太多的操作。写入一个日志项可能会有用。

```

        public void Prepare(PreparingEnlistment preparingEnlistment)
        {
            preparingEnlistment.Prepared();
        }
        public void Commit(Enlistment enlistment)
        {
            parent.Commit(Value, currentTransaction);
            enlistment.Done();
        }

        public void Rollback(Enlistment enlistment)
        {
            parent.Rollback(currentTransaction);
            enlistment.Done();
        }

        public void InDoubt(Enlistment enlistment)
        {
            enlistment.Done();
        }
    }
}

```

现在, 只要类型是可以序列化的, `Transactional<T>` 类就可以用于使非事务类变成事务类, 例如, `int`、`string`, 甚至更复杂的类, 如 `Student`。



可从
wrox.com
下载源代码

```

using System;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    class Program
    {
        static void Main()
        {
            var intVal = new Transactional<int>(1);
            var student1 = new Transactional<Student>(new Student());
            student1.Value.FirstName = "Andrew";
            student1.Value.LastName = "Wilson";

            Console.WriteLine("before the transaction, value: {0}",
                intVal.Value);
            Console.WriteLine("before the transaction, student: {0}",
                student1.Value);

            using (var scope = new TransactionScope())
            {
                intVal.Value = 2;
            }
        }
    }
}

```

```

        Console.WriteLine("inside transaction, value: {0}", intVal.Value);

        student1.Value.FirstName = "Ten";
        student1.Value.LastName = "SixtyNine";

        if (!Utilities.AbortTx())
            scope.Complete();
    }
    Console.WriteLine("outside of transaction, value: {0}",
        intVal.Value);
    Console.WriteLine("outside of transaction, student: {0}",
        student1.Value);
}
}
}

```

代码段 CustomResource/Program.cs

下面的控制台输出显示了应用程序的一次运行及提交的事务。

```

before the transaction, value: 1
before the transaction: student: Andrew Wilson
inside transaction, value: 2

Abort the Transaction (y/n)? n

outside of transaction, value: 2
outside of transaction, student: Ten SixtyNine

```

23.7 Windows 7 和 Windows Server 2008 的事务

可以编写一个自定义稳定资源管理器，来处理 File 类和 Registry 类。基于文件的稳定资源管理器可以复制原始文件，将对临时文件的修改写入一个临时目录中，使这些改变永久保存起来。在提交事务时，原始文件会用临时文件替代。自 Windows Vista 和 Windows Server 2008 以来，不再需要为文件和注册表编写自定义稳定资源管理器。这两个操作系统及以后的操作系统支持用文件系统和注册表进行本地事务。为此，Windows Vista 和 Windows Server 2008 增加了新的 API 调用，如 CreateFileTransacted()、CreateHardLinkTransacted()、CreateSymbolicLinkTransacted()、CopyFileTransacted() 等。这些 API 调用的共同之处是，它们都需要一个句柄来把事务作为变量传递，它们都不支持环境事务。不能从 .NET 4 中进行事务 API 调用，但可以使用 Platform Invoke 创建一个自定义包装器。



Platform Invoke 详见第 26 章。

示例应用程序包装了本地方法 CreateFileTransacted()，以在 .NET 应用程序中创建事务文件流。

在调用本地方法时，本地方法的参数必须映射到 .NET 数据类型。出于安全考虑，SafeHandle 基类可用来映射本地 HANDLE 类型。SafeHandle 基类是一种抽象类型，它包装了操作系统句柄，并支持句柄资源的关键终止操作。根据句柄的允许值，可以使用派生类 SafeHandleMinusOneIsInvalid 和 SafeHandleZeroOrMinusOneIsInvalid 包装本地句柄。SafeFileHandle 类派生自 SafeHandleZeroOrMinusOneIsInvalid。为了把句柄映射到事务上，定义了 SafeTransactionHandle 类。



可从
wrox.com
下载源代码

```
using System;
using System.Runtime.Versioning;
using System.Security.Permissions;
using Microsoft.Win32.SafeHandles;

namespace Wrox.ProCSharp.Transactions
{
    [SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
    internal sealed class SafeTransactionHandle : SafeHandleZeroOrMinusOneIsInvalid
    {
        private SafeTransactionHandle()
            : base(true) { }

        public SafeTransactionHandle(IntPtr preexistingHandle, bool ownsHandle)
            : base(ownsHandle)
        {
            SetHandle(preexistingHandle);
        }

        [ResourceExposure(ResourceScope.Machine)]
        [ResourceConsumption(ResourceScope.Machine)]
        protected override bool ReleaseHandle()
        {
            return NativeMethods.CloseHandle(handle);
        }
    }
}
```

代码段 FileSystemTransactions/SafeTransactionHandle.cs

.NET 中的所有本地方法都用下面的 `NativeMethods` 类定义。在示例中，所需的本地 API 是 `CreateFileTransacted()` 和 `CloseHandle()` 方法，它们定义为类的静态成员。这些方法声明为 `extern`，因为它们没有 C# 实现代码。其实现代码在本地 DLL 中由 `DllImport` 特性定义。这两个方法都可以在本地 DLL `Kernel32.dll` 中找到。通过方法的声明，把用 Windows API 调用定义的参数映射到 .NET 数据类型。`txHandle` 参数表示事务的一个句柄，其类型是以前定义的 `SafeTransactionHandle`。



可从
wrox.com
下载源代码

```
using System;
using System.Runtime.ConstrainedExecution;
using System.Runtime.InteropServices;
using System.Runtime.Versioning;
using Microsoft.Win32.SafeHandles;

namespace Wrox.ProCSharp.Transactions
{
    internal static class NativeMethods
    {
        [DllImport("Kernel32.dll", CallingConvention = CallingConvention.StdCall,
            CharSet = CharSet.Unicode)]
        internal static extern SafeFileHandle CreateFileTransacted(
            String lpFileName,
            uint dwDesiredAccess,
            uint dwShareMode,
            IntPtr lpSecurityAttributes,
            uint dwCreationDisposition,
            int dwFlagsAndAttributes,
            IntPtr hTemplateFile,
```

```

        SafeTransactionHandle txHandle,
        IntPtr miniVersion,
        IntPtr extendedParameter);

[DllImport("Kernel32.dll", SetLastError = true)]
[ResourceExposure(ResourceScope.Machine)]
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
[return: MarshalAs(UnmanagedType.Bool)]
internal static extern bool CloseHandle(IntPtr handle);
}
}

```

代码段 FileSystemTransactions/NativeMethods.cs

IKernelTransaction 接口用于获得事务句柄，并把它传递给经过事务处理的 Windows API 调用。这是一个 COM 接口，且必须使用 COM Interop 特性包装到 .NET 中，如下面所示。GUID 特性必须有与接口定义相同的标识符，因为这是用于 COM 接口定义的标识符。



可从
wrox.com
下载源代码

```

using System;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.Transactions
{
    [ComImport]
    [Guid("79427A2B-F895-40e0-BE79-B57DC82ED231")]
    [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
    internal interface IKernelTransaction
    {
        void GetHandle(out SafeTransactionHandle ktmHandle);
    }
}

```

代码段 FileSystemTransactions/IKernelTransaction.cs

最后，**TransactionFile** 类正是将由 .NET 应用程序使用的类，这个类定义了 **GetTransactedFileStream()** 方法。这个方法的参数需要一个文件名，并返回一个 **System.IO.FileStream**。返回的流是一个一般的 .NET 流，它只引用一个经过事务处理的文件。

在实现代码中，**TransactionInterop.GetDtcTransaction()** 方法创建了 **IKernelTransaction** 的一个接口指针，它指向环境事务，环境事务作为一个参数传递给 **GetDtcTransaction()** 方法。使用 **IKernelTransaction** 接口，创建 **SafeTransactionHandle** 类型的句柄。然后把这个句柄传递给包装的 API，其名称是 **NativeMethods.CreateFileTransacted()**。使自己返回的文件句柄，新建一个 **FileStream** 实例，并把它返回调用者。



可从
wrox.com
下载源代码

```

using System;
using System.IO;
using System.Security.Permissions;
using System.Transactions;
using Microsoft.Win32.SafeHandles;

namespace Wrox.ProCSharp.Transactions
{
    public static class TransactedFile
    {
        internal const short FILE_ATTRIBUTE_NORMAL = 0x80;
        internal const short INVALID_HANDLE_VALUE = -1;
    }
}

```

```

internal const uint GENERIC_READ = 0x80000000;
internal const uint GENERIC_WRITE = 0x40000000;
internal const uint CREATE_NEW = 1;
internal const uint CREATE_ALWAYS = 2;
internal const uint OPEN_EXISTING = 3;

[FileIOPermission(SecurityAction.Demand, Unrestricted=true)]
public static FileStream GetTransactedFileStream(string fileName)
{
    IKernelTransaction ktx = (IKernelTransaction)
        TransactionInterop.GetDtcTransaction(Transaction.Current);

    SafeTransactionHandle txHandle;
    ktx.GetHandle(out txHandle);

    SafeFileHandle fileHandle = NativeMethods.CreateFileTransacted(
        fileName, GENERIC_WRITE, 0,
        IntPtr.Zero, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
        IntPtr.Zero,
        txHandle, IntPtr.Zero, IntPtr.Zero);

    return new FileStream(fileHandle, FileAccess.Write);
}
}
}

```

代码段 FileSystemTransactions/TransactedFile.cs

现在，很容易在.NET 代码中使用事务 API 了。可以用 `TransactionScope` 类创建一个环境事务，在环境事务的作用域内使用 `TransactedFile` 类。如果终止了事务处理，就不会把数据写入文件中。如果提交了事务，就可以在临时目录下找到该文件。



可从
wrox.com
下载源代码

```

using System;
using System.IO;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
    class Program
    {
        static void Main()
        {
            using (var scope = new TransactionScope())
            {
                FileStream stream = TransactedFile.GetTransactedFileStream(
                    "sample.txt");

                var writer = new StreamWriter(stream);
                writer.WriteLine("Write a transactional file");
                writer.Close();

                if (!Utilities.AbortTx())
                    scope.Complete();
            }
        }
    }
}

```

代码段 Windows7Transactions/Program.cs

现在可以在同一个事务中使用数据库、不稳定的资源和文件。

23.8 小结

本章学习了事务的特性，以及如何使用 `System.Transactions` 名称空间中的类创建和管理事务。

事务用 `ACID` 属性来描述：原子性、一致性、隔离性和持久性。并不是所有这些属性都是必需的，例如，不稳定的资源就不需要支持持久性但可以包含隔离选项。

进行事务处理的最简单的方式是创建环境事务，并使用 `TransactionScope` 类。环境事务非常适合于处理没有显式打开和关闭数据库连接的 `ADO.NET` 数据适配器和 `ADO.NET Entity Framework`，`ADO.NET` 详见第 30 章，“核心 `ADO.NET`” `Entity Framework` 详见第 31 章。

在多个线程中使用同一个事务，可以使用 `DependentTransaction` 类创建对另一个事务的一个依赖关系。登记一个实现了 `IEnlistmentNotification` 接口的资源管理器，可以创建参与事务处理的自定义资源。

最后，探讨了如何通过 `.NET Framework` 和 `C#` 使用 `Windows 7` 和 `Windows Server 2008` 事务。

通过 `.NET Enterprise Services`，可以创建使用 `System.Transactions` 的自动事务。这种技术详见第 51 章，第 51 章可从 `Wrox` 网站下载，或者直接在随书附赠光盘中找到。

下一章介绍如何创建在操作系统启动时自动启动的 `Windows` 服务。事务在服务中也很有用。

第 24 章

网 络

本章内容:

- 从 Web 下载文件
- 在 Windows 窗体应用程序中使用 WebBrowser 控件
- 操纵 IP 地址, 执行 DNS 查询
- 用 TCP、UDP 和套接字类进行套接字编程

本章将采取非常实用的方法, 结合示例讨论相关理论和相应的网络概念。本章并不是计算机网络的指南, 但介绍了如何使用 .NET Framework 进行网络通信。

本章将介绍如何在 Windows 窗体环境中使用 WebBrowser 控件, 以及 WebBrowser 控件如何更方便地完成某些 Internet 访问任务。但本章从最简单的示例开始, 阐明怎样给服务器发送请求和存储返回的信息。

本章将讨论通过 .NET 基类提供的工具, 便于使用各种网络协议(尤其是 HTTP 和 TCP)作为客户访问网络和 Internet。本章将介绍通过 .NET Framework 获得这些协议的一些低级方式, 使用第 43 章介绍的 WCF 技术等还可以实现与这些协议通信的其他方式。

在网络环境下, 我们最感兴趣的两个名称空间是 System.Net 和 System.Net.Sockets。System.Net 名称空间通常与较高层的操作有关, 例如, 下载和上传文件, 使用 HTTP 和其他协议进行 Web 请求等, 而 System.Net.Sockets 名称空间包含的类通常与较低层的操作有关。如果要直接使用套接字或 TCP/IP 之类的协议, 这个名称空间中的类就非常有用, 这些类中的方法与 Windows 套接字 (Winsock) API 函数(派生自 Berkeley 套接字接口)非常类似。本章介绍的一些对象在 System.IO 名称空间中。

第 40~42 章讨论了怎样使用 C#、ASP.NET 编写功能强、效率高的动态 Web 页面。大多数情况下, 访问 ASP.NET 页面的客户使用的是 Internet Explorer 或其他 Web 浏览器, 如 Opera 或 FireFox。但是, 有时需要把 Web 浏览特征添加到自己的应用程序中, 或者需要让自己的应用程序从某个 Web 站点以编程方式获取信息。在后一种情况下, 对于站点, 比较好的解决方案通常是实现一个 Web 服务。但是, 如果访问公共的 Internet 站点, 就不能控制站点的实现方式。

24.1 WebClient 类

如果只想从特定的 URI(Uniform Resource Identifier, 统一资源标识符)请求文件, 则可以使用的

最简单的.NET 类是 `System.Net.WebClient`。这个类是非常高层的类，它只用一两条命令执行基本操作。.NET Framework 目前支持以 `http:`、`https:`和 `file:`标识符开头的 URI。



术语 URL(统一资源定位符)在新的技术规范中已不再使用，现在更常使用的是 URI(统一资源标识符)。URI 的含义大致与 URL 相同，但 URI 更通用，因为它不意味着用户正在使用其中一个熟悉的协议，如 HTTP 或 FTP。

24.1.1 下载文件

使用 `WebClient` 类下载文件有两种方法，具体使用哪一种方法取决于文件内容的处理方式。如果只想把文件保存到磁盘上，就应该使用 `DownloadFile()`方法。这个方法有两个参数：即文件的 URI 和保存所请求的数据的位置(路径和文件名)：

```
WebClient Client = new WebClient();
Client.DownloadFile("http://www.reuters.com/", "ReutersHomepage.htm");
```

更为常见的是，应用程序需要处理从 Web 站点检索到的数据。为此，要使用 `OpenRead()`方法，这个方法返回一个 `Stream` 引用。然后，就可以把数据从数据流中提取到内存中：

```
WebClient Client = new WebClient();
Stream strm = Client.OpenRead("http://www.reuters.com/");
```

24.1.2 基本的 WebClient 示例

第一个示例将阐述怎样使用 `WebClient.OpenRead()`方法。这个示例将把下载的页面显示在 `ListBox` 控件中。首先，作为标准的 C# Windows 窗体应用程序新建项目，添加一个名为 `listBox1` 的列表框，将其 `docking` 属性设置为 `DockStyle.Fill`。在文件的开头，需要在 `using` 指令列表中添加 `System.Net` 和 `System.IO` 名称空间引用，然后对主窗体的构造函数进行以下改动：



可从
wrox.com
下载源代码

```
public Form1()
{
    InitializeComponent();
    WebClient client = new WebClient();
    Stream strm = client.OpenRead("http://www.reuters.com");
    StreamReader sr = new StreamReader(strm);
    string line;

    while ( (line=sr.ReadLine()) != null )
    {
        listBox1.Items.Add(line);
    }

    strm.Close();
}
```

代码段 BasicWebClient.sln

在这个示例中，把 `System.IO` 名称空间中的 `StreamReader` 类与网络数据流关联起来。这样，就可以使用高层方法，例如 `ReadLine()`方法，从数据流中以文本的形式获取数据。第 29 章讨论了把数

据移动抽象化为数据流概念的优点, 这个示例就充分体现出了这一点。

这段示例代码的运行结果如图 24-1 所示。

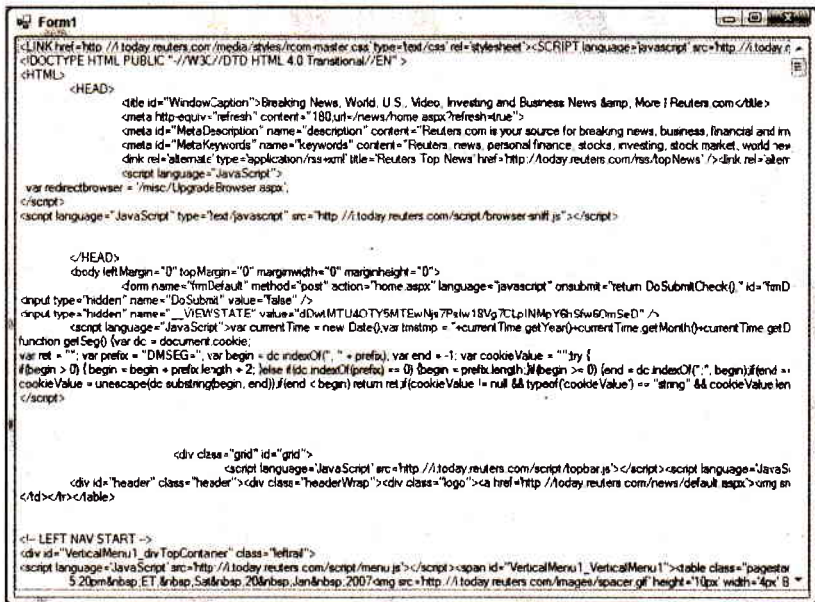


图 24-1

`WebClient` 类还有一个 `OpenWrite()` 方法, 它可以返回一个可写的数据流, 便于用户把数据发送给 `URI`。也可以指定用于把数据发送给主机的方法; 默认的方法是 `POST`。下面的代码段假定在本地计算机上有一个可写的目录 `accept`, 这段代码在该目录下创建 `newfile.txt` 文件, 其内容为 "Hello World":

```
WebClient webClient = new WebClient();
Stream stream = webClient.OpenWrite("http://localhost/accept/newfile.txt", "PUT");
StreamWriter streamWriter = new StreamWriter(stream);
streamWriter.WriteLine("Hello World");
streamWriter.Close();
```

24.1.3 上传文件

`WebClient` 类还提供了 `UploadFile()` 和 `UploadData()` 方法。在需要投递 `HTML` 窗体或上传整个文件时, 就可以使用这两个方法。`UploadFile()` 方法把文件上传到指定的位置, 其中文件名已经给出; 而 `UploadData()` 方法把作为字节数组提供的二进制数据上传至指定的 `URI` (还有一个 `DownloadData()` 方法, 用于从 `URI` 中检索字节数组):

```
WebClient client = new WebClient();
client.UploadFile("http://www.ourwebsite.com/NewFile.htm",
                 "C:\\WebSiteFiles\\NewFile.htm");

byte[] image;
// code to initialize image so it contains all the binary data for
// some jpg file
client.UploadData("http://www.ourwebsite.com/NewFile.jpg", image);
```

尽管 `WebClient` 类使用起来比较简单,但是它的功能非常有限,特别是不能使用它提供身份验证证书。这样,在上传数据时特殊问题就出现了,许多站点都不会接受没有身份验证的上传文件。尽管可以给请求添加标题信息并检查响应中的标题信息,但这仅限于一般意义上的检查,对于任何一个协议,`WebClient` 没有具体的支持。这是由于 `WebClient` 是非常一般的类,可以使用任意协议发送请求和接收响应(如 HTTP、FTP 等)。它不能处理特定于任何协议的任何特性,例如,专用于 HTTP 的 cookie。如果想利用这些特征,就需要使用 `System.Net` 名称空间中以 `WebRequest` 类和 `WebResponse` 类为基础的一系列类。

24.2 WebRequest 类和 WebResponse 类

`WebRequest` 类代表要给某个特定 URI 发送信息的请求,URI 作为参数传递给 `Create()` 方法。`WebResponse` 类代表从服务器检索的数据。调用 `WebRequest.GetResponse()` 方法,实际上是把请求发送给 Web 服务器,并创建一个 `Response` 对象,以检查返回的数据。与 `WebClient` 对象一样,可以得到一个代表数据的数据流,但是,这里的数据流使用 `WebResponse.GetResponseStream()` 方法获得。

本节简要讨论 `WebRequest` 类、`WebResponse` 类和其他相关类支持的几个其他领域。

首先讨论怎样使用这些类下载 Web 页面——这个示例与前面的示例一样,但使用 `WebRequest` 和 `WebResponse` 类。在此过程中,将解释涉及的类的层次结构,然后阐述怎样利用这个层次结构所支持的其他 HTTP 功能。

下面的代码修改了 `BasicWebClient` 示例,以使用 `WebRequest` 类和 `WebResponse` 类。

```
public Form1()
{
    InitializeComponent();

    WebRequest wrq = WebRequest.Create("http://www.reuters.com");
    WebResponse wrs = wrq.GetResponse();
    Stream strm = wrs.GetResponseStream();
    StreamReader sr = new StreamReader(strm);
    string line;

    while ( (line = sr.ReadLine()) != null)
    {
        listBox1.Items.Add(line);
    }

    strm.Close();
}
```

在这段代码中,首先对代表 Web 请求的对象进行实例化。但在此没有使用构造函数,而是调用静态的 `WebRequest.Create()` 方法,如 24.3.6 节所述, `WebRequest` 类是支持不同网络协议的类的层次结构的一部分,为了给请求类型接收一个对正确对象的引用,需要一个工厂机制。`WebRequest.Create()` 方法会为给定的协议创建合适的对象。

HTTP 协议的一个重要方面就是能够利用请求数据流和响应数据流发送扩展的标题信息。标题信息可以包括 cookies 以及发送请求的特定浏览器(用户代理)的详细信息。果然, .NET Framework 为访问最重要的数据提供了全方位的支持。`WebRequest` 类和 `WebResponse` 类提供了读取标题信息的

一些支持。而两个派生的类 `HttpRequest` 和 `HttpResponse` 提供了其他 HTTP 特定的信息。

如 24.3.6 节所述, 用 HTTP URI 创建 `WebRequest` 会生成一个 `HttpRequest` 对象实例。因为 `HttpRequest` 对象实例派生自 `WebRequest` 类, 所以可以在需要 `WebRequest` 类的任何地方使用新实例。另外, 还可以把实例的类型强制转换为 `HttpRequest` 引用, 并访问 HTTP 协议特定的属性。同样, 在使用 HTTP 时, `GetResponse()` 方法调用会把 `HttpResponse` 实例返回为 `HttpResponse` 引用。也可以进行一个简单的强制转换, 以访问 HTTP 特定的功能。

在 `GetResponse()` 方法调用之前添加如下代码, 可以检查两个标题属性的某些内容:

```
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
HttpRequest hwrq = (HttpRequest)wrq;
listBox1.Items.Add("Request Timeout (ms) = " + wrq.Timeout);
listBox1.Items.Add("Request Keep Alive = " + hwrq.KeepAlive);
listBox1.Items.Add("Request AllowAutoRedirect = " + hwrq.AllowAutoRedirect);
```

`Timeout` 属性的单位是毫秒, 其默认值是 100 000。可以设置这个属性, 以控制 `WebRequest` 对象在抛出 `WebException` 异常之前等待相应的响应时间。可以检查 `WebException.Status` 属性, 以查看产生异常的原因。这个枚举包括超时、连接失败、协议错误等的状态码。

因为 `KeepAlive` 属性是对 HTTP 协议的特定扩展, 所以可以通过 `HttpRequest` 引用访问这个属性。`KeepAlive` 属性允许多个请求使用同一个连接, 在后续的请求中节省关闭和重新打开连接的时间。其默认值为 `true`。

`AllowAutoRedirect` 属性也专用于 `HttpRequest` 类, 使用这个属性可以控制 Web 请求是否应自动跟随 Web 服务器上的重定向响应。其默认值也是 `true`。如果只允许有限次数的重定向, 就可以把 `HttpRequest` 类的 `MaximumAutomaticRedirections` 属性设置为期望的数值。

请求类和响应类以属性的形式提供大多数重要的标题, 也可以使用 `Headers` 属性本身显示标题的整个集合。在 `GetResponse()` 方法调用的后面添加如下代码, 可以把所有标题放在列表框控件中:

```
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
WebResponse wrs = wrq.GetResponse();
WebHeaderCollection whc = wrs.Headers;

for(int i = 0; i < whc.Count; i++)
{
    listBox1.Items.Add(string.Format("Header {0}: {1}",
        whc.GetKey(i), whc[i]));
}
```

这段示例代码会产生如图 24-2 所示的标题列表。

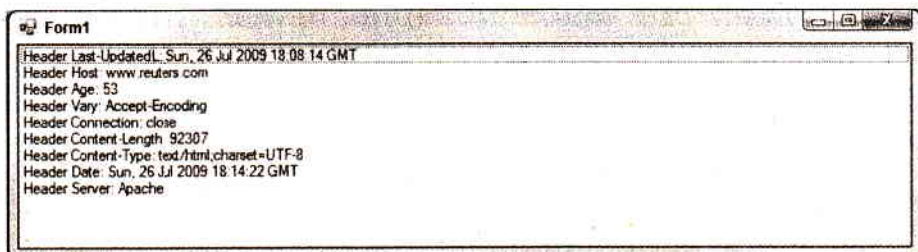


图 24-2

24.2.1 身份验证

`WebRequest` 类中的另一个属性是 `Credentials` 属性。如果需要把身份验证证书附带在请求中, 就可以用用户名和密码创建 `NetworkCredential` 类(也在 `System.Net` 名称空间中)的一个实例。在调用 `GetResponse()` 方法之前, 添加下述代码:

```
NetworkCredential myCred = new NetworkCredential("myusername", "mypassword");
wrq.Credentials = myCred;
```

24.2.2 使用代理

许多公司都需要使用代理服务器进行所有类型的 HTTP 或 FTP 请求。代理服务器常常使用某种形式的安全性(通常是用户名和密码), 路由公司的所有请求和响应。对于使用 `WebClient` 对象或 `WebRequest` 对象的应用程序, 需要考虑这些代理服务器。与前面的 `NetworkCredential` 对象一样, 在进行调用以执行实际请求之前, 需要使用 `WebProxy` 对象。

```
WebProxy wp = new WebProxy("192.168.1.100", true);
wp.Credentials = new NetworkCredential("user1", "user1Password");
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
wrq.Proxy = wp;
WebResponse wrs = wrq.GetResponse();
```

如果除了证书之外, 还需要设计用户的域, 就应在 `NetworkCredential` 实例上使用另一个签名:

```
WebProxy wp = new WebProxy("192.168.1.100", true);
wp.Credentials = new NetworkCredential("user1", "user1Password", "myDomain");
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
wrq.Proxy = wp;
WebResponse wrs = wrq.GetResponse();
```

24.2.3 异步页面请求

`WebRequest` 类的另一个特性就是可以异步地请求页面。这个特性很重要, 因为在给主机发送请求到接收响应之间有很长的延迟。`WebClient.DownloadData()` 和 `WebRequest.GetResponse()` 等方法在响应没有从服务器回来之前, 是不会返回的。如果不希望在那段时间中应用程序处于等待状态, 则最好使用 `BeginGetResponse()` 方法和 `EndGetResponse()` 方法, `BeginGetResponse()` 方法可以异步地工作, 并立即返回。在底层, 运行库会异步地管理一个后台线程, 以从服务器上接收响应。`BeginGetResponse()` 方法不返回 `WebResponse` 对象, 而是返回一个实现 `IAsyncResult` 接口的对象。使用这个接口可以选择或等待可用的响应, 然后调用 `EndGetResponse()` 方法搜集结果。

也可以把一个回调委托传递给 `BeginGetResponse()` 方法。该回调委托的目的地是一个返回类型为 `void` 并把 `IAsyncResult` 引用作为参数的方法, 当工作线程搜集完响应后, 运行库就调用该回调委托, 以通知用户工作已完成。如下面的代码所示, 在回调方法中调用 `EndGetResponse()` 方法可以检索 `WebResponse` 对象:

```
public Form1()
{
    InitializeComponent();
    WebRequest wrq = WebRequest.Create("http://www.reuters.com");
    wrq.BeginGetResponse(new AsyncCallback(OnResponse), wrq);
}
```

```

    }

    protected static void OnResponse(IAsyncResult ar)
    {
        WebRequest wrq = (WebRequest)ar.AsyncState;
        WebResponse wrs = wrq.EndGetResponse(ar);
        // read the response...
    }

```

注意可以作为 `BeginGetResponse()` 的第二个参数传递 `WebRequest` 对象, 检索最初的 `WebRequest` 对象。第二个参数是一个对象引用, 称为状态参数。在回调方法的过程中, 可以使用 `IAsyncResult` 接口的 `AsyncState` 属性检索相同的状态对象。

24.3 把输出结果显示为 HTML 页面

前面几个示例说明了 .NET 基类如何简化从 Internet 上下载和处理数据。但是, 迄今为止, 从 Internet 上下载的文件都是以纯文本显示的。人们总是希望以 Internet Explorer 的界面样式查看 HTML 文件, 其中呈现的 HTML 允许用户查看 Web 文档的实际面貌。但是, Microsoft 的 Internet Explorer 并没有 .NET 版本, 但这并不意味着这个任务不能完成。

在 .NET Framework 2.0 发布之前, 可以引用封装了 Internet Explorer 的 COM(Component Object Model, 组件对象模型)对象, 使用 .NET 交互操作功能, 把应用程序用作浏览器。现在, 自从 .NET Framework 2.0 发布以来, 就可以在 Windows 窗体应用程序中使用内置的 `WebBrowser` 控件。

`WebBrowser` 控件封装了 COM 对象, 甚至可以更方便地完成以前复杂的任务。除了使用 `WebBrowser` 控件之外, 另一个选项是使用编程功能, 从代码中调用 Internet Explorer 实例。

如果不使用新的 `WebBrowser` 控件, 就可以使用 `System.Diagnostics` 名称空间中的 `Process` 类, 通过编程打开 Internet Explorer 进程, 导航到给定的 Web 页。

```

Process myProcess = new Process();
myProcess.StartInfo.FileName = "iexplore.exe";
myProcess.StartInfo.Arguments = "http://www.wrox.com";
myProcess.Start();

```

但是, 上面的代码会把 Internet Explorer 作为单独的窗口打开, 而应用程序并没有与新窗口相连接, 因此不能控制浏览器。

另一方面, 使用 `WebBrowser` 控件, 可以把浏览器作为应用程序的一个集成部分来显示和控制。新的 `WebBrowser` 控件相当复杂, 提供了许多方法、属性和事件。

24.3.1 从应用程序中进行简单的 Web 浏览

为了简单起见, 首先创建一个 Windows 窗体应用程序, 它只有一个 `TextBox` 控件和一个 `WebBrowser` 控件。构建该应用程序, 让最终用户在文本框中输入一个 URL, 并按回车键。`WebBrowser` 控件就会提取 Web 页面, 并显示得到的文档。

在 Visual Studio 2010 设计器中, 应用程序如图 24-3 所示。

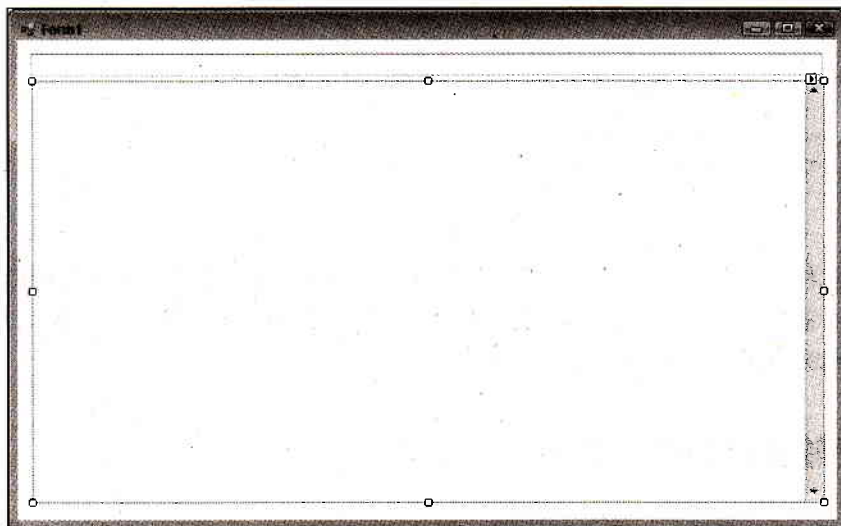


图 24-3

在这个应用程序中，最终用户输入 URL，按回车键后，这个按键动作就会通过应用程序进行注册，然后，WebBrowser 控件就会开始检索请求的页面，然后在该控件中显示它。

该应用程序的代码隐藏如下所示：



可从
wrox.com
下载源代码

```
using System;
using System.Windows.Forms;

namespace Browser
{
    partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();

            private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
            {
                if (e.KeyChar == (char)13)
                {
                    webBrowser1.Navigate(textBox1.Text);
                }
            }
        }
    }
}
```

代码下载 Browser.sln

在这个示例中，最终用户在文本框中按下的每个键都会被 `textBox1_KeyPress` 事件捕获。如果输入的字符是一个回车键(按回车键，其键码是 `(char)13`)，就用 WebBrowser 控件采取行动。使用 WebBrowser 控件的 `Navigate()` 方法，通过 `textBox1.Text` 属性指定 URL(指定为字符串)，最终结果如图 24-4 所示。

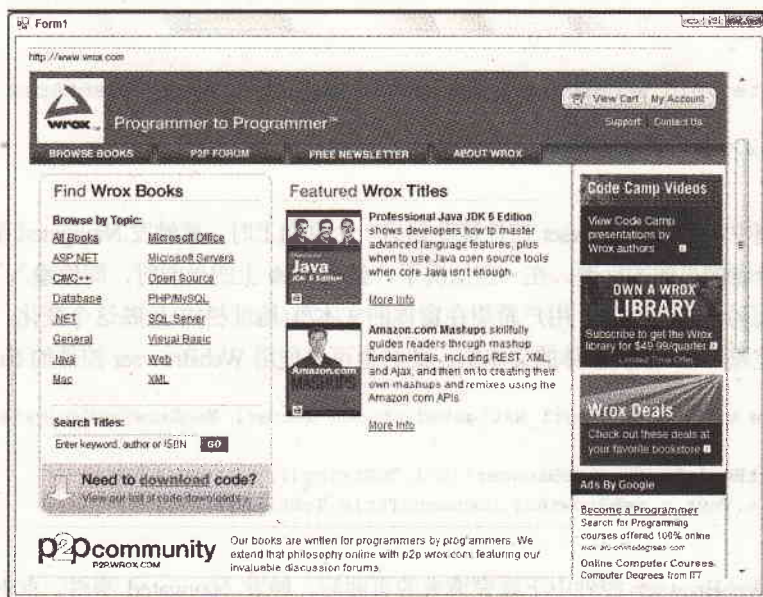


图 24-4

24.3.2 启动 Internet Explorer 实例

读者可能对上一节描述的把浏览器放在应用程序内部不感兴趣，只对让用户在一般的浏览器中查找 Web 站点感兴趣(例如，单击应用程序中的一个链接)。为了演示这个功能，创建一个 Windows 窗体应用程序，其中有一个 LinkLabel 控件。例如，可以在窗体上放置一个 LinkLabel 控件，显示“Visit our company web site!”。

有了这个控件后，就可以使用下面的代码，在一个单独的浏览器中启动公司的 Web 站点，而不是直接在应用程序的窗体中启动：

```
private void linkLabel1_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
{
    WebBrowser wb = new WebBrowser();
    wb.Navigate("http://www.wrox.com", true);
}
```

在这个示例中，用户单击 LinkLabel 控件时，就会新建 WebBrowser 类的一个实例。然后使用 WebBrowser 类的 Navigate()方法，代码指定了 Web 页面的位置和一个布尔值，该布尔值表示是在 Windows 窗体应用程序内部打开这个端点(其值为 false)，还是从一个单独的浏览器中打开这个端点(其值为 true)。它默认设置为 false。在前面的构造过程中，当最终用户单击 Windows 应用程序中的链接时，就实例化一个浏览器实例，并启动 www.Wrox.com 网站。

24.3.3 给应用程序提供更多 IE 类型的功能

在前面的例子中，直接在 Windows 窗体应用程序中使用 WebBrowser 控件时，单击页面中包含的链接，TextBox 控件中的文本不会更新，因此不能显示浏览过程的准确位置的 URL。要更正这个错误，应侦听 WebBrowser 控件中的事件，给控件添加处理程序。

为此，要用 HTML 页面的标题更新窗体的标题。只需使用 Navigated 事件，并更新窗体的 Text

属性即可：

```
private void webBrowser1_Navigated(object sender, EventArgs e)
{
    this.Text = webBrowser1.DocumentTitle.ToString();
}
```

在这个示例中，当 **WebBrowser** 控件移动到另一个页面上时，就触发 **Navigated** 事件，并把窗体的标题改为所查看的页面的标题。在一些情况下，处理 Web 上的页面时，即使输入了指定的地址，也会被重定向到另一个页面上。用户希望在窗体的文本框(地址栏)中反映这个变化。为此，应根据所查看页面的完整 URL 改变窗体的文本框。此时也可以使用 **WebBrowser** 控件的 **Navigated** 事件：

```
private void webBrowser1_Navigated(object sender, WebBrowserNavigatedEventArgs e)
{
    textBox1.Text = webBrowser1.Url.ToString();
    this.Text = webBrowser1.DocumentTitle.ToString();
}
```

这里，在 **WebBrowser** 控件中下载完请求的页面后，触发 **Navigated** 事件。此时，我们只需把 **textBox1** 控件的 **Text** 值更新为页面的 URL 即可。也就是说，页面加载到 **WebBrowser** 控件的 HTML 容器中后，如果 URL 在这个过程中发生变化(例如，有一个重定向过程)，新的 URL 就会显示在文本框中。如果使用这些步骤并导航到 Wrox 网站(<http://www.wrox.com>)，页面的 URL 会立即改为 <http://www.wrox.com/WileyCDA/>。这个过程也说明，如果最终用户单击了 HTML 视图中的其中一个链接，就也会在文本框中显示新请求页面的 URL。

如果进行了这些修改后，运行应用程序，就会发现窗体的标题和地址栏会像 Microsoft 的 Internet Explorer 一样变化，如图 24-5 所示。

接着，创建 IE 样式的工具栏，让最终用户更多地控制 **WebBrowser** 控件。这样，就可以使用 **Back**、**Forward**、**Stop**、**Refresh** 和 **Home** 等按钮。

这里不使用 **ToolBar** 控件，而是在窗体顶部地址栏的上面添加一组 **Button** 控件。在控件的顶部添加 5 个按钮，如图 24-6 所示。

在这个示例中，修改按钮上的文本，以显示按钮的作用。当然，还可以使用屏幕捕捉实用程序，“借用”并使用 IE 的按钮图像。按钮应命名为 **buttonBack**、**buttonForward**、**buttonStop**、**buttonRefresh** 和 **buttonHome**。为了能重置大小，应确保把右边 3 个按钮的 **Anchor** 属性设置为 **Top**、**Right**。

开始时，**buttonBack**、**buttonForward** 和 **buttonStop** 应是禁用的，因为如果没有在 **WebBrowser** 控件中加载初始页面，就不能使用这些按钮。以后应告诉应用程序，根据用户在页面栈的位置，何时启用和禁用 **Back** 按钮和 **Forward** 按钮。另外，在加载页面时，需要启用 **Stop** 按钮，同时，在页



图 24-5

面加载完毕后, 需要禁用 Stop 按钮。在页面上再添加一个 Submit 按钮, 用于提交所请求的 URL。

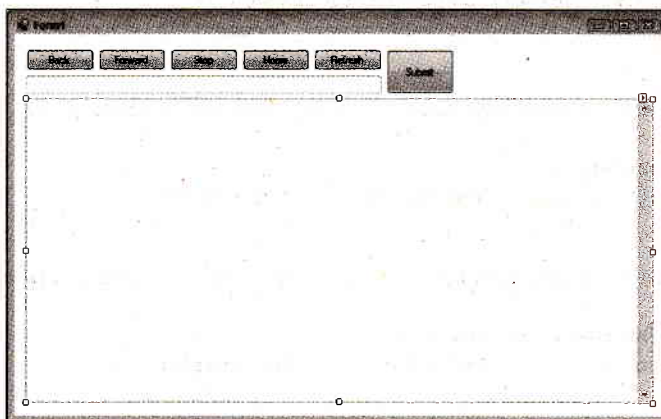


图 24-6

下面首先给按钮添加功能。因为 WebBrowser 类有我们需要的所有方法, 所以这很简单:



可从
wrox.com
下载源代码

```
using System;
using System.Windows.Forms;

namespace Browser
{
    partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
        {
            if (e.KeyChar == (char)13)
            {
                webBrowser1.Navigate(textBox1.Text);
            }
        }

        private void webBrowser1_Navigated(object sender,
            WebBrowserNavigatedEventArgs e)
        {
            textBox1.Text = webBrowser1.Url.ToString();
            this.Text = webBrowser1.DocumentTitle.ToString();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            buttonBack.Enabled = false;
            buttonForward.Enabled = false;
            buttonStop.Enabled = false;

            this.webBrowser1.CanGoBackChanged +=
                new EventHandler(webBrowser1_CanGoBackChanged);
            this.webBrowser1.CanGoForwardChanged +=
```

```
        new EventHandler(webBrowser1_CanGoForwardChanged);
        this.webBrowser1.DocumentTitleChanged +=
            new EventHandler(webBrowser1_DocumentTitleChanged);
    }

    private void buttonBack_Click(object sender, EventArgs e)
    {
        webBrowser1.GoBack();
        textBox1.Text = webBrowser1.Url.ToString();
    }

    private void buttonForward_Click(object sender, EventArgs e)
    {
        webBrowser1.GoForward();
        textBox1.Text = webBrowser1.Url.ToString();
    }

    private void buttonStop_Click(object sender, EventArgs e)
    {
        webBrowser1.Stop();
    }

    private void buttonHome_Click(object sender, EventArgs e)
    {
        webBrowser1.GoHome();
        textBox1.Text = webBrowser1.Url.ToString();
    }

    private void buttonRefresh_Click(object sender, EventArgs e)
    {
        webBrowser1.Refresh();
    }

    private void buttonSubmit_Click(object sender, EventArgs e)
    {
        webBrowser1.Navigate(textBox1.Text);
    }

    private void webBrowser1_Navigating(object sender,
        WebBrowserNavigatingEventArgs e)
    {
        buttonStop.Enabled = true;
    }

    private void webBrowser1_DocumentCompleted(object sender,
        WebBrowserDocumentCompletedEventArgs e)
    {
        buttonStop.Enabled = false;
        if (webBrowser1.CanGoBack)
        {
            buttonBack.Enabled = true;
        }
        else
        {
            buttonBack.Enabled = false;
        }
        if (webBrowser1.CanGoForward)
```

```

        {
            buttonForward.Enabled = true;
        }
        else
        {
            buttonForward.Enabled = false;
        }
    }
}

```

代码段 Browser.sln

在这个示例中要执行许多不同的操作, 因为最终用户在使用这个应用程序时, 有那么多选项。首先, 对于每个按钮单击事件, `WebBrowser` 类都有一个特定的方法来启动该操作。例如, 对于窗体上的 `Back` 按钮, 可以仅使用 `WebBrowser` 控件的 `GoBack()` 方法; 对于 `Forward` 按钮要使用 `GoForward()` 方法; 对于其他按钮要使用 `Stop()`、`Refresh()` 和 `GoHome()` 方法。所以, 很容易创建工具栏, 其操作类似于 Microsoft 的 Internet Explorer。

在第一次加载窗体时, `Form1_Load` 事件禁用相应的按钮。此时, 最终用户可以在文本框中输入一个 URL, 并单击 `Submit` 按钮, 让应用程序检索相应的页面。

为了管理按钮的启用和禁用, 必须输入一组事件。如前所述, 只要开始下载, 就需要启用 `Stop` 按钮。为此, 只需给 `Navigating` 事件添加事件处理程序, 以启用 `Stop` 按钮:

```

private void webBrowser1_Navigating(object sender,
    WebBrowserNavigatingEventArgs e)
{
    buttonStop.Enabled = true;
}

```

接着, 文档加载完毕后, 再次禁用 `Stop` 按钮:

```

private void webBrowser1_DocumentCompleted(object sender,
    WebBrowserDocumentCompletedEventArgs e)
{
    buttonStop.Enabled = false;
}

```

启用和禁用相应的 `Back` 按钮和 `Forward` 按钮, 实际上依赖于在页面栈中后退或前进的功能。这使用 `CanGoForwardChanged()` 和 `CanGoBackChanged()` 事件实现:

```

private void webBrowser1_CanGoBackChanged(object sender, EventArgs e)
{
    if (webBrowser1.CanGoBack)
    {
        buttonBack.Enabled = true;
    }
    else
    {
        buttonBack.Enabled = false;
    }
}

private void webBrowser1_CanGoForwardChanged(object sender, EventArgs e)

```

```
{
    if (webBrowser1.CanGoForward)
    {
        buttonForward.Enabled = true;
    }
    else
    {
        buttonForward.Enabled = false;
    }
}
```

现在运行项目，访问一个 Web 页面，并单击几个链接，还应能使用工具栏，增进浏览体验。最终结果如图 24-7 所示。

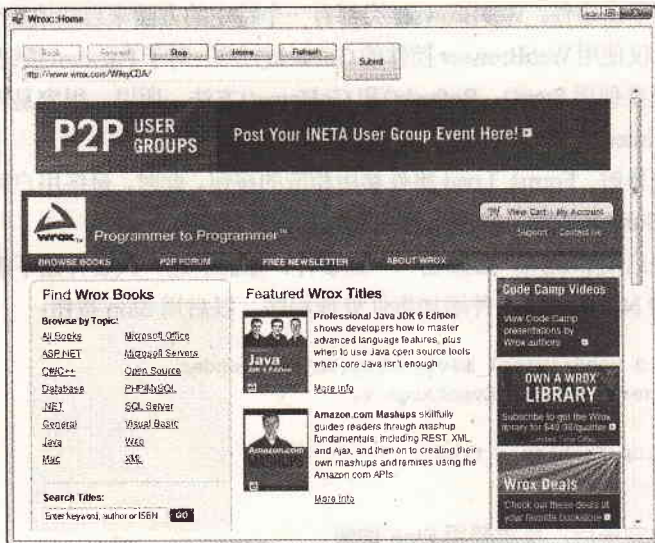


图 24-7

24.3.4 使用 WebBrowser 控件打印

用户不仅可以使⽤ WebBrowser 控件查看页面和文档，还可以使⽤ WebBrowser 控件把这些页面和文档发送到打印机上进行打印。要打印在 WebBrowser 控件中查看的页面或文档，只需使⽤下面的构造函数：

```
webBrowser1.Print();
```

与以前相同，不必查看页面或文档，就可以打印它。例如，可以使⽤ WebBrowser 类加载 HTML 文档，并打印它，而无须显示已加载的文档，其代码如下所示：

```
WebBrowser wb = new WebBrowser();
wb.Navigate("http://www.wrox.com");
wb.Print();
```

24.3.5 显示请求页面的代码

在本章的开头，我们使⽤ WebRequest 类和 Stream 类获得一个远程页面，以显示所请求页面的代码。下面的代码也可以完成这个任务：

```

public Form1()
{
    InitializeComponent();
    System.Net.WebClient Client = new WebClient();
    Stream strm = Client.OpenRead("http://www.reuters.com");
    StreamReader sr = new StreamReader(strm);
    string line;

    while ( (line=sr.ReadLine()) != null )
    {
        listBox1.Items.Add(line);
    }

    strm.Close();
}

```

然而，现在，引入了 WebBrowser 控件后，这个任务就更容易完成。为此，需要修改本章前面开发的浏览器应用程序，只需在 Document_Completed 事件中添加一行代码即可进行修改，如下所示：

```

private void webBrowser1_DocumentCompleted(object sender,
    WebBrowserDocumentCompletedEventArgs e)
{
    buttonStop.Enabled = false;
    textBox2.Text = webBrowser1.DocumentText;
}

```

在应用程序中，在 WebBrowser 控件的下面添加另一个 TextBox 控件。在最终用户请求页面时，不仅要在 TextBox 控件中显示页面的可视化部分，还要显示页面的代码。要显示页面的代码，只需使用 WebBrowser 控件的 DocumentText 属性，它会把整个页面的内容显示为一个字符串。另一个选项是使用 DocumentStream 属性把页面的内容作为一个数据流。添加第二个文本框，可以把页面的内容显示为字符串，结果如图 24-8 所示。

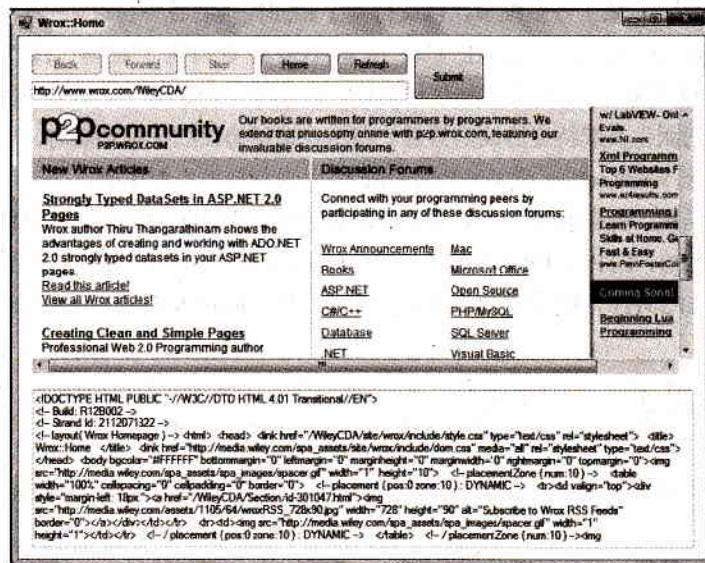


图 24-8

24.3.6 WebRequest 类和 WebResponse 类的层次结构

本节详细讨论 WebRequest 类和 WebResponse 类的底层体系结构。
图 24-9 显示了相关类的继承层次结构。

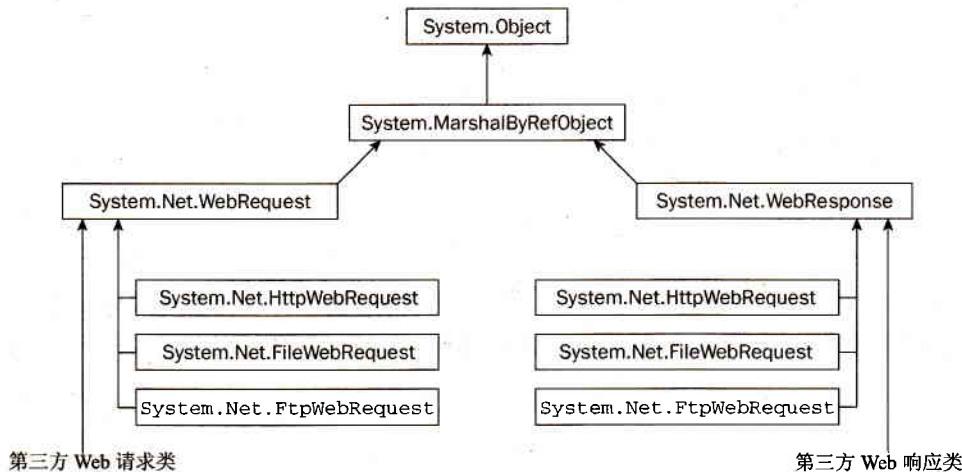


图 24-9

这个层次结构不仅仅包含刚才在代码中使用的两个类。实际上，应该知道 WebRequest 类和 WebResponse 类都是抽象的，不能进行实例化。这些基类提供了用于处理 Web 请求和响应的通用功能，这些功能独立于给定操作所使用的协议。请求总是通过某一特定协议(如 HTTP、FTP、SMTP 等)实现，并由为该协议编写的派生类处理相应请求。Microsoft 称之为“可插入协议”。

在 24.2 节的代码中，变量定义为对基类的引用，但是 WebRequest.Create()方法实际上给出了一个 HttpWebRequest 对象，GetResponse()方法实际上返回一个 HttpWebResponse 对象。这个基于工厂的机制在客户端代码中隐藏了许多细节，以支持基于相同代码的各种协议。

有了 WebRequest.Create()方法，在 URI 中就不需要专门用于处理 HTTP 协议的对象。WebRequest.Create()方法检查 URI 中的协议说明符，以实例化和返回一个适当类的对象。这样代码就不必了解所使用的派生类或特定协议的信息。在需要访问协议的特定功能时，应使用派生类的属性和方法，此时要把 WebRequest 或 WebResponse 引用强制转换为派生类。

有了这个体系结构，就应能使用任意通用协议发送请求。但是，Microsoft 目前提供的派生类只适用于 HTTP、HTTPS、FTP 和 FILE 协议。.NET Framework 自从 2.0 版本以来支持 FTP。如果要利用其他的协议，如 SMTP，则需要使用 WCF(替代了 Windows API)或 SmtpClient 对象。

24.4 实用工具类

本节将讨论一些实用工具类，它们在处理 URI 和 IP 地址时可简化 Web 编程。

24.4.1 URI

Uri 和 UriBuilder 是 System(注意：不是 System.Net)名称空间中的两个类，它们都用于表示 URI。

UriBuilder 类允许把给定的字符串当作 URI 的组成部分,从而构建一个 URI,而 Uri 类允许分析、组合和比较 URI。

对于 Uri 类,构造函数需要一个完整的 URI 字符串:

```
Uri MSPage = new
Uri("http://www.Microsoft.com/SomeFolder/SomeFile.htm?Order=true");
```

Uri 类提供了许多只读属性。当 Uri 对象构造出来之后,它就不能修改了。

```
string Query = MSPage.Query;           // ?Order=true;
string AbsolutePath = MSPage.AbsolutePath; // /SomeFolder/SomeFile.htm
string Scheme = MSPage.Scheme;         // http
int Port = MSPage.Port;                // 80 (the default for http)
string Host = MSPage.Host;              // www.microsoft.com
bool IsDefaultPort = MSPage.IsDefaultPort; // true since 80 is default
```

然而,UriBuilder 类的属性较少:只允许构建一个完整的 URI。这些属性可读写。

可以给构造函数提供构建 URI 所需的各个组件:

```
UriBuilder MSPage = new
UriBuilder("http", "www.microsoft.com", 80, "SomeFolder/SomeFile.htm");
```

或者把值赋给属性,构建 URI 的组件。

```
UriBuilder MSPage = new UriBuilder();
MSPage.Scheme = "http";
MSPage.Host = "www.microsoft.com";
MSPage.Port = 80;
MSPage.Path = "SomeFolder/SomeFile.htm";
```

在完成 UriBuilder 类的初始化后,就可以使用 Uri 属性获得相应的 Uri 对象。

```
Uri CompletedUri = MSPage.Uri;
```

24.4.2 IP 地址和 DNS 名称

在 Internet 上,服务器和客户端都由 IP 地址或主机名(也称作 DNS 名称)标识。通常,主机名是在 Web 浏览器的窗口中输入的友好名称,如 www.wrox.com 或 www.microsoft.com 等。另一方面,IP 地址是计算机用于互相标识的标识符,它实际上是用于确保 Web 请求和响应到达相应计算机的地址。计算机甚至可以有多个 IP 地址。

目前,IP 地址一般是一个 32 位的值。例如 192.168.1.100 就是一个 32 位的 IP 地址。IP 地址的这个格式称为 Internet Protocol 版本 4。目前有许多计算机和其他设备在竞争 Internet 上的一个地点,所以人们开发了一种较新的地址 Internet Protocol 版本 6。IPv6 提供了 64 位的 IP 地址。IPv6 至多可以提供 3×10^{28} 个不同的地址。.NET Framework 允许应用程序使用 IPv4 和 IPv6。

为了使这些主机名发挥作用,首先必须发送一个网络请求,把主机名翻译成 IP 地址,翻译工作由一个或几个 DNS 服务器完成。

DNS 服务器中保存的一个表把主机名映射为它知道的所有计算机的 IP 地址,以及用于在该表中查找它不知道的主机名的其他 DNS 服务器的 IP 地址。本地计算机至少要知道一个 DNS 服务器。

网络管理员在计算机启动时配置该信息。

在发送请求之前，计算机首先应要求 DNS 服务器指出与输入的主机名相对应的 IP 地址。找到正确的 IP 地址后，计算机就可以定位请求，并通过网络发送它。所有这些工作一般都在用户浏览 Web 时在后台进行。

1. 用于 IP 地址的 .NET 类

.NET Framework 提供了许多能够帮助寻找 IP 地址和主机信息的类。

(1) IPAddress 类

IPAddress 类代表 IP 地址。地址本身可以作为 GetAddressBytes 属性，并使用 ToString() 方法把 IP 地址转化为用小点隔开的十进制格式。此外，IPAddress 类也实现静态的 Parse() 方法，这个方法的作用与 ToString() 方法正好相反，把小点隔开的十进制字符串转化为 IP 地址。

```
IPAddress ipAddress = IPAddress.Parse("234.56.78.9");
byte[] address = ipAddress.GetAddressBytes();
string ipString = ipAddress.ToString();
```

在上面的示例中，byte 整型数 address 的值是 IP 地址的二进制表示，字符串 ipString 的值为文本“234.56.78.9”。

IPAddress 类还提供了许多静态的常量字段，以返回特殊的 IP 地址。例如，Loopback 地址允许计算机给它自己发送消息，而 Broadcast 地址允许多播到本地网络上。

```
// The following line will set loopback to "127.0.0.1".
// the loopback address indicates the local host.
string loopback = IPAddress.Loopback.ToString();

// The following line will set broadcast address to "255.255.255.255".
// the broadcast address is used to send a message to all machines on
// the local network.
string broadcast = IPAddress.Broadcast.ToString();
```

(2) IPEndPoint 类

IPEndPoint 类用于封装与某台特定的主机相关的信息。通过这个类的 HostName 属性(这个属性返回一个字符串)，可以使用主机名；通过 AddressList 属性返回一个 IPAddress 对象数组。下一个示例 DnsLookupResolver 将使用 IPEndPoint 类。

(3) Dns 类

Dns 类能够与默认的 DNS 服务器进行通信，以检索 IP 地址。Dns 类有两个重要的静态方法：Resolve() 方法和 GetHostByAddress() 方法。给 Resolve() 方法提供主机名，Resolve() 方法就可以使用 DNS 服务器获取主机的详细信息；给 GetHostByAddress() 方法提供 IP 地址，GetHostByAddress() 方法也可以返回主机的详细信息。这两个方法都返回一个 IPEndPoint 对象。

```
IPEndPoint wroxHost = Dns.Resolve("www.wrox.com");
IPEndPoint wroxHostCopy = Dns.GetHostByAddress("208.215.179.178");
```

在这段代码中，两个 IPEndPoint 对象将包含 Wrox.com 服务器的详细信息。

Dns 类与 IPAddress 类和 IPEndPoint 类的不同之处在于：Dns 实际上可以与服务器进行通信，以获取有关信息；而 IPAddress 类和 IPEndPoint 类只是包含许多便利属性的简单数据结构，可以访

问底层的数据。

2. DnsLookup 示例

下面通过查找 DNS 名称的示例 DnsLookup, 来阐明与 DNS 和 IP 相关的类, 如图 24-10 所示。

该示例让用户在主文本框中输入 DNS 名称, 当用户单击 Resolve 按钮时, 这个示例就使用 Dns.Resolve() 方法检索 IPHostEntry 引用, 并显示主机名和 IP 地址。注意, 显示的主机名也许与输入的名称不同, 如果一个 DNS 名称 (www.microsoft.com) 仅作为另一个 DNS 名称 (lb1.www.ms.akadns.net) 的代理, 就会发生这种情况。

DnsLookup 应用程序是一个标准的 C# Windows 应用程序。把这个应用程序添加如图 24-10 所示的控件, 这些控件分别命名为 textBoxInput、btnResolve、textBoxHostName 和 listBoxIPs。然后, 把下面的方法添加到 Form1 类中, 作为 btnResolve 的 Click 事件的事件处理程序。

```
void btnResolve_Click (object sender, EventArgs e)
{
    try
    {
        IPHostEntry iphost = Dns.GetHostEntry(textBoxInput.Text);
        foreach (IPAddress ip in iphost.AddressList)
        {
            string ipaddress = ip.AddressFamily.ToString();
            listBoxIPs.Items.Add(ipaddress);
            listBoxIPs.Items.Add(" " + ip.ToString());
        }
        textBoxHostName.Text = iphost.HostName;
    }
    catch (Exception ex)
    {
        MessageBox.Show("Unable to process the request because " +
            "the following problem occurred:\n" +
            ex.Message, "Exception occurred");
    }
}
```

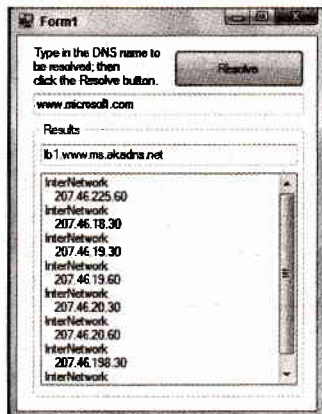


图 24-10

注意, 在这段代码中如何捕获异常。如果用户输入了无效的 DNS 名称, 或者网络处于断开状态, 就会产生异常。

在检索到 IPHostEntry 实例之后, 使用它的 AddressList 属性获取包含 IP 地址的数组, 再用 foreach 循环遍历该数组。对于每一项, 都使用 IPAddress.AddressFamily.ToString() 方法把 IP 地址显示为整数和字符串。

24.5 较低层的协议

本节简要介绍一些用于在较低层次上进行通信的 .NET 类。

System.Net.Sockets 名称空间包含一些相关类。例如，这些类允许直接发送 TCP 网络请求或在某个特定端口上侦听 TCP 网络请求。其中主要的类如表 24-1 所示。

表 24-1

类	用 途
Socket	这个低层的类用于管理连接。WebRequest、TcpClient 和 UdpClient 等类在内部使用这个类
NetworkStream	这个类是从 Stream 派生的，它表示来自网络的数据流
SmtpClient	允许通过 SMTP 发送消息(邮件)
TcpClient	允许创建和使用 TCP 连接
TcpListener	允许侦听引入的 TCP 连接请求
UdpClient	用于为 UDP 客户创建连接(UDP 是 TCP 的一种替代协议，但它没有得到广泛的使用，主要用于本地网络)

网络的通信分为几个不同的层次，本章迄今为止讨论的类都工作在最高层，即处理某些特定命令的一层。如果考虑使用 FTP 传输文件，这个概念就非常容易理解。尽管目前的 GUI 应用程序隐藏了许多 FTP 细节，但在命令行提示符上执行 FTP 还是不久之前的事。在这个环境中，显式地输入一些要发送至服务器的命令，以下载、上传和列出文件。

FTP 并不是依赖于文本命令的唯一高层协议，HTTP、SMTP、POP 和其他协议都基于相似的文本文命令。同样许多现代的图形工具隐藏了命令的传输过程，因此用户一般意识不到这些命令的存在。例如，在 Web 浏览器中输入 URL 和把 Web 请求发送给服务器时，浏览器实际上发送给服务器的是一条纯文本的 GET 命令，这条命令与 FTP 的 get 命令相似。此外，浏览器也可以发送 POST 命令，该命令表示浏览器在请求上附有其他数据。

但是，这些协议本身都不足以实现计算机之间的通信。即使客户和服务器都理解某个协议，如 HTTP，它们仍然不能互相理解，除非另外有协议说明字符是如何传输的，使用的是什么二进制格式？更进一步地，认真考虑最低层问题，什么电压用于代表二进制数据中的 0 和 1？这些问题都需要通过协议配置和规定它们，网络领域的开发人员和硬件工程师通常要查阅协议栈。在列出两个主机进行通信所需的各种协议和机制时，创建一个协议栈，其中既有最高层的协议，也有最底层的协议。这种方法利用模块化和分层的方式获得了高效的通信。

幸运的是，对于大多数的开发工作，我们都不需要使用协议栈或处理电压级别。但是，如果要编写代码，该代码需要在计算机之间进行高效的通信，则需要编写的代码可以直接在计算机之间发送二进制数据包。这是 TCP 之类协议的领域，Microsoft 提供的许多类都允许方便地使用该层次上的二进制数据来工作。

24.5.1 使用 SmtpClient

SmtpClient 对象可以通过 SMTP 传送邮件消息。使用 SmtpClient 对象的一个简单示例如下：

```
SmtpClient sc = new SmtpClient("mail.mySmtpHost.com");
sc.Send("evjen@yahoo.com", "editor@wrox.com",
    "The latest chapter", "Here is the latest.");
```

在其最简单的形式中, 使用了 `SmtpClient` 对象的一个实例。在这个例子中, 该实例还提供给 SMTP 服务器的主机, SMTP 服务器用来在 Internet 上发送邮件消息。还可以使用 `Host` 属性完成相同的任务:

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
sc.Send("evjen@yahoo.com", "editor@wrox.com",
    "The latest chapter", "Here is the latest.");
```

有了 `SmtpClient` 对象后, 就可以调用 `Send()` 方法, 提供 `From` 地址、`To` 地址、主题以及邮件的消息正文。

在许多情况下, 邮件消息都比这里的示例复杂。为此, 还可以给 `Send()` 方法传递一个 `MailMessage` 对象:

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
MailMessage mm = new MailMessage();
mm.From = new MailAddress("evjen@yahoo.com", "Bill Evjen");
mm.To.Add(new MailAddress("editor@wrox.com", "Paul Reese"));
mm.To.Add(new MailAddress("marketing@wrox.com", "Wrox Marketing"));
mm.CC.Add(new MailAddress("publisher@wrox.com", "Barry Pruett"));
mm.Subject = "The latest chapter";
mm.Body = "<b>Here you can put a long message</b>";
mm.IsBodyHtml = true;
mm.Priority = MailPriority.High;
sc.Send(mm);
```

使用 `MailMessage` 对象可以细调构建邮件消息的方式。我们可以发送 HTML 消息, 添加任意多个 `To` 和 `CC` 接收人, 修改消息的优先级, 使用消息编码, 以及添加附件。添加附件的功能在下面的代码段中定义:

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
MailMessage mm = new MailMessage();
mm.From = new MailAddress("evjen@yahoo.com", "Bill Evjen");
mm.To.Add(new MailAddress("editor@wrox.com", "Paul Reese"));
mm.To.Add(new MailAddress("marketing@wrox.com", "Wrox Marketing"));
mm.CC.Add(new MailAddress("publisher@wrox.com", "Barry Pruett"));
mm.Subject = "The latest chapter";
mm.Body = "<b>Here you can put a long message</b>";
mm.IsBodyHtml = true;
mm.Priority = MailPriority.High;
Attachment att = new Attachment("myExcelResults.zip",
    MediaTypeNames.Application.Zip);
mm.Attachments.Add(att);
sc.Send(mm);
```

这段代码创建了一个 `Attachment` 对象, 使用 `Add()` 方法给 `MailMessage` 对象添加该对象, 之后调用 `Send()` 方法。

24.5.2 使用 TCP 类

传输控制协议(TCP)类为连接和发送两个端点之间的数据提供了简单的方法。端点是 IP 地址和端口号的组合。已有的协议很好地定义了端口号,例如,HTTP 使用端口 80,而 SMTP 使用端口 25,Internet 分号机构(即 IANA, <http://www.iana.org/>)把端口号赋予这些已知的服务。除非实现某个已知的服务,否则应选择大于 1024 的端口号。

TCP 流量构成了目前 Internet 上的主要流量。TCP 通常是首选的协议,因为它提供了有保证的传输、错误校正和缓存。TcpClient 类封装了 TCP 连接,提供了许多属性来控制连接,包括缓冲、缓冲区的大小和超时。通过 GetStream()方法请求 NetworkStream 对象时可以附带读写功能。

TcpListener 类用 Start()方法侦听引入的 TCP 连接。当连接请求到达时,可以使用 AcceptSocket()方法返回一个套接字,以与远程计算机通信,或使用 AcceptTcpClient()方法通过高层的 TcpClient 对象进行通信。阐明 TcpListener 类和 TcpClient 类如何工作的最简单的方式是举一个示例。

24.5.3 TcpSend 和 TcpReceive 示例

为了说明这两个类的工作原理,需要构建两个应用程序。第一个应用程序是 TcpSend,如图 24-11 所示。这个应用程序打开一个到服务器的 TCP 连接,并为自己发送 C#源代码。

再创建一个 C# Windows 应用程序,其中的窗体包含两个文本框(txtHost 和 txtPort),分别用于主机名和端口,该窗体还有一个按钮(btnSend),单击它可以启动一个连接。首先,确保包含相关的名称空间:

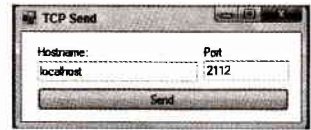


图 24-11

```
using System;
using System.IO;
using System.Net.Sockets;
using System.Windows.Forms;
```

按钮的 Click 事件的事件处理程序如下所示。

```
private void btnSend_Click(object sender, System.EventArgs e)
{
    TcpClient tcpClient = new TcpClient(txtHost.Text, Int32.Parse(txtPort.Text));
    NetworkStream ns = tcpClient.GetStream();
    FileStream fs = File.Open("form1.cs", FileMode.Open);

    int data = fs.ReadByte();

    while(data != -1)
    {
        ns.WriteByte((byte)data);
        data = fs.ReadByte();
    }

    fs.Close();
    ns.Close();
    tcpClient.Close();
}
```

这个示例用主机名和端口号创建 TcpClient 类。另外,如果有 IPEndPoint 类的一个实例,就可以

把该实例传递给 `TcpClient` 类的构造函数。在检索到 `NetworkStream` 类的一个实例后，打开源代码文件，并开始读取字节。与许多二进制流一样，这里也需要将 `ReadByte()` 方法的返回值和 `-1` 相比较，以确定是否到达流的末尾。循环读取了所有的字节，并把它们发送给网络流后，就应关闭所有打开的文件、连接和流。

在连接的另一端，`TcpReceive` 应用程序显示传输完成后接收到的文件，如图 24-12 所示。

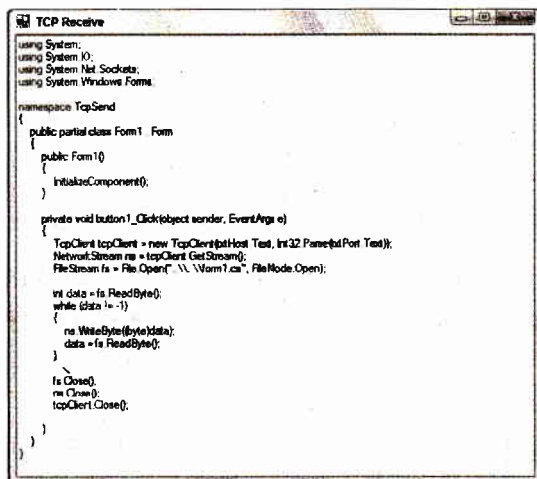


图 24-12

该窗体只包含一个 `TextBox` 控件 `txtDisplay`。`TcpReceive` 应用程序使用 `TcpListener` 等待引入的连接。为了避免应用程序界面的冻结，我们使用一个后台线程来等待，然后从连接中读取。因此还需要包含 `System.Threading` 名称空间以及这些其他名称空间：

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Windows.Forms;
```

在窗体的构造函数中，添加一个后台线程：

```
public Form1()
{
    InitializeComponent();
    Thread thread = new Thread(new ThreadStart(Listen));
    thread.Start();
}
```

其他重要的代码如下所示。

```
public void Listen()
{
    IPAddress localAddr = IPAddress.Parse("127.0.0.1");
    Int32 port = 2112;
    TcpListener tcpListener = new TcpListener(localAddr, port);
    tcpListener.Start();
```

```

        TcpClient tcpClient = tcpListener.AcceptTcpClient();

        NetworkStream ns = tcpClient.GetStream();
        StreamReader sr = new StreamReader(ns);
        string result = sr.ReadToEnd();
        Invoke(new UpdateDisplayDelegate(UpdateDisplay), new object[] { result } );
        tcpClient.Close();
        tcpListener.Stop();
    }

    public void UpdateDisplay(string text)
    {
        txtDisplay.Text = text;
    }

    protected delegate void UpdateDisplayDelegate(string text);

```

该线程在 `Listen()` 方法中开始执行，并允许在不暂停界面的情况下阻塞对 `AcceptTcpClient()` 方法的调用。注意这里把 IP 地址 127.0.0.1 和端口号 2112 硬编码到应用程序中，因此需要从客户端应用程序中输入相同的端口号。

我们使用 `AcceptTcpClient()` 方法返回的 `TcpClient` 对象打开一个新流，进行读取。与本章前面的示例类似，创建一个 `StreamReader`，把引入的网络数据转换为字符串。在关闭客户端和停止侦听程序前，更新窗体的文本框。因为我们不想从后台线程中直接访问文本框，所以使用窗体的 `Invoke()` 方法和一个委托，把得到的字符串作为 `object` 参数数组的第一个元素来传递。`Invoke()` 方法可确保用户的调用正确地封送到线程中，该线程拥有用户界面中的控制句柄。

24.5.4 TCP 和 UDP

本节要介绍的另一个协议是 UDP(用户数据报协议)。UDP 是一个几乎没有什么功能的简单协议，且几乎没有什么系统开销。开发人员常常在速度和性能要求比可靠性更高的应用程序中使用 UDP，例如，视频流。相反，TCP 提供了许多功能来确保数据的传输，它还提供了错误校正，和当数据丢失或数据包损坏时重新传输它们的功能。最后，TCP 可缓存传入和传出的数据，还保证在传输过程中，在把数据包传送给应用程序之前，重组杂乱的一系列数据包。即使有一些额外的开销，TCP 仍是在 Internet 上使用最广泛的协议，因为它有非常高的可靠性。

24.5.5 UDP 类

可以看出，与 `TcpClient` 类相比，`UdpClient` 类提供了一个较小、较简单的界面。这反映出 UDP 协议相对简单的本质。尽管 TCP 类和 UDP 类都在后台使用套接字，但 `UdpClient` 类不包含返回用来网络流以读写数据的方法。相反，成员函数 `Send()` 把一个字节数组作为参数，`Receive()` 函数则返回一个字节数组。另外，因为 UDP 是一个无连接的协议，所以可以指定把通信的端点作为 `Send()` 方法和 `Receive()` 方法的一个参数，而不是在前面的构造函数或 `Connect()` 方法中指定。也可以在某个后续的发送或接收过程中修改端点。

下面的代码段使用 `UdpClient` 类给回应服务(echo service)发送一条消息。带有回显服务的服务器在端口 7 处接受 TCP 或 UDP 连接。回显服务只把发送给服务器的数据再发送回客户端。这个服务可用于诊断和测试(尽管许多系统管理员从安全的角度考虑，禁用回显服务)。

```

using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
namespace Wrox.ProCSharp.InternetAccess.UdpExample
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            UdpClient udpClient = new UdpClient();
            string sendMsg = "Hello Echo Server";
            byte [] sendBytes = Encoding.ASCII.GetBytes(sendMsg);
            udpClient.Send(sendBytes, sendBytes.Length, "SomeEchoServer.net", 7);
            IPEndPoint endPoint = new IPEndPoint(0,0);
            byte [] rcvBytes = udpClient.Receive(ref endPoint);
            string rcvMessage = Encoding.ASCII.GetString(rcvBytes,
                                                         0,
                                                         rcvBytes.Length);

            // should print out "Hello Echo Server"
            Console.WriteLine(rcvMessage);
        }
    }
}

```

Encoding.ASCII 类常常用于把字符串转换为字节数组,或把字节数组转换为字符串。还要注意, IPEndPoint 应通过引用传递给 Receive()方法。因为 UDP 不是一个面向连接的协议,对 Receive()方法的每次调用都会从不同的端点读取数据,所以 Receive()方法会用发送主机的 IP 地址和端口填充该参数。

UdpClient 类和 TcpClient 类在最低层的 Socket 类上提供了一个抽象层。

24.5.6 Socket 类

Socket 类提供了网络编程中最高级的控制。说明该类最简单的方式是用 Socket 类重写 TcpReceive 应用程序。更新后的 Listen()方法如下例所示:

```

public void Listen()
{
    Socket listener = new Socket(AddressFamily.InterNetwork,
                                SocketType.Stream,
                                ProtocolType.Tcp);

    listener.Bind(new IPEndPoint(IPAddress.Any, 2112));
    listener.Listen(0);
    Socket socket = listener.Accept();
    Stream netStream = new NetworkStream(socket);
    StreamReader reader = new StreamReader(netStream);

    string result = reader.ReadToEnd();
    Invoke(new UpdateDisplayDelegate(UpdateDisplay),
           new object[] {result} );
    socket.Close();
}

```

```

        listener.Close();
    }

```

Socket 类需要再编写几行代码来完成相同的任务。对于初学者, 构造函数的参数需要为使用 TCP 协议的流套接字指定 IP 寻址方案。这些参数只可用于 Socket 类的许多组合中的一个, TcpClient 类会配置这些设置。接着把侦听器的套接字绑定到一个端口上, 开始侦听引入的连接。当引入的请求到达时, 就可以使用 Accept() 方法创建一个新的套接字, 来处理该连接。最后为套接字附加一个 StreamReader 实例, 来读取引入的数据, 其方式与前面的大致相同。

Socket 类也包含许多方法, 用于异步地接受、连接、发送和接收数据。通过回调委托使用这些方法的方式与前面用 WebRequest 类请求异步页面的方式相同。如果确实需要了解套接字的内部情况, 就可以使用 GetSocketOption() 方法和 SetSocketOption() 方法, 它们允许查看和配置各种选项, 包括超时、生存期和其他低级选项。下面介绍另一个使用套接字的例子。

1. 构建服务器控制台应用程序

为了进一步探讨 Socket 类, 下一个例子创建一个控制台应用程序, 该应用程序作为引入的入套接字请求的服务器。从那里, 并行地创建第二个例子(另一个控制台应用程序), 它把一条消息发送给服务器控制台应用程序。

第一个应用程序是用作服务器的控制台应用程序, 它会在指定的 TCP 端口上打开一个套接字, 侦听传入的消息。该控制台应用程序的代码如下:

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace SocketConsole
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Starting: Creating Socket object");
            Socket listener = new Socket(AddressFamily.InterNetwork,
                                         SocketType.Stream,
                                         ProtocolType.Tcp);

            listener.Bind(new IPEndPoint(IPAddress.Any, 2112));
            listener.Listen(10);

            while (true)
            {
                Console.WriteLine("Waiting for connection on port 2112");
                Socket socket = listener.Accept();
                string receivedValue = string.Empty;

                while (true)
                {
                    byte[] receivedBytes = new byte[1024];
                    int numBytes = socket.Receive(receivedBytes);
                    Console.WriteLine("Receiving .");
                    receivedValue += Encoding.ASCII.GetString(receivedBytes,

```

```

        0, numBytes);
        if (receivedValue.IndexOf("[FINAL]") > -1)
        {
            break;
        }
    }

    Console.WriteLine("Received value: {0}", receivedValue);
    string replyValue = "Message successfully received.";
    byte[] replyMessage = Encoding.ASCII.GetBytes(replyValue);
    socket.Send(replyMessage);
    socket.Shutdown(SocketShutdown.Both);
    socket.Close();
}
listener.Close();
}
}
}

```

这个例子使用 `Socket` 类建立了一个套接字。该套接字使用 TCP 协议，并通过端口 2112 接收从任意 IP 地址引入的消息。把通过打开的套接字接收到的值写入控制台屏幕上。使用这个应用程序会继续接收字节，直到接收到 `[FINAL]` 字符串为止。这个 `[FINAL]` 字符串表示引入的消息的末尾，之后就可以解释消息了。

从客户端上接收到消息的末尾后，就把一条回应消息发送给该客户端。之后，使用 `Close()` 方法关闭套接字，控制台应用程序继续等待接收新的消息。

2. 构建客户端应用程序

下一步是构建一个客户端应用程序，该应用程序给第一个控制台应用程序发送一条消息。客户端只要遵循该应用程序建立的规则，就可以给服务器控制台应用程序发送任意消息。第一条规则是服务器控制台应用程序只使用特定的协议侦听。本例的服务器应用程序使用 TCP 协议侦听消息。另一条规则是服务器应用程序只侦听特定的端口，对于本例是端口 2112。最后一条规则是对于任意正在发送的消息，消息的最后一位必须以字符串 `[FINAL]` 结尾。

下面的客户端控制台应用程序遵循这些规则：

```

using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace SocketConsoleClient
{
    class Program
    {
        static void Main()
        {
            byte[] receivedBytes = new byte[1024];
            IPHostEntry ipHost = Dns.Resolve("127.0.0.1");
            IPAddress ipAddress = ipHost.AddressList[0];
            IPEndPoint ipEndPoint = new IPEndPoint(ipAddress, 2112);
            Console.WriteLine("Starting: Creating Socket object");

```

```

Socket sender = new Socket(AddressFamily.InterNetwork,
                           SocketType.Stream,
                           ProtocolType.Tcp);

sender.Connect(ipEndPoint);
Console.WriteLine("Successfully connected to {0}",
                 sender.RemoteEndPoint);
string sendingMessage = "Hello World Socket Test";
Console.WriteLine("Creating message: Hello World Socket Test");
byte[] forwardMessage = Encoding.ASCII.GetBytes(sendingMessage
        + "[FINAL]");
sender.Send(forwardMessage);
int totalBytesReceived = sender.Receive(receivedBytes);
Console.WriteLine("Message provided from server: {0}",
                  Encoding.ASCII.GetString(receivedBytes,
        0, totalBytesReceived));
sender.Shutdown(SocketShutdown.Both);
sender.Close();
Console.ReadLine();
}

```

在这个例子中，使用 `localhost` 的 IP 地址和服务器控制台应用程序要求的端口 2112 创建一个 `IPEndPoint` 对象。这里创建了一个套接字，调用了 `Connect()` 方法。打开套接字并连接到服务器控制台应用程序的套接字实例后，使用 `Send()` 方法把一个文本字符串发送给服务器应用程序。因为服务器应用程序会返回一条消息，所以使用 `Receive()` 方法获取这条消息(把它放在一个字节数组中)。之后，将字节数组转换为一个字符串，并把它显示在控制台应用程序上，最后关闭套接字。

运行这个应用程序，结果如图 24-13 所示。



图 24-13

查看图 24-13 中的两个控制台应用程序，会发现服务器应用程序打开并等待引入的消息。引入的消息从客户端应用程序中发送。接着，所发送的字符串由服务器应用程序显示。在接收和显示第一条消息后，服务器应用程序会等待其他消息的传入。关闭客户端应用程序，然后再次运行它，就会看到这一点。接着会看到服务器应用程序再次显示接收到的消息。

24.6 小结

本章回顾了 `System.Net` 名称空间中用于网络通信的 .NET Framework 类。从中可了解到，某些 .NET 基类可处理在网络和 Internet 上打开的客户端连接，如何给服务器发送请求和从服务器接收响应，最常见的应用就是接收 HTML 页面。利用 .NET 4 中的 `WebBrowser` 控件，可以轻松地从桌面应用程序中使用 Internet Explorer。

作为一般的规则，在使用 `System.Net` 名称空间中的类编程时，应尽可能一直使用最通用的类。例如，使用 `TCPClient` 类代替 `Socket` 类，可以把代码与许多低级套接字细节分离开来。更进一步，`WebRequest` 类允许利用 .NET Framework 中可插入协议体系结构。代码应利用新的应用程序级协议，因为 Microsoft 和其他第三方引入了新功能。

最后，我们讨论了网络类中异步功能的使用，该功能给 Windows 窗体应用程序提供了用户响应界面的专业外观。

下一章将介绍 Windows 服务。

第25章

Windows 服务

本章内容:

- Windows 服务的体系结构
- Windows 服务的安装程序
- Windows 服务的控制程序
- Windows 服务的故障排除

Windows 服务是可以在系统启动时自动打开(不需要任何人登录计算机)的程序。首先讨论 Windows 服务的体系结构。之后讨论 Windows 服务的创建、监控、控制和问题的解决。

25.1 Windows 服务

Windows 服务指的是操作系统启动时可以自动打开的应用程序。Windows 服务可以在没有交互式用户登录系统的情况下运行,在后台进行某些处理。

例如,在 Windows Server 上,系统网络服务应可以从客户端访问,无需用户登录到服务器上。在客户端系统上,服务可以从 Internet 上获取新软件版本,或在本地磁盘上进行文件清理工作。

可以把 Windows 服务配置为从已经过特殊配置的用户账户或系统用户账户上运行,用户账户的权限比系统管理员的权限更大。



除非特别说明,否则把 Windows 服务简称为服务。

下面是一些服务的示例:

- Simple TCP/IP Services 是驻留在一些小型 TCP/IP 服务器中的服务程序,如 echo、daytime 和 quote 等。
- World Wide Publishing Service 是 IIS(Internet Information Server, Internet 信息服务器)的服务。
- Event Log 服务用于把消息记录到事件日志系统中。
- Windows Search 服务用于在磁盘上创建数据的索引。

可以使用 Services 管理工具查看系统上的所有服务,如图 25-1 所示。这个程序可以通过管理工具找到。

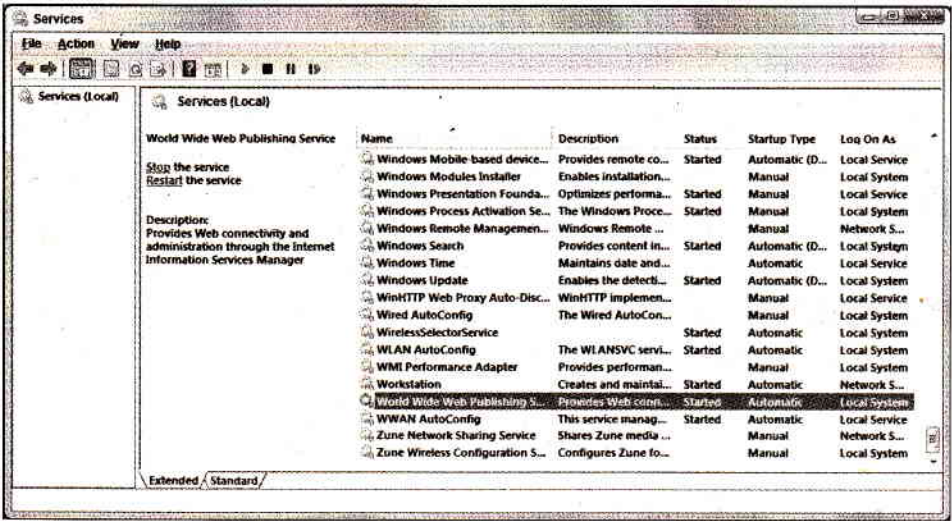


图 25-1

25.2 Windows 服务的体系结构

操作 Windows 服务需要 3 种程序：

- 服务程序
- 服务控制程序
- 服务配置程序

服务程序本身用于提供需要的实际功能。服务控制程序可以把控制请求发送给服务，如开始、停止、暂停和继续。使用服务配置程序可以安装服务，这意味着服务不但要复制到文件系统中，还要写到注册表中，并配置为一个服务。尽管 .NET 组件可只通过 `xcopy` 安装——因为 .NET 组件不需要把信息写入注册表中，所以可以使用 `xcopy` 命令安装它们；但是，服务的安装需要注册表配置。此外，服务配置程序也可以在以后改变服务的配置。

下面介绍 Windows 服务的 3 个组成部分。

25.2.1 服务程序

在讨论服务的 .NET 实现方式之前，从一种独立的观点，首先讨论服务的 Windows 体系结构和服务的内部功能。

服务程序实现服务的功能。服务程序需要 3 部分：

- 主函数
- `service-main` 函数
- 处理程序

在讨论这些部分前，首先需要介绍服务控制管理器(Service Control Manager, SCM)。对于服务，SCM 的作用非常重要，它可以把启动服务或停止服务的请求发送给服务。

1. 服务控制管理器

SCM 是操作系统的一个组成部分，它的作用是与服务进行通信。图 25-2 阐明了这种通信处理序列图的方式。

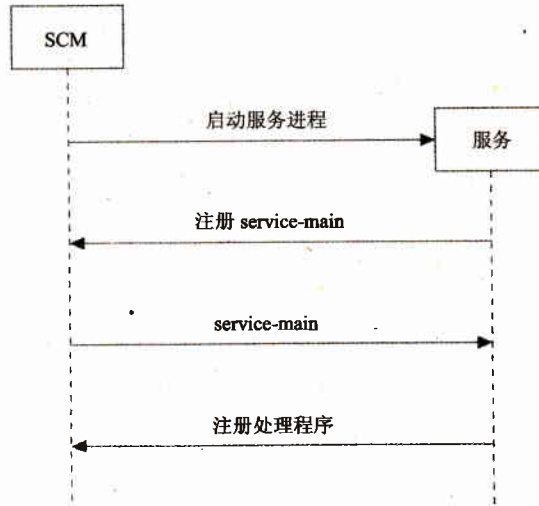


图 25-2

如果将服务设置为自动启动，则在系统启动时，将启动该服务的进程，进而调用该进程的主函数。服务负责为它的每一个服务都注册一个 `service-main` 函数。主函数是服务程序的入口点，在这里，`service-main` 函数的入口点必须用 SCM 注册。

2. 主函数、`service-main` 和处理程序

服务的主函数是程序的一般入口点，即 `Main()` 方法，它可以注册多个 `service-main` 函数，`service-main` 函数包含服务的实际功能。服务必须为所提供的每个服务注册一个 `service-main` 函数。服务程序可以在一个程序中提供许多服务，例如，`<windows>\system32\services.exe` 这个服务程序就包括 `Alert`、`Application Management`、`Computer Browser` 和 `DHCP Client` 等服务项。

SCM 现在为每一个应该启动的服务调用 `service-main` 函数。`service-main` 函数的一个重要任务是用 SCM 注册一个处理程序。

处理程序函数是服务程序的第 3 部分，处理程序必须响应来自 SCM 的事件。服务可以停止、暂停或重新开始，处理程序必须响应这些事件。

使用 SCM 注册了处理程序后，服务控制程序可以把停止、暂停和继续服务的请求发送给 SCM。服务控制程序独立于 SCM 和服务本身。在操作系统中有许多服务控制程序，例如，以前介绍的 MMC `Services` 管理单元。也可以编写自己的服务控制程序，一个比较好的服务控制程序是 `SQL Server Service Manager`，如图 25-3 所示。

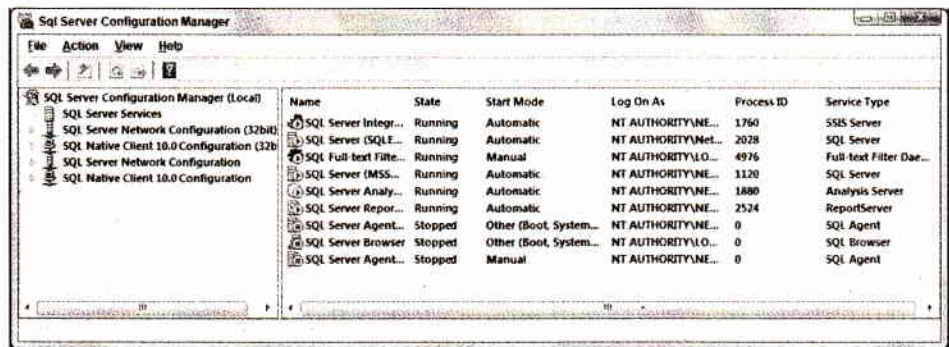


图 25-3

25.2.2 服务控制程序

顾名思义，使用服务控制程序可以控制服务。为了停止、暂停和继续服务，可以把控制代码发送给服务，处理程序应该响应这些事件。此外，还可以询问服务的实际状态，并实现一个响应自定义控制代码的自定义处理程序。

25.2.3 服务配置程序

不能使用 xcopy 安装服务，服务必须在注册表中配置，注册表包含了服务的启动类型，该启动类型可以设置为自动、手动或禁用。必须配置服务程序的用户、服务的依赖关系(例如，一个服务必须在另一个服务开始之前启动)。所有这些配置工作都在服务配置程序中进行。虽然安装程序可以使用服务配置程序配置服务，但是服务配置程序也可以用于在以后改变服务配置参数。

25.2.4 Windows 服务的类

在.NET Framework 中,可以在 System.ServiceProcess 名称空间中找到实现服务的 3 部分的服务类:

- 必须从 ServiceBase 类继承才可以实现服务。ServiceBase 类用于注册服务、响应开始和停止请求。
- ServiceController 类用于实现服务控制程序。使用这个类，可以把请求发送给服务。
- 顾名思义， ServiceProcessInstaller 类和 ServiceInstaller 类用于安装和配置服务程序。

下面介绍怎样新建服务。

25.3 创建 Windows 服务程序

本章创建的服务将驻留在引用服务器内。对于客户发出的每一个请求，引用服务器都返回引用文件的一个随机引用。解决方案的第一部分由 3 个程序集完成，一个用于客户端，两个用于服务器，图 25-4 显示了这个解决方案。程序集 QuoteServer 包含实际的功能。服务可以在内存缓存中读取引用，然后在套接字服务器的帮助下响应引用的请求。QuoteClient 是 WPF 胖客户端应用程序。这个应用程序创建客户端套接字，以便与 Quote Server 进行通信。第 3 个程序集是一个实际的服务，Quote Service 开始和停止 QuoteServer，服务将控制服务器。

在创建程序的服务部分之前，先在额外的 C#类库(在服务进程中使用这个类库)中建立一个简单

的套接字服务器。

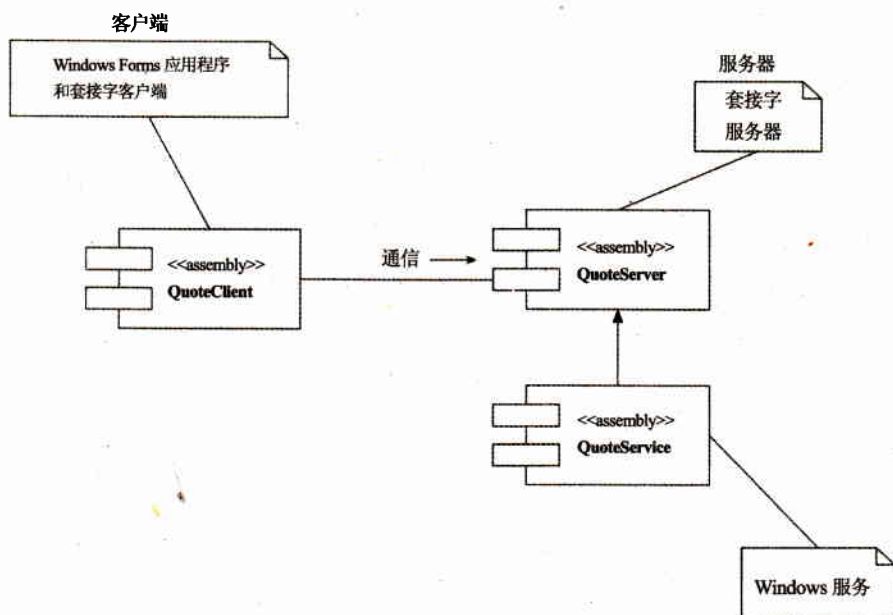


图 25-4

25.3.1 创建服务的核心功能

可以在 Windows 服务中建立任何功能，如扫描文件、进行备份或病毒检查，或者启动 WCF 服务器。但所有服务程序都有一些类似的地方。这种程序必须能启动(并返回给调用者)、停止和暂停。下面讨论用套接字服务器实现的程序。

对于 Windows 7，可以作为 Windows 组件的一个组成部分安装 Simple TCP/IP Services。Simple TCP/IP Services 的一部分是“quote of the day”或 qotd TCP/IP 服务器，这个简单的服务在端口 17 处侦听，并使用文件<windir>\system32\drivers\etc\quotes 中的随机消息响应每一个请求。使用相同的服务，我们将在这里构建一个相似的服务器，它返回一个 Unicode 字符串，而不是像 qotd 服务器那样返回 ASCII 代码。

首先创建一个 QuoteServer 类库，并实现服务器的代码。下面详细解释 QuoteServer.cs 文件中 QuoteServer 类的源代码：



可从
wrox.com
下载源代码

```

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;

namespace Wrox.ProCSharp.WinServices
{
    public class QuoteServer

```

```

{
    private TcpListener listener;
    private int port;
    private string filename;
    private List <string> quotes;
    private Random random;
    private Thread listenerThread;
}

```

代码段 QuoteServer/QuoteServer.cs

重载 QuoteServer()构造函数，以便把文件名和端口传递给主调程序。只传递文件名的构造函数使用服务器的默认端口 7890，默认的构造函数把引用的默认文件名定义为 quotes.txt:

```

public QuoteServer()
    : this ("quotes.txt")
{
}
public QuoteServer(string filename)
    : this(filename, 7890)
{
}
public QuoteServer(string filename, int port)
{
    this.filename = filename;
    this.port = port;
}

```

ReadQuotes()是一个帮助方法，它从构造函数指定的文件中读取所有引用，把所有引用添加到 List<string>引用中。此外，创建 Random 类的一个实例，Random 类用于返回随机引用:

```

protected void ReadQuotes()
{
    quotes = File.ReadAllLines(filename).ToList();
    random = new Random();
}

```

另一个辅助方法是 GetRandomQuoteOfTheDay()，它返回引用集合中的一个随机引用:

```

protected string GetRandomQuoteOfTheDay()
{
    int index = random.Next(0, quotes.Count);
    return quotes[index];
}

```

在 Start()方法中，使用帮助函数 ReadQuotes()，可以在 List<string>引用中读取包含引用的完整文件。在启动新的线程之后，它立即调用 Listener()方法。这类似于第 24 章的 TcpReceive 示例。

这里使用了线程，因为 Start()方法不能停下来等待客户，它必须立即返回给调用者(即 SCM)。如果方法没有及时返回给调用者(30 秒)，SCM 就假定启动失败。把侦听线程设置为后台线程，这样应用程序就可以在不停止该线程的情况下退出。设置线程的 Name 属性，是因为这有助于调试，因为其名称会显示在调试器中:

```

public void Start()
{

```

```

ReadQuotes();
listenerThread = new Thread(ListenerThread);
listenerThread.IsBackground = true;
listenerThread.Name = "Listener";
listenerThread.Start();
}

```

线程函数 `ListenerThread()` 创建一个 `TcpListener` 实例。`AcceptSocket()` 方法等待客户进行连接。客户一连接, `AcceptSocket()` 方法就返回一个与客户相关联的套接字。之后使用 `socket.Send()` 方法, 调用 `GetRandomQuoteOfTheDay()` 方法把返回的随机引用发送给客户:

```

protected void ListenerThread()
{
    try
    {
        IPAddress ipAddress = IPAddress.Parse("127.0.0.1");
        listener = new TcpListener(ipAddress, port);
        listener.Start();
        while (true)
        {
            Socket clientSocket = listener.AcceptSocket();
            string message = GetRandomQuoteOfTheDay();
            UnicodeEncoding encoder = new UnicodeEncoding();
            byte[] buffer = encoder.GetBytes(message);
            clientSocket.Send(buffer, buffer.Length, 0);
            clientSocket.Close();
        }
    }
    catch (SocketException ex)
    {
        Trace.TraceError(String.Format("QuoteServer {0}", ex.Message));
    }
}

```

除了 `Start()` 方法之外, 还需要如下方法来控制服务: `Stop()`、`Suspend()` 和 `Resume()`:

```

public void Stop()
{
    listener.Stop();
}
public void Suspend()
{
    listener.Stop();
}
public void Resume()
{
    Start();
}

```

另一个公共方法是 `RefreshQuotes()`。如果包含引用的文件发生了变化, 就要使用这个方法重新读取文件:

```

public void RefreshQuotes()
{

```

```
        ReadQuotes();
    }
}
```

在服务器上建立服务之前，首先应该建立一个测试程序，这个测试程序仅创建 QuoteServer 类的一个实例，并调用 Start()方法。这样，不需要处理与具体服务相关的问题，就能够测试服务的功能。测试服务器必须手动启动，使用调试器，可以很容易调试代码。

测试程序是一个 C#控制台应用程序 TestQuoteServer，我们必须引用 QuoteServer 类的程序集。包含引用的文件必须复制到 c:\ProCSharp\services 目录中(或者必须在构造函数中改动参数，以指定在什么地方复制文件)。在调用构造函数之后，就调用 QuoteServer 实例的 Start()方法。Start()方法在创建线程之后立即返回，因此，在按回车键之前，控制台应用程序一直处于运行状态。



```
static void Main()
{
    var qs = new QuoteServer("quotes.txt", 4567);
    qs.Start();
    Console.WriteLine("Hit return to exit");
    Console.ReadLine();
    qs.Stop();
}
```

代码段 TestQuoteServer/Program.cs

注意，QuoteServer 示例将运行在使用这个程序的本地主机的 4567 端口上——后面的内容将需要在客户端中使用这些设置。

25.3.2 QuoteClient 示例

客户端是一个简单的 WPF Windows 应用程序，可以在此请求来自服务器的引用。客户端应用程序使用 TcpClient 类连接到正在运行的服务器，然后接收返回的消息，并把它显示在文本框中。用户界面仅包含一个按钮和一个文本框。给按钮的 Click 事件指定 OnGetQuote()方法，把文本框的 x:Name 属性设置为 textQuote。

连接到服务器的服务器名称和端口信息用应用程序的设置来配置。在项目的属性中，可以用 Settings 选项卡来添加设置，如图 25-5 所示。这里定义了 ServerName 和 PortName 设置，以及一些默认值。把 Scope 设置为 User，该设置就会保存到用户特定的配置文件中，应用程序的每个用户都可以有不同的设置。Visual Studio 的 Settings 特性也会创建一个 Settings 类，以便使用一个强类型化的类来读写设置。

必须在代码中使用下面的 using 语句：



```
using System;
using System.Net.Sockets;
using System.Text;
using System.Windows;
using System.Windows.Input;
```

Name	Type	Scope	Value
ServerName	string	User	localhost
PortNumber	int	User	4567
*			

图 25-5

代码段 QuoteClient/MainWindow.xaml.cs

客户端的主要功能体现在 Get Quote 按钮的单击事件的处理程序中。

```
protected void OnGetQuote(object sender, RoutedEventArgs e)
{
    const int bufferSize = 1024;
    Cursor currentCursor = this.Cursor;
    this.Cursor = Cursors.Wait;

    string serverName = Properties.Settings.Default.ServerName;
    int port = Properties.Settings.Default.PortNumber;

    var client = new TcpClient();
    NetworkStream stream = null;
    try
    {
        client.Connect(serverName, port);
        stream = client.GetStream();
        byte[] buffer = new byte[bufferSize];
        int received = stream.Read(buffer, 0, bufferSize);
        if (received <= 0)
        {
            return;
        }
        textQuote.Text = Encoding.Unicode.GetString(buffer).Trim('\0');
    }
    catch (SocketException ex)
    {
        MessageBox.Show(ex.Message, "Error Quote of the day",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
    finally
    {
        if (stream != null)
        {
            stream.Close();
        }
        if (client.Connected)
        {
            client.Close();
        }
    }
    this.Cursor = currentCursor;
}
```

在打开测试服务器和这个 Windows 应用程序的客户端之后, 就可以对功能进行测试。如果运行成功, 就可以得到如图 25-6 所示的结果。

现在继续在服务器中实现服务功能。程序已经在运行, 还需要添加什么呢? 通常, 在系统启动时, 不需要任何人登录系统, 服务器程序就应该自动地启动。我们希望使用服务控制程序对此进行控制。

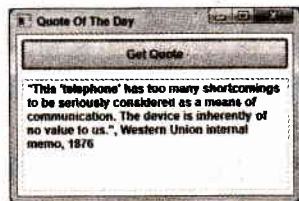


图 25-6

25.3.3 Windows 服务程序

使用 C# Windows 服务的新项目模板, 下面开始创建 Windows 服务程序, 该服务命名为 QuoteService。

在单击 OK 按钮开始创建 Windows 服务程序之后, 就会出现设计器界面, 但是不能在其中插入

UI 组件, 因为应用程序不能直接在屏幕上显示任何信息, 本章后面将使用设计器界面添加安装对象、性能计数器和事件日志等其他组件。

选择这个服务的属性, 可以打开 Properties editor 窗口。在其中可以配置如下值:

- AutoLog 指定把启动和停止服务的事件自动写到事件日志中。
- CanPauseAndContinue、CanShutdown 和 CanStop 指定服务可以处理暂停、继续、关闭和停止服务的请求。
- ServiceName 是写到注册表中的服务的名称, 使用这个名称可以控制服务。
- CanHandleSessionChangeEvent 确定服务是否能处理终端服务会话中的改变事件。
- CanHandlePowerEvent 选项对运行在笔记本电脑或移动设备上的服务有效。如果启用这个选项, 服务就可以响应低电源事件, 并相应地改变服务的行为。电源事件包括电量低、电源状态改变(因为与 A/C 电源之间的切换)开关和改为终止电源。



不管项目的名称是什么, 默认的服务名称都是 Service1。可以只安装一个 Service1 服务。如果在测试过程中出现了安装错误, 就有可能已经安装了一个 Service1 服务。因此, 在服务开发的初始阶段, 一定要用属性编辑器把服务的名称改为比较适当的名称。

使用属性编辑器改变上述属性, 在 InitializeComponent()方法中设置 ServiceBase 的派生类的值。Windows 窗体应用程序也使用 InitializeComponent()方法, 对于服务, 这个方法的使用方式与 Windows 窗体应用程序相似。

虽然向导将生成代码, 但是把文件名改为 QuoteService.cs, 把名称空间的名称改为 Wrox.ProCSharp.WinServices, 并把类名改为 QuoteService。后面将详细讨论该服务的代码。

1. ServiceBase 类

ServiceBase 类是所有用 .NET Framework 开发的 Windows 服务的基类。QuoteService 类派生自 ServiceBase 类; QuoteService 类使用一个未归档的辅助类 System.ServiceProcess.NativeMethods 与 SCM 进行通信, System.ServiceProcess.NativeMethods 类是 Win32 API 调用的包装类。ServiceBase 类是内部的, 因此, 不能在这里的代码中使用它。

图 25-7 显示了 SCM、QuoteService 类和 System.ServiceProcess 名称空间中的类是怎样相互作用的。在这个序列图中, 垂直方向为对象的生命线, 水平方向为通信情况, 通信是按照时间的先后顺序自上而下进行的。

SCM 启动应该启动的服务的进程。首先调用 Main()方法。在示例服务的 Main()方法中, 调用 ServiceBase 基类的 Run()方法。Run()方法使用 SCM 中的 NativeMethods.StartServiceCtrlDispatcher()方法注册 ServiceMainCallback()方法, 并把记录写到事件日志中。

接下来, SCM 在服务程序中调用已注册的 ServiceMainCallback()方法。ServiceMainCallback()方法本身使用 NativeMethods.RegisterServiceCtrlHandler[Ex]()方法在 SCM 中注册处理程序, 并在 SCM 中设置服务的状态。之后调用 OnStart()方法。在 OnStart()方法中, 必须实现启动代码。如果 OnStart()方法执行成功, 就把字符串 “Service Started Sucessful” 写到事件日志中。

处理程序是在 `ServiceCommandCallback()` 方法中执行的。当改变了对服务的请求时，SCM 就调用 `ServiceCommandCallback()` 方法。`ServiceCommandCallback()` 方法再把请求发送给 `OnPause()`、`OnContinue()`、`OnStop()`、`OnCustomCommand()` 和 `OnPowerEvent()` 方法。

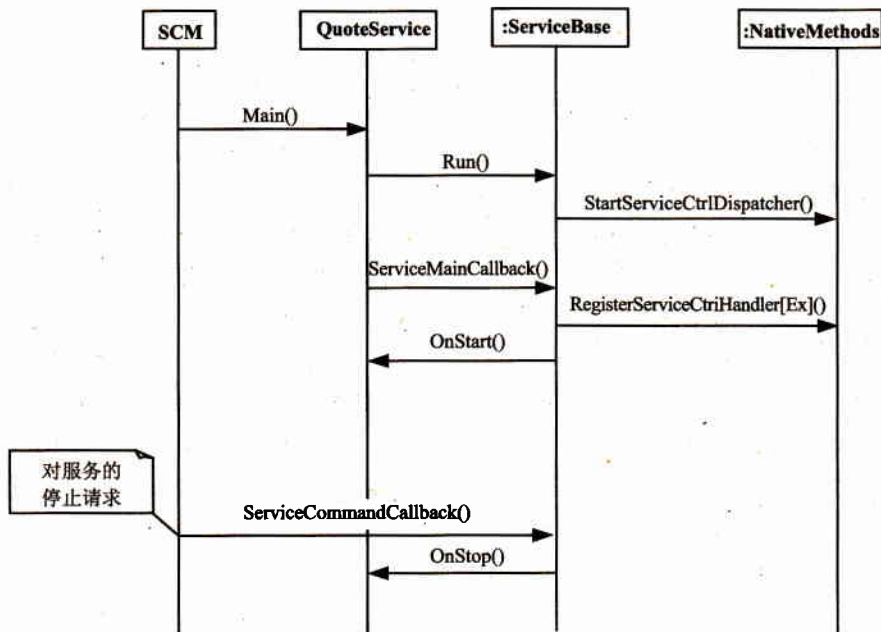


图 25-7

2. 主函数

现在讨论服务进程中由应用程序模板生成的主函数。在主函数中，声明了一个对应元素为 `ServiceBase` 类的数组 `ServicesToRun`。创建 `QuoteService` 类的一个实例，并作为 `ServicesToRun` 数组的第一个元素。如果在这个服务进程中要运行多个服务，就需要把具体服务类的多个实例添加到数组中。然后把 `ServicesToRun` 数组传递给 `ServiceBase` 类的静态方法 `Run()`。使用 `ServiceBase` 类的 `Run()` 方法，可以把 SCM 引用提供给服务的入口点。服务进程的主线程现在处于停滞状态，等待服务的结束。

下面是自动生成的代码：



可从
wrox.com
下载源代码

```

/// <summary>
/// The main entry point for the application.
/// </summary>
static void Main()
{
    ServiceBase[] ServicesToRun;
    ServicesToRun = new ServiceBase[]
    {
        new QuoteService()
    };
    ServiceBase.Run(ServicesToRun);
}
  
```

代码段 QuoteService/Program.cs

如果进程中只有一个服务，就可以删除数组。由于 `Run()` 方法接受从 `ServiceBase` 派生的单个对象，因此 `Main()` 方法可以简化为：

```
ServiceBase.Run(new QuoteService());
```

服务程序 `Services.exe` 包含多个服务。如果有类似的服务，其中有多多个服务运行在一个进程中，且需要初始化多个服务的某些共享状态，则共享的初始化必须在 `Run()` 方法运行之前完成。在运行 `Run()` 方法时，主线程处于停滞状态，直到服务进程停止为止，以后的指令在服务结束之前不能执行。

初始化花费的时间不应该超过 30 秒。如果初始化代码所花费的时间过多，SCM 就认为服务启动失败了。初始化时间不应该超过 30 秒，必须是指针对速度最慢的计算机而言。如果初始化的时间过长，就应该在另一线程中开始初始化，以便主线程及时地调用 `Run()` 方法。然后，事件对象可以用信号通知线程已经完成了它的工作。

3. 服务的启动

在服务启动时，调用 `OnStart()` 方法。这时，可以启动前面创建的套接字服务器。为了使用 `QuoteServer` 类，必须引用 `QuoteServer` 程序集。调用 `OnStart()` 方法的线程不能停滞下来，`OnStart()` 方法必须返回给调用者（即 `ServiceBase` 类的 `ServiceMainCallback()` 方法）。`ServiceBase` 类注册处理程序，并在调用 `OnStart()` 方法之后把服务成功启动的消息通知给 SCM：



可从
wrox.com
下载源代码

```
protected override void OnStart(string[] args)
{
    quoteServer = new QuoteServer(
        Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "quotes.txt"),
        5678);
    quoteServer.Start();
}
```

代码段 QuoteService/QuoteService.cs

把 `quoteServer` 变量声明为类中的私有成员：

```
namespace Wrox.ProCSharp.WinServices
{
    public partial class QuoteService: ServiceBase
    {
        private QuoteServer quoteServer;
```

4. 处理程序方法

当停止服务时，调用 `OnStop()` 方法。应该在 `OnStop()` 方法中停止服务的功能：

```
protected override void OnStop()
{
    quoteServer.Stop();
}
```

除了 `OnStart()` 和 `OnStop()` 方法之外，还可以重写服务类中的下列处理程序：

- `OnPause()`——在暂停服务时，调用这个方法。

- **OnContinue()**——当服务从暂停状态返回到正常操作时，调用这个方法。为了调用已重写的 **OnPause()**方法和 **OnContinue()**方法，**CanPauseAndContinue** 属性必须设置为 **true**。
- **OnShutdown()**——当 Windows 操作系统关闭时，调用这个方法。通常情况下，**OnShutdown()**方法的行为应该与 **OnStop()**方法的实现代码相似。如果需要更多的时间关闭服务，则可以申请更多的时间。与 **OnPause()**方法和 **OnContinue()**方法相似，必须设置一个属性启用这种行为，即 **CanShutdown** 属性必须设置为 **true**。
- **OnPowerEvent()**——在系统的电源状态发生变化时，调用这个方法。电源状态发生变化的信息在 **PowerBroadcastStatus** 类型的参数中，**PowerBroadcastStatus** 是一个枚举类型，其值是 **Battery Low** 和 **PowerStatusChange**。在这个方法中，还可以获得系统是否要挂起(**QuerySuspend**)的信息，此时可以同意或拒绝挂起。电源事件详见本章后面的内容。
- **OnCustomCommand()**——这个处理程序可以为服务控制程序发送过来的自定义命令提供服务。**OnCustomCommand()**方法有一个用于获取自定义命令编号的 **int** 参数，编号的取值范围是 128~256，小于 128 的值是为系统预留的值。在我们的服务中，使用自定义命令编号为 128 的命令重新读取引用文件：

```
protected override void OnPause()
{
    quoteServer.Suspend();
}

protected override void OnContinue()
{
    quoteServer.Resume();
}

public const int commandRefresh = 128;
protected override void OnCustomCommand(int command)
{
    switch (command)
    {
        case commandRefresh:
            quoteServer.RefreshQuotes();
            break;

        default:
            break;
    }
}
```

25.3.4 线程和服务

如前所述，如果服务的初始化花费的时间过多，则 SCM 就假定服务启动失败。为了解决这个问题，必须创建线程。

服务类中的 **OnStart()**方法必须及时返回。如果从 **TcpListener** 类中调用一个 **AcceptSocket()**之类的停滞方法，就必须启动一个线程去完成调用工作。使用能处理多个客户端的网络服务器时，线程池也非常有用。**AcceptSocket()**方法应接收调用，并在线程池的另一个线程中进行处理，这样就不需等待代码的执行，系统看起来似乎是立即响应的。

25.3.5 服务的安装

服务必须在注册表中配置，所有服务都可以在 HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services 中找到。使用 regedit 命令，可以查看注册表项。在注册表中，可以看到服务的类型、显示名称、可执行文件的路径、启动配置，以及其他信息，图 25-8 显示了 W3SVC 服务的注册表配置。

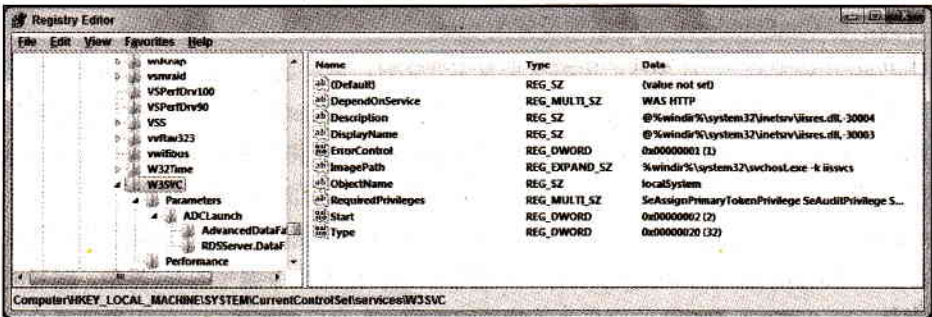


图 25-8

使用 System.ServiceProcess 名称空间中的安装程序类，可以完成服务在注册表中的配置。下面讨论这些内容。

25.3.6 安装程序

切换到 Visual Studio 的设计视图，从弹出的上下文菜单中选择 Add Installer 选项，就可以给服务添加安装程序。使用 Add Installer 选项时，新建一个 ProjectInstaller 类、一个 ServiceInstaller 实例和一个 ServiceProcessInstaller 实例。

图 25-9 显示了服务的安装程序类。

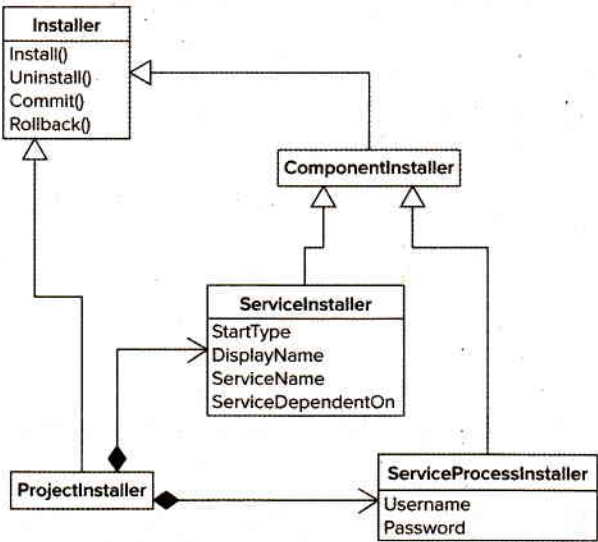


图 25-9

根据图 25-9，下面详细讨论由 Add Installer 选项创建的 ProjectInstaller.cs 文件中的源代码。

1. 安装程序类

`ProjectInstaller` 类派生自 `System.Configuration.Install.Installer`，它是所有自定义安装程序类的基类。使用 `Installer` 类，可以构建基于事务的安装程序。使用基于事务的安装时，如果安装失败，系统就可以回滚到以前的状态，安装程序所做的所有修改都会被取消。如图 25-9 所示，`Installer` 类中有 `Install()`、`Commit()`、`Rollback()` 和 `Uninstall()` 方法，这些方法都从安装程序中调用。

如果 `RunInstaller` 特性的值为 `true`，则在安装程序集时会调用 `ProjectInstaller` 类。自定义安装程序和 `installutil.exe` (这个程序以后将用到) 都能检查该特性。

在 `ProjectInstaller` 类的构造函数内部调用 `InitializeComponent()`：



可从
wrox.com
下载源代码

```
using System.ComponentModel;
using System.Configuration.Install;

namespace Wrox.ProCSharp.WinServices
{
    [RunInstaller(true)]
    public partial class ProjectInstaller: Installer
    {
        public ProjectInstaller()
        {
            InitializeComponent();
        }
    }
}
```

代码段 QuoteService/ ProjectInstaller.cs

2. ServiceProcessInstaller 类和 ServiceInstaller 类

在 `InitializeComponent()` 方法的实现代码中，创建了 `ServiceProcessInstaller` 类和 `ServiceInstaller` 类的实例。这两个类都派生从 `ComponentInstaller` 类，`ComponentInstaller` 类本身派生于 `Installer` 类。

派生自 `ComponentInstaller` 类的类可以用作安装进程的一个部分。注意，一个服务进程可以包括多个服务。`ServiceProcessInstaller` 类用于配置进程，为这个进程中的所有服务定义值，而 `ServiceInstaller` 类用于服务的配置，因此，每个服务都需要 `ServiceInstaller` 类的一个实例。如果进程中有 3 个服务，则必须添加额外的 `ServiceInstaller` 对象，本例需要 3 个 `ServiceInstaller` 实例：



可从
wrox.com
下载源代码

```
partial class projectInstaller
{
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.IContainer components = null;

    /// <summary>
    /// Required method for Designer support to not modify
    /// the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        this.serviceProcessInstaller1 =
            new System.ServiceProcess.ServiceProcessInstaller();
        this.serviceInstaller1 =
```

```
        new System.ServiceProcess.ServiceInstaller();
    //
    // serviceProcessInstaller1
    //
    this.serviceProcessInstaller1.Password = null;
    this.serviceProcessInstaller1.Username = null;
    //
    // serviceInstaller1
    //
    this.serviceInstaller1.ServiceName = "QuoteService";
    //
    // ProjectInstaller
    //
    this.Installers.AddRange(
        new System.Configuration.Install.Installer[]
        {this.serviceProcessInstaller1,
        this.serviceInstaller1});
    }
    private System.ServiceProcess.ServiceProcessInstaller
        serviceProcessInstaller1;
    private System.ServiceProcess.ServiceInstaller serviceInstaller1;
}
```

代码段 QuoteService/ProjectInstaller.Designer.cs

ServiceProcessInstaller 类安装一个实现 ServiceBase 类的可执行文件。ServiceProcessInstaller 类包含用于整个进程的属性。在进程中所有服务共享的属性如表 25-1 所示。

表 25-1

属 性	描 述
Username、 Password	如果把 Account 属性设置为 ServiceAccount.User，则 Username 属性和 Password 属性指出服务在 哪一个账户下运行
Account	使用这个属性，可以指定服务的账户类型
HelpText	HelpText 是只读属性，它返回的文本用于帮助设置用户名和密码

用于运行服务的进程可以用 ServiceProcessInstaller 类的 Account 属性指定，其值可以是 ServiceAccount 枚举的任意值。Account 属性的不同值如表 25-2 所示。

表 25-2

值	意 义
LocalSystem	设置这个值可以指定服务在本地系统上使用权限很高的用户账户，并用作网络上的计算机
NetworkService	类似于 LocalService，这个值指定把计算机的证书传递给远程服务器，但与 LocalService 不同， 这种服务可以以非授权用户的身份登录本地系统。顾名思义，这个账户只能用于需要从网络 上获得资源的服务
LocalService	这个账户类型给任意远程服务器提供计算机的匿名证书，其本地权限与 NetworkService 相同
User	把 Account 属性设置为 ServiceAccount.User，表示可以指定应从服务中使用的账户

`ServiceInstaller` 是每一个服务都需要的类, 这个类的属性可以用于进程中的每一个服务, 其属性有 `StartType`、`DisplayName`、`ServiceName` 和 `ServicesDependentOn`, 如表 25-3 所示。

表 25-3

属 性	说 明
<code>StartType</code>	<code>StartType</code> 属性指出服务是手动启动还是自动启动。它的值可以是: <code>ServiceStartMode.Automatic</code> 、 <code>ServiceStartMode.Manual</code> 、 <code>ServiceStartMode.Disabled</code> 。如果使用 <code>ServiceStartMode.Disabled</code> , 服务就不能启动。这个选项可用于不应在系统中启动的服务。例如, 如果没有得到需要的硬件控制器, 就可以把该选项设置为 <code>Disabled</code>
<code>DelayedAutoStart</code>	如果 <code>StartType</code> 属性没有设置为 <code>Automatic</code> , 就忽略这个属性。此时可以指定服务是否应在系统启动时不启动, 而是在以后启动。这个用于 Windows 服务的属性是 .NET 4 新增的, Windows Vista 操作系统及以后版本支持它
<code>DisplayName</code>	<code>DisplayName</code> 属性是服务显示给用户的友好名称。这个名称也用于控制和监视服务的管理工具
<code>ServiceName</code>	<code>ServiceName</code> 属性是服务的名称。这个值必须与服务程序中 <code>ServiceBase</code> 类的 <code>ServiceName</code> 属性一致, 这个名称把 <code>ServiceInstaller</code> 类的配置与需要的服务程序关联起来
<code>ServicesDependentOn</code>	指定必须在服务启动之前启动的一个服务数组。当服务启动时, 所有依赖的服务都自动启动, 并且我们的服务也将启动



如果在 `ServiceBase` 的派生类中改变了服务的名称, 则还必须修改 `ServiceInstaller` 对象中 `ServiceName` 属性的值。



在测试阶段, 最好把 `StartType` 属性的值设置为 `Manual`。如果服务因程序中的错误不能停止, 就仍可以重新启动系统。如果把 `StartType` 属性的值设置为 `Automatic`, 服务就会在重新启动系统时自动启动! 当确信没有问题时, 可以在以后改变这个配置。

3. `ServiceInstallerDialog` 类

`System.ServiceProcess.Design` 名称空间中的另一个安装程序类是 `ServiceInstallerDialog`。在安装过程中, 如果希望系统管理员输入该服务应使用的账号(具体方法是指定用户名和密码), 就可以使用这个类。

如果把 `ServiceProcessInstaller` 类的 `Account` 属性设置为 `ServiceAccount.User`, `Username` 和 `Password` 属性设置为 `null`, 则在安装时, 将自动显示图 25-10 所示的 `Set Service Login` 对话框。此时, 也可以取消安装。

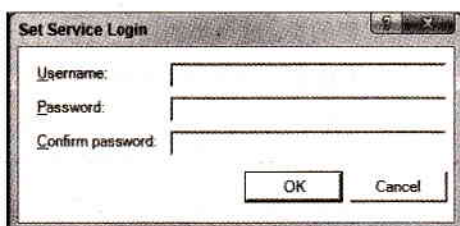


图 25-10

4. installutil

在把安装程序类添加到项目中之后,就可以使用 `installutil.exe` 实用程序安装和卸载服务。这个实用程序可以用于安装包含 `Installer` 类的所有程序集。`installutil.exe` 实用程序调用派生自 `Installer` 类的 `Installer()` 方法进行安装,调用 `UnInstaller()` 方法进行卸载。

安装和卸载服务的命令分别是:

```
installutil quoteservice.exe  
installutil /u quoteservice.exe
```



如果安装失败了,一定要检查安装日志文件 `InstallUtil.InstallLog` 和 `<servicename>.InstallLog`。通常,在安装日志文件中可以发现一些非常有用的信息,例如:“指定的服务已存在”。

5. 客户端

在成功地安装服务后,就可以从 `Services MMC` 中手动启动服务(详细内容请参阅 25.4 节),并启动客户端应用程序。

25.4 服务的监视和控制

可以使用 `Services MMC` 管理单元对服务进行监视和控制。`Services MMC` 管理单元是 `Computer Management` 管理工具的一部分。每个 Windows 操作系统还有一个命令行实用程序 `net.exe`,使用这个程序可以控制服务。`sc.exe` 是另一个命令行实用程序,它的功能比 `net.exe` 更强大。还可以使用 `Visual Studio Server Explorer` 直接控制服务。本节将创建一个小的 Windows 应用程序,它利用 `System.ServiceProcess.ServiceController` 类监视和控制服务。

25.4.1 MMC 管理单元

如图 25-11 所示,使用 MMC 的 `Services` 管理单元,可以查看所有服务的状态,也可以把停止、启用或禁用服务的控制请求发送给服务,并改变它们的配置。`Services` 管理单元既是服务控制程序,又是服务配置程序。

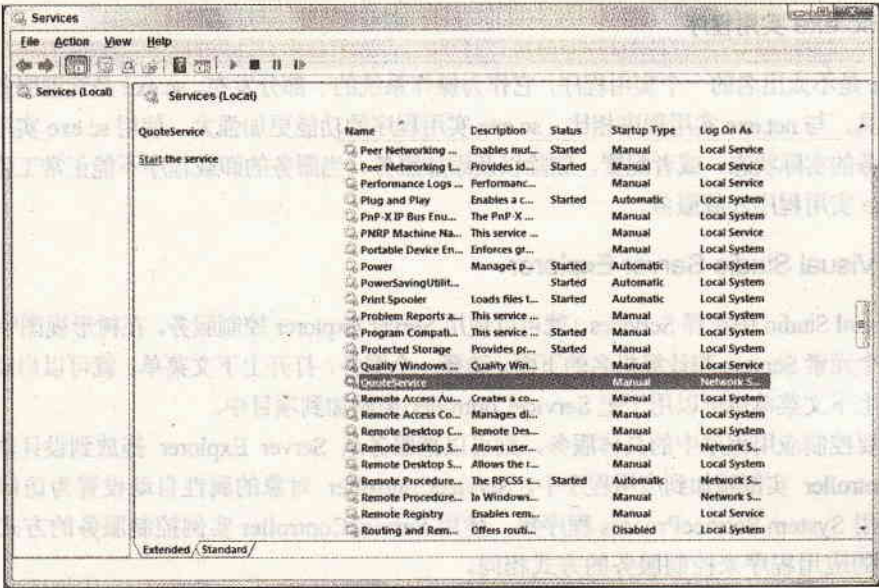


图 25-11

双击 QuoteService，打开如图 25-12 所示的 QuoteService Properties 对话框。在这个对话框中，可以看到服务的名称、描述、可执行文件的路径、启动类型和状态。目前服务已启动。使用这个对话框中的 Log On 选项卡，可以改变服务进程的账户。

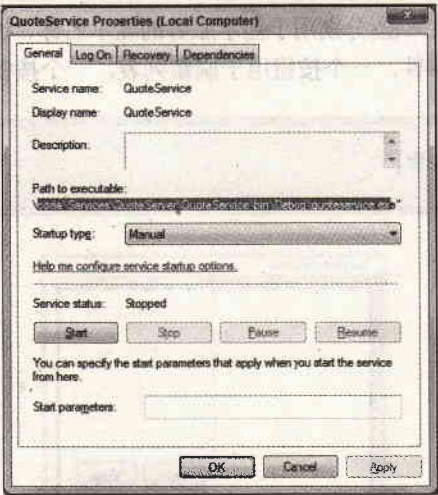


图 25-12

25.4.2 net.exe 实用程序

Services 管理单元使用起来很简单，但是系统管理员不能使其自动化，原因是它不能用在管理脚本中。要通过脚本实现的工具自动控制服务，可以用命令行实用程序 net.exe 命令来完成。net start 命令显示所有正在运行的服务，net start servicename 启动服务，net stop servicename 向服务发送停止请求。此外使用 net pause 和 net continue 可以暂停和继续服务(当然，它们只有在服务允许的情况下才能使用)。

25.4.3 sc.exe 实用程序

sc.exe 是不太出名的一个实用程序，它作为操作系统的一部分发布。sc.exe 是管理服务的一个很有用的工具。与 net.exe 实用程序相比，sc.exe 实用程序的功能更加强大。使用 sc.exe 实用程序，可以检查服务的实际状态，或者配置、删除以及添加服务。当服务的卸载程序不能正常工作时，可以使用 sc.exe 实用程序卸载服务。

25.4.4 Visual Studio Server Explorer

在 Visual Studio 中选择 Services，就可以使用 Server Explorer 控制服务。在树形视图中，Services 项在第一个元素 Servers 和计算机名的下面。选择一个服务，打开上下文菜单，就可以启动和停止服务。这个上下文菜单也可以用于把 ServiceController 类添加到项目中。

如果要控制应用程序中的具体服务，则可以把服务从 Server Explorer 拖放到设计器中：即把 ServiceController 实例添加到应用程序中，ServiceController 对象的属性自动设置为访问选中的服务，并引用 System.ServiceProcess 程序集。使用 ServiceController 实例控制服务的方式与使用下一节开发的应用程序来控制服务的方式相同。

25.4.5 编写自定义 ServiceController 类

下面创建一个小的 Windows 应用程序，该应用程序使用 ServiceController 类监视和控制 Windows 服务。

创建一个 WPF 应用程序，其用户界面如图 25-13 所示。这个应用程序的主窗口包含一个显示所有服务的列表框、4 个文本框(分别用于显示服务的显示名称、状态、类型和名称)和 6 个按钮，其中 4 个按钮用于发送控制事件，一个按钮用于刷新列表，一个按钮用于退出应用程序。

 WPF 详见第 35 章。

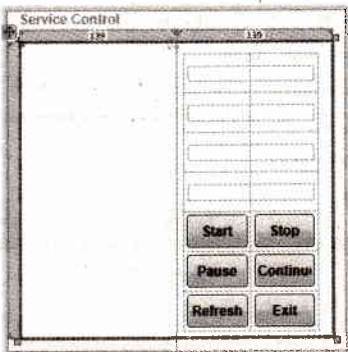


图 25-13

1. 服务的监视

使用 ServiceController 类，可以获取每一个服务的相关信息。表 25-4 列出了 ServiceController 类的属性。

表 25-4

属 性	描 述
CanPauseAndContinue	如果暂停和继续服务的请求可以发送给服务, 则这个属性返回 true
CanShutdown	如果服务有助于关闭系统的处理程序, 则它的值为 true
CanStop	如果服务是可以停止的, 则它的值为 true
DependentServices	它返回一个依赖服务的集合。如果停止服务, 则所有依赖的服务都预先停止
ServicesDependentOn	返回这个服务所依赖的服务集合
DisplayName	指定服务应该显示的名称
MachineName	指定运行服务的计算机名
ServiceName	指定服务的名称
ServiceType	指定服务的类型。服务可以运行在共享的进程中。在共享的进程中, 多个服务使用同一进程(Win32ShareProcess), 此外, 服务也可以运行在只包含一个服务的进程(Win32OwnProcess)中。如果服务可以与桌面交互, 对应的类型就是 InteractiveProcess
Status	指定服务的状态。状态可以是正在运行、停止、暂停或处于某些中间模式(如启动待决、停止待决)等。状态值在 ServiceControllerStatus 枚举中定义

在示例应用程序中, 使用 DisplayName、ServiceName、ServiceType 和 Status 属性显示服务信息。此外, CanPauseAndContinue 和 CanStop 属性用于启用和禁用 Pause、Continue 和 Stop 按钮。

为了得到用户界面的所有必要信息, 创建一个 ServiceControllerInfo 类。这个类可以用于数据绑定, 并提供状态信息、服务名称、服务类型, 以及哪些控制服务的按钮应启用或禁用的信息。



因为使用了 System.ServiceProcess.ServiceController 类, 所以必须引用 System.ServiceProcess 程序集。

ServiceControllerInfo 类包含一个嵌入的 ServiceController 类, 它用 ServiceControllerInfo 类的构造函数设置。还有一个只读属性 Controller, 它用来访问嵌入的 ServiceController 类。



可从
wrox.com
下载源代码

```
public class ServiceControllerInfo
{
    private readonly ServiceController controller;

    public ServiceControllerInfo(ServiceController controller)
    {
        this.controller = controller;
    }

    public ServiceController Controller
    {
        get { return controller; }
    }
}
```

代码段 ServiceControl/ServiceControllerInfo.cs

为了显示服务的当前信息, 可以使用 ServiceControllerInfo 类的只读属性 DisplayName、

ServiceName、ServiceTypeName 和 ServiceStatusName。DisplayName 和 ServiceName 属性的实现代码只访问底层类 ServiceController 的 DisplayName 和 ServiceName 属性。通过 ServiceTypeName 和 ServiceStatusName 属性的实现代码中，完成更多的工作：服务的状态和类型不太容易返回，因为要显示一个字符串，而不是只显示 ServiceController 类返回的数字。ServiceTypeName 属性返回一个表示服务类型的字符串。从 ServiceController.ServiceType 属性中得到的 ServiceType 代表一组标记，使用按位 OR 运算符，可以把这组标记组合在一起。InteractiveProcess 位可以与 Win32OwnProcess 和 Win32ShareProcess 一起设置。首先，在检查其他值之前，一定要先检查 InteractiveProcess 位以前是否设置过。使用这些服务，返回的字符串将是“Win32 Service Process”或“Win32 Shared Process”。

```
public string ServiceTypeName
{
    get
    {
        ServiceType type = controller.ServiceType;
        string serviceTypeName = "";
        if ((type & ServiceType.InteractiveProcess) != 0)
        {
            serviceTypeName = "Interactive ";
            type -= ServiceType.InteractiveProcess;
        }
        switch (type)
        {
            case ServiceType.Adapter:
                serviceTypeName += "Adapter";
                break;

            case ServiceType.FileSystemDriver:
            case ServiceType.KernelDriver:
            case ServiceType.RecognizerDriver:
                serviceTypeName += "Driver";
                break;

            case ServiceType.Win32OwnProcess:
                serviceTypeName += "Win32 Service Process";
                break;

            case ServiceType.Win32ShareProcess:
                serviceTypeName += "Win32 Shared Process";
                break;

            default:
                serviceTypeName += "unknown type " + type.ToString();
                break;
        }
        return serviceTypeName;
    }
}

public string ServiceStatusName
{

```

```

get
{
    switch (controller.Status)
    {
        case ServiceControllerStatus.ContinuePending:
            return "Continue Pending";
        case ServiceControllerStatus.Paused:
            return "Paused";
        case ServiceControllerStatus.PausePending:
            return "Pause Pending";
        case ServiceControllerStatus.StartPending:
            return "Start Pending";
        case ServiceControllerStatus.Running:
            return "Running";
        case ServiceControllerStatus.Stopped:
            return "Stopped";
        case ServiceControllerStatus.StopPending:
            return "Stop Pending";
        default:
            return "Unknown status";
    }
}

public string DisplayName
{
    get { return controller.DisplayName; }
}

public string ServiceName
{
    get { return controller.ServiceName; }
}

```

ServiceControllerInfo 类还有一些属性可以启用 Start、Stop、Pause 和 Continue 按钮: **EnableStart**、**EnableStop**、**EnablePause** 和 **EnableContinue**, 这些属性根据服务的当前状态返回一个布尔值。

```

public bool EnableStart
{
    get
    {
        return controller.Status == ServiceControllerStatus.Stopped;
    }
}

public bool EnableStop
{
    get
    {
        return controller.Status == ServiceControllerStatus.Running;
    }
}

```

```

public bool EnablePause
{
    get
    {
        return controller.Status == ServiceControllerStatus.Running & &
            controller.CanPauseAndContinue;
    }
}

public bool EnableContinue
{
    get
    {
        return controller.Status == ServiceControllerStatus.Paused;
    }
}
}

```

在 `ServiceControlWindow` 类中, `RefreshServiceList()` 方法使用 `ServiceController.GetServices()` 方法获取在列表框中显示的所有服务。`GetServices()` 方法返回一个 `ServiceController` 实例数组, 它们表示在操作系统上安装的所有 Windows 服务。`ServiceController` 类还有一个静态方法 `GetDevices()`, 该方法返回一个表示所有设备驱动程序的 `ServiceController` 数组。返回的数组利用泛型方法 `Array.Sort()` 按照 `DisplayName` 属性来排序, 这是传递给 `Sort()` 方法的 Lambda 表达式定义的属性。使用 `Array.ConvertAll()` 方法, 将 `ServiceController` 实例转换为 `ServiceControllerInfo` 类型。这里传递了一个 Lambda 表达式, 它调用每个 `ServiceController` 对象的 `ServiceControllerInfo()` 构造函数。最后, 将 `ServiceControllerInfo` 数组赋予窗口的 `DataContext` 属性, 进行数据绑定。



可从
wrox.com
下载源代码

```

protected void RefreshServiceList()
{
    ServiceController[] services = ServiceController.GetServices();

    Array.Sort(services, (s1, s2) =>
        s1.DisplayName.CompareTo(s2.DisplayName));

    this.DataContext =
        Array.ConvertAll(services, controller =>
            new ServicesControllerInfo(controller));
}

```

代码段 `ServiceControl/ServiceControlWindow.xaml.cs`

在列表框中获得所有服务的 `RefreshServiceList()` 方法在 `ServiceControlWindow` 类的构造函数中调用。这个构造函数还为按钮的 `Click` 事件定义了事件处理程序:

```

public ServiceControlWindow()
{
    InitializeComponent();
    RefreshServiceList();
}

```

现在, 就可以定义 XAML 代码, 把信息绑定到控件上。

首先, 为显示在列表框中的信息定义一个 `DataTemplate`。列表框包含一个标签, 其 `Content` 属性

绑定到数据源的 `DisplayName` 属性上。在绑定 `ServiceControllerInfo` 对象数组时，用 `ServiceControllerInfo` 类定义 `DisplayName` 属性：



可从
wrox.com
下载源代码

```
<Window.Resources>
    <DataTemplate x:Key="listTemplate">
        <Label Content="{Binding Path=DisplayName}" />
    </DataTemplate>
</Window.Resources>
```

代码段 `ServiceControl/ServiceControlWindow.xaml`

放在窗口左边的列表框将 `ItemsSource` 属性设置为 `{Binding}`。这样，显示在列表中的数据就从 `RefreshServiceList()` 方法设置的 `DataContext` 属性中获得。`ItemTemplate` 属性引用了前面用 `DataTemplate` 定义的资源 `listTemplate`。把 `IsSynchronizedWithCurrentItem` 属性设置为 `True`，从而使位于同一个窗口中的文本框和按钮控件绑定到列表框中当前选择的项上。

```
<ListBox Grid.Row="0" Grid.Column="0" HorizontalAlignment="Left"
    Name="listBoxServices" VerticalAlignment="Top"
    ItemsSource="{Binding}"
    ItemTemplate="{StaticResource listTemplate}"
    IsSynchronizedWithCurrentItem="True">
</ListBox>
```

对于文本框控件，`Text` 属性绑定到 `ServiceControllerInfo` 实例的对应属性上。按钮控件是启用还是禁用也从数据绑定中定义，即把 `IsEnabled` 属性绑定到 `ServiceControllerInfo` 实例的对应属性上，该属性返回一个布尔值：

```
<TextBlock Grid.Row="0" Grid.ColumnSpan="2" Name="textDisplayName"
    Text="{Binding Path=DisplayName, Mode=OneTime}" />
<TextBlock Grid.Row="1" Grid.ColumnSpan="2" Name="textStatus"
    Text="{Binding Path=ServiceStatusName, Mode=OneTime}" />
<TextBlock Grid.Row="2" Grid.ColumnSpan="2" Name="textType"
    Text="{Binding Path=ServiceTypeName, Mode=OneTime}" />
<TextBlock Grid.Row="3" Grid.ColumnSpan="2" Name="textName"
    Text="{Binding Path=ServiceName, Mode=OneTime}" />
<Button Grid.Row="4" Grid.Column="0" Name="buttonStart" Content="Start"
    IsEnabled="{Binding Path=EnableStart, Mode=OneTime}"
    Click="OnServiceCommand" />
<Button Grid.Row="4" Grid.Column="1" Name="buttonStop" Content="Stop"
    IsEnabled="{Binding Path=EnableStop, Mode=OneTime}"
    Click="OnServiceCommand" />
<Button Grid.Row="5" Grid.Column="0" Name="buttonPause" Content="Pause"
    IsEnabled="{Binding Path=EnablePause, Mode=OneTime}"
    Click="OnServiceCommand" />
<Button Grid.Row="5" Grid.Column="1" Name="buttonContinue"
    Content="Continue" IsEnabled="{Binding Path=EnableContinue,
    Mode=OneTime}" Click="OnServiceCommand" />
<Button Grid.Row="6" Grid.Column="0" Name="buttonRefresh" Content="Refresh"
    Click="OnRefresh" />
<Button Grid.Row="6" Grid.Column="1" Name="buttonExit"
    Content="Exit" Click="OnExit" />
```

2. 服务的控制

使用 `ServiceController` 类，也可以把控制请求发送给服务，该类的方法如表 25-5 所示。

表 25-5

方 法	说 明
<code>Start()</code>	<code>Start()</code> 方法告诉 SCM 应启动服务。在服务程序示例中，调用了 <code>OnStart()</code> 方法
<code>Stop()</code>	如果 <code>CanStop</code> 属性在服务类中的值是 <code>true</code> ，则在 SCM 的帮助下， <code>Stop()</code> 方法调用服务程序中的 <code>OnStop()</code> 方法
<code>Pause()</code>	如果 <code>CanPauseAndContinue</code> 属性的值是 <code>true</code> ，则 <code>Pause()</code> 方法调用 <code>OnPause()</code> 方法
<code>Continue()</code>	如果 <code>CanPauseAndContinue</code> 属性的值是 <code>true</code> ，则 <code>Continue()</code> 方法调用 <code>OnContinue()</code> 方法
<code>ExecuteCommand()</code>	使用 <code>ExecuteCommand()</code> 可以把定制的命令发送给服务

下面就是控制服务的代码。因为启动、停止、挂起和暂停服务的代码是相似的，所以仅为这 4 个按钮使用一个处理程序：



可从
wrox.com
下载源代码

```
protected void OnServiceCommand(object sender, RoutedEventArgs e)
{
    Cursor oldCursor = this.Cursor;
    try
    {
        this.Cursor = Cursors.Wait;
        ServiceControllerInfo si =
            (ServiceControllerInfo)listBoxServices.SelectedItem;
        if (sender == this.buttonStart)
        {
            si.Controller.Start();
            si.Controller.WaitForStatus(ServiceControllerStatus.Running,
                TimeSpan.FromSeconds(10));
        }
        else if (sender == this.buttonStop)
        {
            si.Controller.Stop();
            si.Controller.WaitForStatus(ServiceControllerStatus.Stopped,
                TimeSpan.FromSeconds(10));
        }
        else if (sender == this.buttonPause)
        {
            si.Controller.Pause();
            si.Controller.WaitForStatus(ServiceControllerStatus.Paused,
                TimeSpan.FromSeconds(10));
        }
        else if (sender == this.buttonContinue)
        {
            si.Controller.Continue();
            si.Controller.WaitForStatus(ServiceControllerStatus.Running,
                TimeSpan.FromSeconds(10));
        }
    }
    int index = listBoxServices.SelectedIndex;
```

```

        RefreshServiceList();
        listBoxServices.SelectedIndex = index;
    }
    catch (System.ServiceProcess.TimeoutException ex)
    {
        MessageBox.Show(ex.Message, "Timout Service Controller",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
    catch (InvalidOperationException ex)
    {
        MessageBox.Show(String.Format("{0} {1}", ex.Message,
            ex.InnerException != null ? ex.InnerException.Message :
                String.Empty),
            MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
    finally
    {
        this.Cursor = oldCursor;
    }
}

protected void OnExit(object sender, RoutedEventArgs e)
{
    Application.Current.Shutdown();
}

protected void OnRefresh_Click(object sender, RoutedEventArgs e)
{
    RefreshServiceList();
}

```

代码段 ServiceControl/ServiceControlWindow.xaml.cs

由于控制服务要花费一定的时间, 因此, 光标在第一条语句中切换为等待光标。然后, 根据被按的按钮调用 `ServiceController` 类的方法。使用 `WaitForStatus()` 方法, 表明用户正在等待检查服务把状态改为被请求的值, 但是, 我们最多等待 10 秒。在 10 秒之后, 就会刷新列表框中的信息, 其目的是当用户选择与以前相同的服务时, 能够显示服务的新状态。

因为应用程序需要管理权限, 大多数服务都需要管理权限来启动和停止, 所以把一个应用程序清单添加到项目中, 并把 `requestedExecutionLevel` 属性设置为 `requireAdministrator`。



可从
wrox.com
下载源代码

```

<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0"
    xmlns="urn:schemas-microsoft-com:asm.v1"
    xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
    xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
      </requestedPrivileges>
    </security>
  </trustInfo>

```

</asmv1:assembly>

代码段 ServiceControl/app.manifest

最后，运行应用程序的结果如图 25-14 所示。

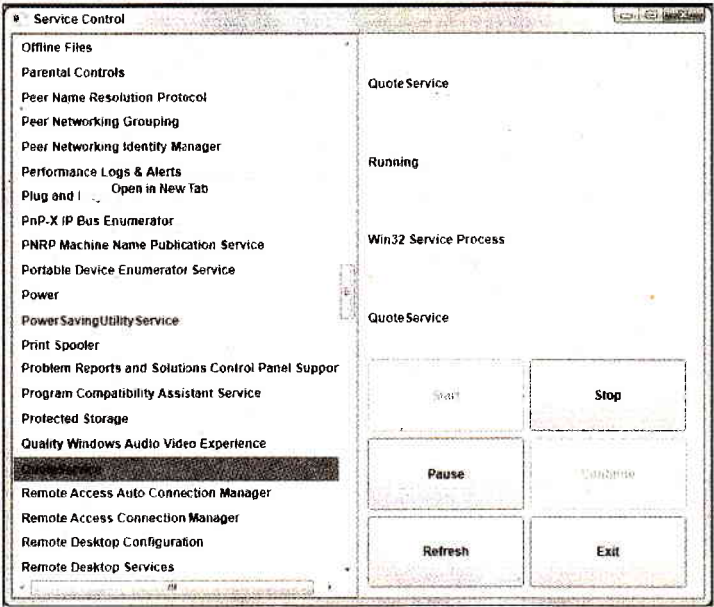


图 25-14

25.5 故障排除和事件日志

服务方面的故障排除与标准应用程序的故障排除并不相同。本节将讨论一些服务问题、交互式服务的问题和事件日志。

创建服务最好的方式就是在实际创建服务之前，先创建一个具有所需功能的程序集和一个测试客户，以便进行正常的调试和错误处理。应用程序一运行，就可以使用那个程序集创建服务。当然，对于服务，仍然存在下列问题：

- 在服务中，错误信息不显示在消息框中(除了运行在客户端系统上的交互式服务之外)，而是使用事件日志服务把错误写入事件日志中。当然，在使用服务的客户端应用程序中，可以显示一个消息框，以通知用户出现了错误。
- 虽然服务不能从调试器中启动，但是调试器可以与正在运行的服务联系起来。打开带有服务源代码的解决方案，并且设置断点。从 Visual Studio 的 Debug 菜单中选择 Processes 命令，捕获正在运行的服务进程。
- 性能监视器可以用于监视服务的行为。可以把自己的性能对象添加到服务中，这样可以添加一些有用的信息，以便进行调试。例如，通过 Quote 服务，可以建立一个对象，让它给出返回的引用总数和初始化花费的时间等。

把事件添加到事件日志中，服务就可以报告错误和其他信息。当 AutoLog 属性设置为 true 时，从 ServiceBase 类中派生的服务类可以自动把事件写入到日志中。ServiceBase 类检查 AutoLog 属性，并

且在启动、停止、暂停和继续请求时编写日志条目。

图 25-15 是服务中的一个日志条目示例。

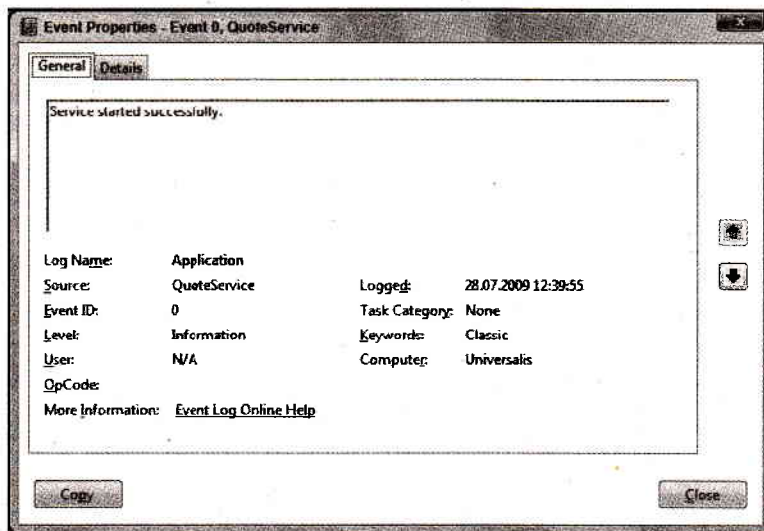


图 25-15



事件日志和如何编写自定义事件的内容详见第 19 章。

25.6 小结

本章讨论了 Windows 服务的体系结构和如何使用 .NET Framework 创建 Windows 服务。应用程序可以与 Windows 服务一起在系统启动时自动启动，也可以把具有特权的 System 账户用作服务的用户。Windows 服务从主函数、service-main 函数和处理程序中创建，本章还介绍了与 Windows 服务相关的其他程序，如服务控制程序和服务安装程序。

.NET Framework 对 Windows 服务提供了很好的支持。创建、控制和安装服务所需的代码都封装在 System.ServiceProcess 名称空间的 .NET Framework 类中。从 ServiceBase 类中派生一个类，就可以重写服务暂停、继续或停止时调用的方法。对于服务的安装，ServiceProcessInstaller 类和 ServiceInstaller 类可以处理服务所需的所有注册表配置。还可以使用 ServiceController 类控制和监视服务。

下一章介绍与本机代码的交互操作。许多 .NET 类在后台使用本地代码。例如，ServiceBase 类封装了 Windows API CreateService()。下一章还会探讨如何在自己的类中使用本地方法和 COM 对象。

第 26 章

互 操 作 性

本章内容:

- COM 和 .NET 技术
- 在 .NET 应用程序中使用 COM 对象
- 从 COM 客户端中使用 .NET 组件
- 调用本地方法的 Platform Invoke(平台调用)

如果您在学习 .NET 之前编写过 Windows 程序,通常就没有时间和资源用 .NET 再重写以前的程序。有时重写代码有助于重构,或者重新思考应用程序的体系架构。从长远来看,重写还有助于提高效率,更便于用新技术添加新特性。但是,我们不会因为一种新技术可用而重写已有的代码。我们本来有数千行已存在的可运行代码,重写它们需要的精力太多,还不如把它们迁移到托管环境中。

这也同样适用于 Microsoft。在 System.DirectoryService 名称空间中,Microsoft 并没有重写 COM 对象来访问层次结构的数据存储器,这个名称空间中的类实际上是访问 ADSI COM 对象的包装器。System.Data.OleDb 名称空间也是这样,由这个名称空间中的类所使用的 OLE DB 提供程序的确包含相当复杂的 COM 接口。

我们自己的解决方案也会面临相同的问题。如果在 .NET 应用程序中要使用已有的 COM 对象,或者倒过来,要编写在旧 COM 客户端中使用的 .NET 组件,就应使用本章介绍的 COM 互操作性。

如果没有要与应用程序集成的 COM 组件,或旧 COM 客户端要使用一些 .NET 组件,就应跳过本章。

与其他章节一样,本章的示例代码也可以从 Wrox 网站 www.wrox.com 或者本书附赠光盘上找到。

26.1 .NET 和 COM

COM 是 .NET 以前的技术。COM 定义了一个组件模型,在该模型中,组件可以用不同的编程语言编写。用 C++ 编写的组件可以在 VB 客户端中使用。组件还可以在本地的进程中使用,跨进程使用或跨网络使用。看起来是不是很熟悉?当然,.NET 的目标也是这样。但这些目标的实现方式不同。COM 概念使用起来越来越复杂,且已经不能扩展了。.NET 实现了与 COM 类似的目标,但引入了新概念,实现起来更容易。

即使到了今天,使用 COM 和 .NET 交互操作的主要问题是要理解 COM。是 COM 客户端使

用.NET 组件, 还是.NET 应用程序使用 COM 组件并不重要, 而是必须理解 COM。所以这里首先比较 COM 和.NET 的功能。

如果您已经熟练掌握了 COM 技术, 本节就复习 COM 知识。否则, 您将学习 COM 的概念——现在使用.NET——我们不再需要再在日常工作中处理它。但是, 在把 COM 技术集成到.NET 应用程序中时, COM 的问题仍旧存在。

COM 和.NET 有许多类似的概念和使用它们的不同解决方案。下面将讨论:

- 元数据
- 释放内存
- 接口
- 方法绑定
- 数据类型
- 注册
- 线程
- 错误处理
- 事件处理

下面几节讨论这些概念和编组机制。

26.1.1 元数据

在 COM 中, 组件的所有信息都存储在类型库中。类型库包含的信息有接口、方法和参数的名称和 ID 等。而在.NET 中, 所有这些信息都可以存储在程序集中, 如第 14 章和第 18 章所述。COM 存在的问题是, 类型库是不能扩展的。在 C++ 中, IDL(Interface Definition Language, 接口定义语言) 文件用于描述接口和方法。其中一些 IDL 修饰符不在类型库中, 因为 Visual Basic(和负责开发类型库的 Visual Basic 小组) 不能使用这些 IDL 修饰符。而在.NET 中, 不存在这个问题, 因为.NET 元数据可以使用自定义特性来扩展。

因此, 一些 COM 组件有类型库, 而其他 COM 组件没有。如果没有类型库可用, 就可以使用 C++ 头文件来描述接口和方法。在.NET 中, 使用带有类型库的 COM 组件比较容易, 也可使用不带类型库的 COM 组件。在这种情况下, 必须使用 C# 代码重新定义 COM 接口。

26.1.2 释放内存

在.NET 中, 内存的释放由垃圾收集器完成。这完全不同于 COM, COM 依赖的是引用计数。

IUnknown 接口是每个 COM 对象必须实现的一个接口, 它提供了 3 个方法。其中两个方法与引用计数有关。如果需要另一个接口指针, 客户端就必须调用 AddRef() 方法, 这个方法会递增引用计数。Release() 方法会递减引用计数, 如果所得的引用计数是 0, 就销毁对象, 释放内存。

26.1.3 接口

接口是 COM 的核心, 它区分了在客户端和对象之间使用的契约和实现方式。接口(协定)定义了由组件提供的方法, 可以由客户端使用。而在.NET 中, 接口也有非常重要的作用。

COM 区分 3 种接口类型: 自定义接口、调度接口(dispatch interface)和双重接口。

1. 自定义接口

自定义接口派生自 IUnknown 接口。因为自定义接口定义了虚拟表(vtable)中方法的顺序,所以客户端可以直接访问接口的方法。这也表示在开发阶段客户端需要能识别虚拟表,因为方法的绑定使用内存地址进行。因此,自定义接口不能由脚本客户端使用。图 26-1 显示了自定义接口 IMath 的虚拟表,除了接口 IUnknown 接口的方法之外,该接口还提供了 Add()和 Sub()方法。

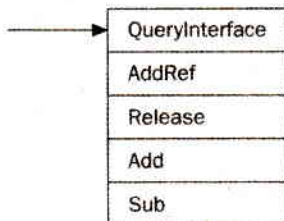


图 26-1

2. 调度接口

因为脚本客户端(和早期的 Visual Basic 客户端)不支持自定义接口,所以需要另外一种接口类型,而在调度接口中,可用于客户端可用的接口总是 IDispatch 接口。IDispatch 接口派生自 IUnknown 接口,除了接口 IUnknown 接口的方法之外,它还提供了 4 个方法,其中两个最重要的方法是 GetIDsOfNames()和 Invoke()。如图 26-2 所示,在调度接口中需要两个表。第一个表把方法名或属性名映射到调度 ID(dispatch ID)上,第二个表把调度 ID 映射到方法或属性的实现代码上。

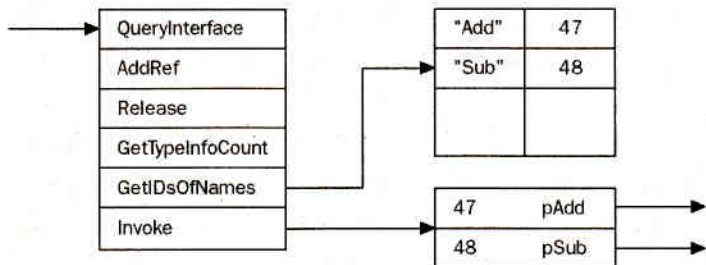


图 26-2

在客户端调用组件中的方法时,它先调用 GetIDsOfNames()方法,并给它传递要调用的方法的名称。GetIDsOfNames()方法会查找名称-ID 表,返回调度 ID,客户端再使用这个 ID 调用 Invoke()方法。



通常, IDispatch 接口的两个表存储在类型库中,但这不是必需的,一些组件把这两个表存储在其他地方。

3. 双重接口

可以想像,调度接口比自定义接口慢得多。另一方面,脚本客户端不能使用自定义接口。双重接口可以解决这个问题。如图 26-3 所示,双重接口派生自 IDispatch 接口,但提供了可以在虚拟表

中直接使用的接口的方法。脚本客户端可以使用 IDispatch 接口调用 Add()和 Sub()方法，而能够识别虚拟表的客户端可以直接调用 Add()和 Sub()方法。

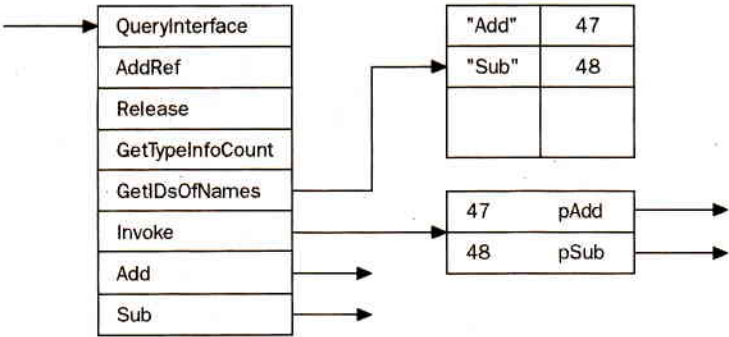


图 26-3

4. 类型强制转换和 QueryInterface

如果.NET 类实现多个接口，就可以进行类型强制转换，得到一个接口或另一个接口。而在 COM 中，IUnknown 接口通过 QueryInterface()方法提供了类似的机制。如上一节所述，因为 IUnknown 接口是其他接口的基接口，所以可以以任何方式使用 QueryInterface()方法。

26.1.4 方法的绑定

客户端映射方法的方式用术语早期绑定和后期绑定来定义。后期绑定表示要调用的方法是在运行期间确定的。.NET 使用 System.Reflection 名称空间来实现后期绑定(参阅第 14 章)。

COM 使用上面讨论的 IDispatch 接口进行后期绑定。后期绑定可以使用调度接口和双重接口来实现。

在 COM 中，早期绑定有两个不同的选项。早期绑定的一种方式也称为虚拟表绑定，它直接使用虚拟表，这可以通过自定义接口和双重接口来实现。早期绑定的第二种方式也称为 ID 绑定。其中调度 ID 存储在客户端代码中，在运行期间只需要调用一次 Invoke()方法。GetIDsOfNames()方法在设计期间调用。对于这种客户端，记住不必改变调度 ID 非常重要。

26.1.5 数据类型

对于双重接口和调度接口，COM 能使用的数据类型局限于一个自动兼容的数据类型列表。IDispatch 接口的 Invoke()方法接受 VARIANT 数据类型的数组。VARIANT 是许多不同数据类型的联合，如 BYTE、SHORT、LONG、FLOAT、DOUBLE、BSTR、IUnknown*、IDispatch*等。VARIANT 类型在 Visual Basic 中很容易使用，但它在 C++中使用时就比较复杂。在.NET 中，使用 Object 类代替 VARIANT 类型。

在自定义接口中，C++能使用的所有数据类型也可用于 COM。但是，使用这个组件的客户端只能采用某些编程语言。

26.1.6 注册

.NET 区分私有程序集和共享程序集，详见第 18 章。而在 COM 中，通过注册表配置的所有组

件都是全局可用的。

所有 COM 对象都有一个唯一的标识符, 该标识符由一个 128 位的数字组成, 也称为类 ID(CLSID)。创建 COM 对象时, COM API 调用 `CoCreateInstance()` 方法, 仅在注册表中查找 CLSID 和到 DLL 或 EXE 的路径, 然后加载 DLL 或启动 EXE, 并实例化组件。

因为这个 128 位的数字不容易记忆, 所以许多 COM 对象还有一个 ProgID, 该 ID 很容易记忆, 如 `Excel.Application` 就映射到一个 CLSID 上。

除了 CLSID 之外, COM 对象还为每个接口和类型库指定了一个唯一的标识符(分别为 IID 和 typelib id)。

本章后面将详细讨论注册表中的信息。

26.1.7 线程

COM 使用单元模型, 这样程序员就不必考虑线程问题。但是, 这也增加了复杂性。对于不同的操作系统版本添加了不同的单元类型。本节讨论单线程单元和多线程单元。



.NET 中的线程详见第 20 章。

1. 单线程单元

单线程单元(STA)是在 Windows NT 3.51 中引入的。在 STA 中, 只允许一个线程(创建实例的线程)访问组件。但是, 在一个进程中允许多个 STA, 如图 26-4 所示。

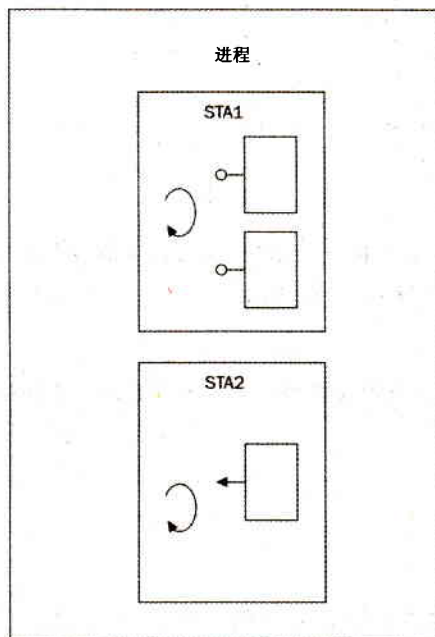


图 26-4

在图 26-4 中, 里面带棒棒糖的矩形表示 COM 组件。组件和线程(弯曲箭头)包含在单元中。外

部的矩形表示一个进程。

在 STA 中,不需要防止多个线程访问实例变量,因为这种保护由 COM 设备实现,只有一个线程可以访问组件。

COM 对象在编程时不是线程安全的,因此 STA 需要在注册表中把注册键 `ThreadingModel` 设置为 `Apartment`。

2. 多线程单元

Windows NT 4.0 引入了多线程单元(MTA)的概念。在 MTA 中,多个线程可以同时访问组件。图 26-5 显示了带一个 MTA 和两个 STA 的进程。

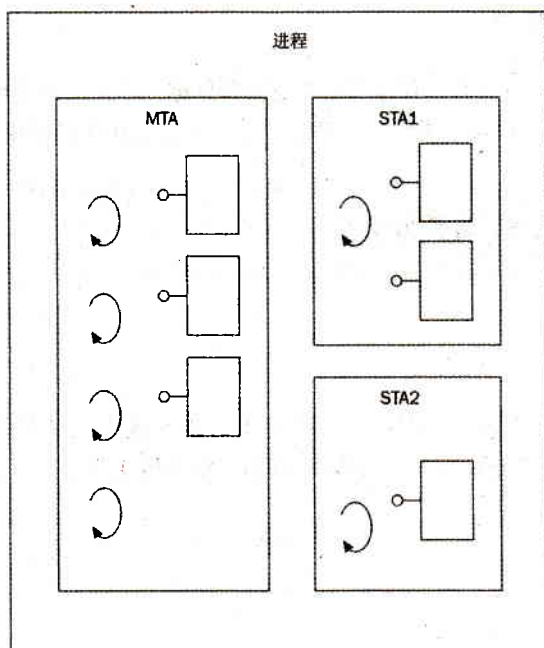


图 26-5

COM 对象在编程时是线程安全的,因此 MTA 需要在注册表中把 `ThreadingModel` 键设置为 `Free`。`Both` 值用于不考虑单元类型的线程安全的 COM 对象。



Visual Basic 6.0 不支持多线程单元。如果使用以 VB6 开发的 COM 对象,就必须了解这一点。

26.1.8 错误处理

在 .NET 中,通过抛出异常来生成错误。在较旧的 COM 技术中,通过方法返回 `HRESULT` 值来定义错误。`HRESULT` 的值是 `S_OK`, 它表示方法成功。

如果 COM 组件提供了详细的错误消息,COM 组件就实现 `ISupportErrorInfo` 接口,该接口不但

提供了错误消息,还提供了帮助文件的链接、错误源,一返回方法就会返回一个错误信息对象。在.NET中,实现 `ISupportErrorInfo` 接口的对象会自动映射到详细的错误信息和一个.NET 异常。



如何跟踪和记录错误的内容详见第19章。

26.1.9 事件

.NET 用 C# 关键字 `event` 和 `delegate` 提供了回调机制(参阅第8章)。

图 26-6 显示了 COM 的事件处理体系结构。在 COM 事件中,组件必须实现 `IConnectionPointContainer` 接口和一个或多个实现 `IConnectionPoint` 接口的连接点对象(CPO)。在图 26-6 中,组件还定义了一个由 CPO 调用的输出接口 `ICompletedEvents`。客户端必须在 sink 对象中实现这个输出接口,而 sink 对象本身是一个 COM 对象。在运行过程中,客户端在服务器中查询 `IConnectionPointContainer` 接口。通过这个接口,客户端让 CPO 通过 `FindConnectionPoint()` 方法,获得指向所返回的 `IConnectionPoint` 接口的指针。客户端再使用这个接口指针调用 `Advise()` 方法,并把指向 sink 对象的指针传递给服务器。接着,组件就可以在客户端的 sink 对象中调用方法。

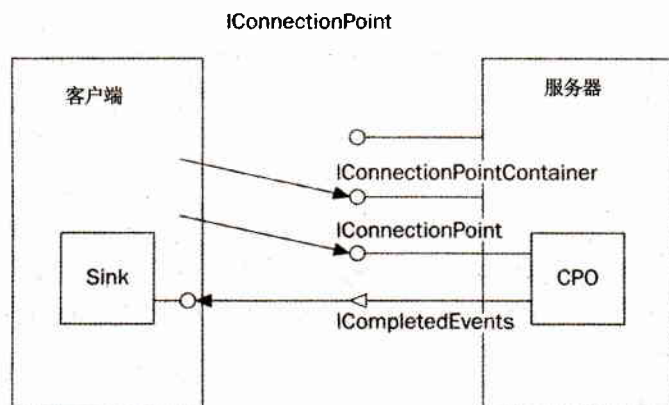


图 26-6

本章后面将讨论如何映射.NET 事件和 COM 事件,以便.NET 客户端处理 COM 事件,COM 事件处理.NET 端。

26.2 编组

从.NET 传递给 COM 组件和从 COM 组件传递给.NET 的数据必须转换为相应的表示法,这种机制也称为编组(marshaling)。具体的转换过程取决于所传递数据的数据类型。这里必须区分 blittable 和 nonblittable 数据类型。

blittable 数据类型在.NET 和 COM 中有相同的表示法,不需要转换。简单的数据类型如 `byte`、`short`、`int` 和 `long`,仅包含这些简单数据类型的类和数组都属于 blittable 数据类型。blittable 类型的数组必须是一维的。

nonblittable 数据类型需要进行转换。表 26-1 列出了一些 nonblittable 的 COM 数据类型及其对应的 .NET 数据类型。因为 nonblittable 数据类型需要转换，所以需要更高的系统开销。

表 26-1

COM 数据类型	.NET 数据类型
SAFEARRAY	Array
VARIANT	Object
BSTR	String
IUnknown*, IDispatch*	Object

26.3 从.NET 客户端中使用 COM 组件

要理解 .NET 应用程序如何使用 COM 组件，首先必须创建 COM 组件。创建 COM 组件不能使用 C# 或 Visual Basic 2010，而应使用 Visual Basic 6.0 或 C++ (或其他支持 COM 的语言)。本章使用 ATL (Active Template Library, 活动模板库)、C++ 和 Visual Studio 2010。

这里先创建一个简单的 COM 组件，在一个 RCW (Runtime Callable Wrapper, 运行时可调用包装器) 中使用它。该组件还与新的 C# 4 动态语言扩展一起使用。再讨论线程问题，最后把 COM 连接点映射到 .NET 事件上。



使用 C# 或 Visual Basic 9.0 可以创建 .NET 组件，Visual Basic 10.0 和 C# 通过一个封装器就可以把该 .NET 组件用作 COM 对象，而封装器是真正的 COM 组件。封装自 COM 组件中的 .NET 组件由 .NET 客户端通过 COM 交互操作功能来使用是没有意义的。

因为本书讲述的不是 COM，所以不讨论代码的各个方面，只讨论构建示例所需要的代码。

26.3.1 创建 COM 组件

要用 ATL 和 C++ 创建 COM 组件，先新建一个 ATL 项目。选择 File | New | Project 命令后，就会在 Visual C++ Projects 组中看到 ATL Project 向导。把名称设置为 COMServer。在 Application Settings 中，选择 Dynamic Link Library，再单击 Finish 按钮。



因为一个建立步骤在注册表中注册 COM 组件，这需要管理权限，所以 Visual Studio 应以提升的模式启动，来编写 ATL COM 对象。

ATL Project 向导刚才已为服务器创建了基础代码。还需要一个 COM 对象。在 Solution Explorer 窗口中添加一个类，选择 ATL Simple Object。在打开的对话框中，为 Short name 字段输入 COMDemo。

其他字段都是自动填充的，但把接口名改为 `IWelcome`，把 `ProgID` 改为 `COMServer.COMDemo`，如图 26-7 所示。单击 `Finish` 按钮为类和接口创建占位程序代码。

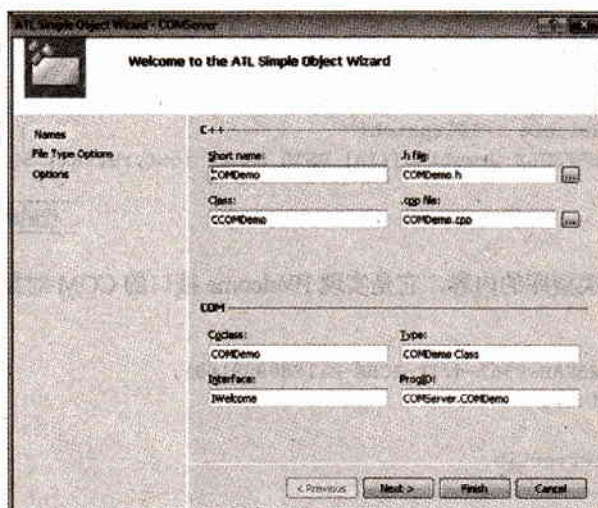


图 26-7

COM 组件提供了两个接口，以便用户可以看到 `QueryInterface()` 方法是如何从 .NET 中映射的。COM 组件还提供了 3 个简单的方法，以便我们可以看到交互操作是如何进行的。在类视图中，选择 `IWelcome` 接口，添加 `Greeting()` 方法，如图 26-8 所示，该方法有 3 个参数：

```
HRESULT Greeting([in] BSTR name, [out, retval] BSTR* message);
```

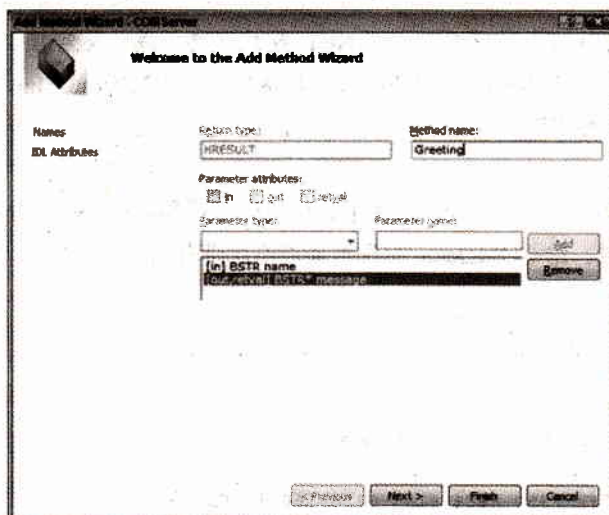


图 26-8

IDL 文件 `COMDemo.idl` 为 COM 定义了接口。向导从 `COMDemo.idl` 文件中生成的代码类似如下代码。唯一标识符 `uuid` 会有所不同。`IWelcome` 接口定义了 `Greeting()` 方法。关键字 `interface` 之前的方括号定义了接口的一些功能。`uuid` 定义接口的 ID，`dual` 标记接口的类型。



可从
wrox.com
下载源代码

```
[
    object,
    uuid(EB1E5898-4DAB-4184-92E2-BBD8F9341AFD),
    dual,
    nonextensible,
    pointer_default(unique)
]
interface IWelcome : IDispatch{
    [id(1)] HRESULT Greeting([in] BSTR name, [out,retval] BSTR* message);
};
```

代码段 COMServer/COMServer.idl

IDL 文件还定义了类型库的内容,它是实现 IWelcome 接口的 COM 对象(coclass):

```
[
    uuid(8C123EAE-F567-421F-ACBE-E11F89909160),
    version(1.0),
]
library COMServerLib
{
    importlib("stdole2.tlb");
    [
        uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4)
    ]
    coclass COMDemo
    {
        [default] interface IWelcome;
    };
};
```



在自定义特性中,可以改变由.NET 包装器类生成的类和接口的名称。只需给 custom 特性添加标识符 0F21F359-AB84-41e8-9A78-36D110E6D2F9,并给.NET 中显示的内容指定名称即可。

在 IWelcome 接口的头文件部分,添加带有相同标识符和名称 Wrox.ProCSharp.Interop.Server.IWelcome 的自定义特性。给 COMDemo 组件对象类添加带有相应名称的同一特性。

```
[
    object,
    uuid(EB1E5898-4DAB-4184-92E2-BBD8F9341AFD),
    dual,
    nonextensible,
    helpstring("IWelcome Interface"),
    pointer_default(unique),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.Interop.Server.IWelcome")
]
interface IWelcome : IDispatch{
    [id(1)] HRESULT Greeting([in] BSTR name, [out,retval] BSTR* message);
};
[
    uuid(8C123EAE-F567-421F-ACBE-E11F89909160),
```

```

        version(1.0),
    ]
    library COMServerLib
    {
        importlib("stdole2.tlb");
        [
            uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4),
            helpstring("COMDemo Class"),
            custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
                "Wrox.ProCSharp.Interop.Server.COMDemo")
        ]
        coclass COMDemo
        {
            [default] interface IWelcome;
        };
    };
};

```

现在给文件 `COMDemo.idl` 添加第二个接口。可以把 `IWelcome` 接口的头文件部分复制到新接口 `IMath` 的头文件部分，但要确保修改变用 `uuid` 关键字定义的唯一标识符。可以用实用程序 `guidgen` 生成这样一个 ID。 `IMath` 接口提供了两个方法 `Add()` 和 `Sub()`。

```

// IMath
[
    object,
    uuid(2158751B-896E-461d-9012-EF1680BE0628),
    dual,
    nonextensible,
    helpstring("IMath Interface"),
    pointer_default(unique),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.Interop.Server.IMath")
]
interface IMath: IDispatch
{
    [id(1)] HRESULT Add([in] LONG val1, [in] LONG val2,
        [out, retval] LONG* result);
    [id(2)] HRESULT Sub([in] LONG val1, [in] LONG val2,
        [out, retval] LONG* result);
};

```

`COMDemo` 组件对象类必须修改，使之实现 `IWelcome` 和 `IMath` 接口，`IWelcome` 接口是默认接口。

```

importlib("stdole2.tlb");
[
    uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4),
    helpstring("COMDemo Class"),
    custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.Interop.Server.COMDemo")
]
coclass COMDemo
{
    [default] interface IWelcome;
    interface IMath;
};

```

};

现在可以把注意力从 IDL 文件转向 C++ 代码。COMDemo.h 文件包含 COM 对象的类定义。CCOMDemo 类使用多重继承机制，以派生自模板类 CComObjectRootEx、CComCoClass 和 IDispatchImpl 接口。CComObjectRootEx 类提供 IUnknown 接口功能的实现代码，如 AddRef() 和 Release() 方法的实现。CComCoClass 类创建一个工厂，来实例化模板参数的对象，这里是 CComDemo。IDispatchImpl 提供 IDispatch 接口中方法的实现代码。

利用 BEGIN_COM_MAP 和 END_COM_MAP 中的宏，创建一个映射，定义 COM 类实现的所有 COM 接口。这个映射由 QueryInterface() 方法的实现代码使用。



可从
wrox.com
下载源代码

```
class ATL_NO_VTABLE CCOMDemo:
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
/*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
    CCOMDemo()
    {
    }
    DECLARE_REGISTRY_RESOURCEID(IDR_COMDEMO)

    BEGIN_COM_MAP(CCOMDemo)
        COM_INTERFACE_ENTRY(IWelcome)
        COM_INTERFACE_ENTRY(IDispatch)
    END_COM_MAP()

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

    void FinalRelease()
    {
    }

public:
    STDMETHOD(Greeting)(BSTR name, BSTR* message);
};

OBJECT_ENTRY_AUTO(__uuidof(COMDemo), CCOMDemo)
```

代码段 COMServer/COMDemo.h

有了这个类定义，还必须添加第二个接口 IMath，以及用 IMath 接口定义的方法：

```
class ATL_NO_VTABLE CCOMDemo:
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
/*wMajor =*/ 1, /*wMinor =*/ 0>
```

```

    public IDispatchImpl<IMath, &IID_IMath, &LIBID_COMServerLib, 1, 0>
    {
    public:
        CCOMDemo()
        {
        }

    DECLARE_REGISTRY_RESOURCEID(IDR_COMDEMO)

    BEGIN_COM_MAP(CCOMDemo)
        COM_INTERFACE_ENTRY(IWelcome)
        COM_INTERFACE_ENTRY(IMath)
        COM_INTERFACE_ENTRY2(IDispatch, IWelcome)
    END_COM_MAP()

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

    void FinalRelease()
    {
    }

    public:
        STDMETHOD(Greeting)(BSTR name, BSTR* message);
        STDMETHOD(Add)(long val1, long val2, long* result);
        STDMETHOD(Sub)(long val1, long val2, long* result);
    };

    OBJECT_ENTRY_AUTO(__uuidof(CCOMDemo), CCOMDemo)

```

现在可以在文件 `COMDemo.cpp` 中用下面的代码实现 3 个方法。`CComBSTR` 是一个很容易处理 `BSTR` 的 ATL 类。在 `Greeting()` 方法中, 只返回一条欢迎消息, 该消息把第一个参数传入的名称添加到返回的消息中。`Add()` 方法把两个值加在一起, 而 `Sub()` 方法进行减法操作, 并返回相减的结果。



可从
wrox.com
下载源代码

```

STDMETHODIMP CCOMDemo::Greeting(BSTR name, BSTR* message)
{
    CComBSTR tmp("Welcome, ");
    tmp.Append(name);
    *message = tmp;
    return S_OK;
}

STDMETHODIMP CCOMDemo::Add(LONG val1, LONG val2, LONG* result)
{
    *result = val1 + val2;
    return S_OK;
}

STDMETHODIMP CCOMDemo::Sub(LONG val1, LONG val2, LONG* result)
{
    *result = val1 - val2;
}

```

```

    return S_OK;
}

```

代码段 COMServer/COMDemo.cpp

现在就可以构建组件。构建过程也在注册表中配置组件。

26.3.2 创建 RCW

现在可以在 .NET 中使用 COM 组件。为此，必须创建一个 RCW。使用 RCW，.NET 客户端就可以使用 .NET 对象而不是 COM 组件，所以不需要处理 COM 特性，这由包装器来处理。RCW 隐藏了 IUnknown 和 IDispatch 接口(如图 26-9 所示)，并处理 COM 对象的引用计数。

RCW 可以使用命令行实用程序 `tlbimp` 或 Visual Studio 来创建。启动命令：

```
tlbimp COMServer.dll /out:Interop.COMServer.dll
```

这条命令会创建文件 `Interop.COMServer.dll`，其中包含带有包装器类的 .NET 程序集。在这个生成的程序集中，可以找到 `COMWrapper` 名称空间、`CCOMDemoCalss` 类、`CCOMDecmo`、`IMath` 和 `IWelcome` 接口。名称空间的名称可以使用实用程序 `tlbimp` 的选项来修改。`/namespace` 选项允许指定不同的名称空间，`/asmversion` 选项可以定义程序集的版本号。

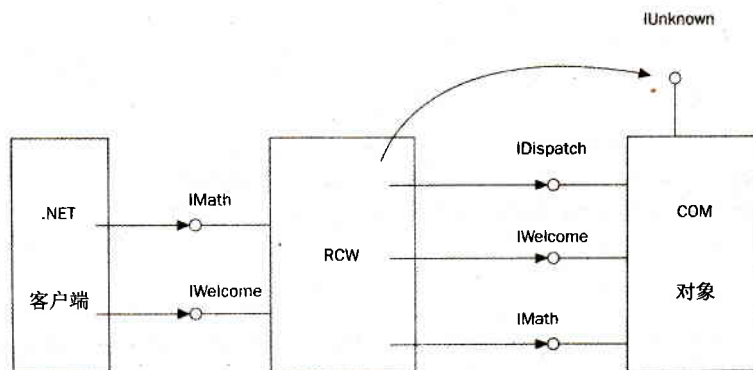


图 26-9



这个命令行实用程序的另一个重要选项是 `/keyfile`，它可以把一个强名赋予所生成的程序集。关于强名，详见第 18 章。

RCW 还可以使用 Visual Studio 来创建。要创建一个简单的示例应用程序，应创建一个 C# 控制台项目。在 Solution Explorer 窗口中，选择 Add Reference 对话框中的 COM 选项卡，向下滚动到 COMServer 1.0 Type Library 选项上，可以添加对 COM 服务器的引用。这里列出了在注册表中配置的所有 COM 对象。从列表选择一个 COM 组件，创建一个 RCW 类。在 Visual Studio 2010 中，把 Embed Interop Types 属性设置为默认值 `true`，就可以在项目的主程序集中创建这个包装器类。把该属性设置为 `false`，会创建一个部署应用程序时需要的单独的交互操作程序集。

26.3.3 使用 RCW

创建包装器类后, 就可以为应用程序编写代码来实例化和访问组件。因为在 C++ 文件中有自定义特性, 所以 RCW 类生成的名称空间是 `Wrox.ProCSharp.COMInterop.Server`。在上面的声明中添加这个名称空间和 `System.Runtime.InteropServices` 名称空间。在 `System.Runtime.InteropServices` 名称空间中, 使用 `Marshal` 类可以释放 COM 对象。



可从
wrox.com
下载源代码

```
using System;
using System.Runtime.InteropServices;
using Wrox.ProCSharp.Interop.Server

namespace Wrox.ProCSharp.Interop.Client
{
    class Program
    {
        [STAThread]
        static void Main()
        {
```

代码段 DotnetClient/Program.cs

现在可以像使用 .NET 类那样使用 COM 组件。obj 是 `COMDemo` 类型的一个变量, `COMDemo` 是一个 .NET 接口, 它提供了 `IWelcome` 和 `IMath` 接口的方法。还可以对特定的接口进行数据类型强制转换, 如 `IWelcome` 接口。通过声明为 `IWelcome` 类型的变量, 可以调用 `Greeting()` 方法。



尽管 `COMDemo` 是一个接口, 但可以实例化 `COMDemo` 类型的新对象。与一般的接口不同, 可以对封装的 COM 接口进行这类操作。

```
var obj = new COMDemo();
IWelcome welcome = obj;
Console.WriteLine(welcome.Greeting("Stephanie"));
```

如果对象(如本例所示)提供了多个接口, 就可以声明其他接口的变量, 通过使用简单的赋值语句和类型强制转换运算符, 包装器类就可以通过 `QueryInterface()` 方法和 COM 对象返回第二个接口指针。使用 `IMath` 变量可以调用 `IMath` 接口的方法。

```
IMath math;
math = (IMath)welcome;
int x = math.Add(4, 5);
Console.WriteLine(x);
```

如果在垃圾收集器清理对象之前释放 COM 对象, 静态方法 `Marshal.ReleaseComObject()` 就调用组件的 `Release()` 方法, 这样组件就可以销毁它自己, 并释放内存。

```
Marshal.ReleaseComObject(math);
```



前面提到，一旦引用计数为 0，就释放 COM 对象。Marshal.ReleaseComObject() 方法会调用 Release() 方法给引用计数递减 1。因为 RCW 仅调用一次 AddRef() 方法，来递增引用计数，所以无论引用 RCM 多少次，调用一次 Marshal.ReleaseComObject() 方法，就足以释放对象了。

在使用 Marshal.ReleaseComObject() 方法释放 COM 对象后，就不能使用引用该对象的变量了。在本例中，使用 math 变量释放 COM 对象。welcome 变量也引用 COM 对象，它不能在释放对象后使用。否则，就会生成一个 InvalidComObjectException 类型的异常。



COM 对象在不再需要时就释放，这很重要。COM 对象使用本地内存堆，而 .NET 对象使用托管的内存堆。垃圾收集器只处理托管内存。

可以看出，有了 RCW，就可以像使用 .NET 对象那样来使用 COM 组件。

26.3.4 使用 COM 服务器和动态语言扩展

C# 4 包含一个扩展功能，可从 C# 中使用动态语言。这也是使用提供了 IDispatch 接口的 COM 服务器的一个优点。如 26.1.3 节所述，这个接口在运行期间使用 GetIdsOfNames() 和 Invoke() 方法解析。利用 dynamic 关键字和后台使用的 COM 绑定器，可以在不创建 RCW 对象的情况下调用 COM 组件。

声明一个 dynamic 类型的变量，并给它赋予一个 COM 对象，以使用 COM 绑定器，接着就可以调用默认接口的方法。要在不使用 RCW 的情况下创建 COM 对象的示例，可以使用 Type.GetTypeFromProgID() 方法获取 Type 对象，再通过 Activator.CreateInstance() 方法实例化 COM 对象。通过 dynamic 关键字不能得到 IntelliSense，但可以使用 COM 非常常见的可选参数：



可从
wrox.com
下载源代码

```
using System;

namespace Wrox.ProCSharp.Interop
{
    class Program
    {
        static void Main()
        {
            Type t = Type.GetTypeFromProgID("COMServer.COMDemo");
            dynamic o = Activator.CreateInstance(t);
            Console.WriteLine(o.Greeting("Angela"));
        }
    }
}
```

代码段 DynamicDotnetClient/Program.cs



C# 的动态语言扩展参见第 12 章。

26.3.5 线程问题

如本章前面所述, COM 组件根据实现的线程安全与否, 标记它所在的单元(STA 或 MTA)。但是, 线程必须加入单元中。添加线程的单元可以用[STAThread]和[MTAThread]特性定义, 这两个特性可以应用于应用程序的 Main()方法。[STAThread]特性表示线程加入 STA。而[MTAThread]特性表示线程加入 MTA。如果没有应用特性, 默认情况下就加入 MTA。

还可以使用 Thread 类的 ApartmentState 特性编程设置单元状态。ApartmentState 特性允许设置 ApartmentState 枚举中的一个值。ApartmentState 枚举的值有 STA 和 MTA(如果没有设置, 就使用 Unknown)。注意线程的单元状态只能设置一次。如果第二次设置它, 就会忽略第二次的设置。



如果线程选择了组件所不支持的单元, 该怎么办? COM 运行库会自动创建 COM 组件的正确单元。但是, 如果在调用组件的方法时跨越了单元边界, 性能就会降低。

26.3.6 添加连接点

为了解 COM 事件在 .NET 应用程序中的处理方式, 首先必须扩展 COM 组件。首先必须在接口定义文件 COMDemo.idl 中添加另一个接口。_ICompletedEvents 接口由客户端(.NET 应用程序)实现, 并由组件调用。在本例中, 在完成计算后, Completes()方法由组件调用。这个接口也称为输出接口。输出接口必须是委托接口或自定义接口。所有的客户端都支持委托接口。ID 为 0F21F359-AB84-41e8-9A78-36D110E6D2F9 的自定义特性定义了 RCW 中创建的接口的名称。输出接口还必须在 coclass 部分中写入组件支持的接口, 并把它标记为 source 接口。



可从
wrox.com
下载源代码

```
library COMServerLib
{
    importlib("stdole2.tlb");

    [
        uuid(5CFF102B-0961-4EC6-8BB4-759A3AB6EF48),
        helpstring("_ICompletedEvents Interface"),
        custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
            "Wrox.ProCSharp.COMInterop.Server.ICompletedEvents"),
    ]
    dispinterface _ICompletedEvents
    {
        properties:
        methods:
            [id(1)] void Completed(void);
    };

    [
        uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4),
        helpstring("COMDemo Class")
        custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
            "Wrox.ProCSharp.COMInterop.Server.COMDemo"),
    ]
    coclass COMDemo
    {
        [default] interface IWelcome;
```

```

interface IMath;
[default, source] dispinterface _ICompletedEvents;
};

```

代码段 COMServer/COMServer.idl

使用向导可以创建实现代码，将事件发送回客户端。打开类视图，选择 CComDemo 类，打开上下文菜单，选择 Add | Add Connection Point 命令，启动 Implement Connection Point Wizard 对话框，如图 26-10 所示。给带连接点的实现代码选择源接口 ICompletedEvents。

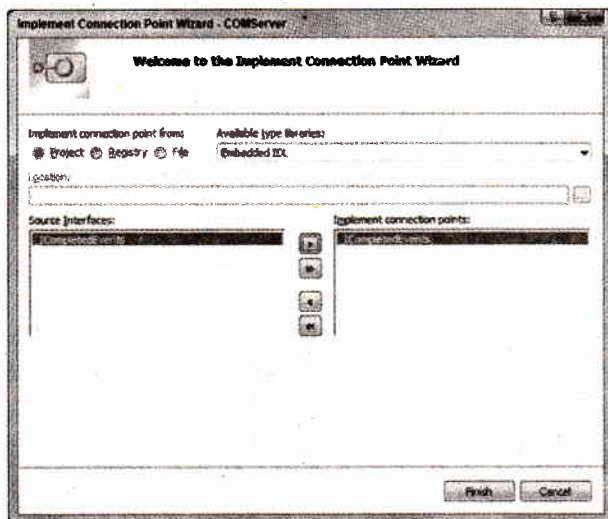


图 26-10

向导创建了代理类 CProxy_ICompletedEvents，将事件发送给客户端。另外，还修改了 CCOMDemo 类。这个类现在继承自 IConnectionPointContainerImpl 接口和代理类。把 IConnectionPointContainer 接口添加到接口映射中，把连接点映射添加到源接口 ICompletedEvents 中。



可从
wrox.com
下载源代码

```

class ATL_NO_VTABLE CCOMDemo:
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
/*wMajor =*/ 1, /*wMinor =*/ 0>,
public IDispatchImpl<IMath, &IID_IMath, &LIBID_COMServerLib, 1, 0>,
public IConnectionPointContainerImpl<CCOMDemo>,
public CProxy_ICompletedEvents<CCOMDemo>
{
public:
//.

BEGIN_COM_MAP(CCOMDemo)
COM_INTERFACE_ENTRY(IWelcome)
COM_INTERFACE_ENTRY(IMath)
COM_INTERFACE_ENTRY2(IDispatch, IWelcome)
COM_INTERFACE_ENTRY(IConnectionPointContainer)
END_COM_MAP()

//.

```

```

public:
    BEGIN_CONNECTION_POINT_MAP(CCOMDemo)
        CONNECTION_POINT_ENTRY(__uuidof(_ICompletedEvents))
    END_CONNECTION_POINT_MAP()
};

```

代码段 COMServer/COMDemo.h

最后, 在文件 COMDemo.cpp 的 Add() 和 Sub() 方法中调用代理类的 Fire_Completed() 方法。



可从
wrox.com
下载源代码

```

STDMETHODIMP CCOMDemo::Add(LONG val1, LONG val2, LONG* result)
{
    *result = val1 + val2;
    Fire_Completed();
    return S_OK;
}

STDMETHODIMP CCOMDemo::Sub(LONG val1, LONG val2, LONG* result)
{
    *result = val1 - val2;
    Fire_Completed();
    return S_OK;
}

```

代码段 COMServer/COMDemo.cpp

在重新构建 COM DLL 之后, 就可以把 .NET 客户端改为使用这些 COM 事件, 这与使用一般的 .NET 事件一样。



可从
wrox.com
下载源代码

```

static void Main()
{
    COMDemo obj = new COMDemo();

    IWelcome welcome = obj;
    Console.WriteLine(welcome.Greeting("Stephanie"));

    obj.Completed += () => Console.WriteLine("Calculation completed");

    IMath math = (IMath)welcome;
    int result = math.Add(3, 5);
    Console.WriteLine(result);

    Marshal.ReleaseComObject(math);
}

```

代码段 DotnetClient/Program.cs

可以看出, RCW 提供了从 COM 事件到 .NET 事件的自动映射。可以在 .NET 客户端中像使用 .NET 事件那样使用 COM 事件。

26.4 从 COM 客户端中使用 .NET 组件

前面讨论了如何从 .NET 客户端中访问 COM 组件。在使用 VB6、MFC 或 ATL 编写的旧 COM 客户端中寻求访问 .NET 组件的解决方案也同样十分有意义。

本节用 .NET 代码定义一个 COM 对象, COM 客户端通过 CCW(COM 可调用包装, COM.Caller

Wrapper)来使用该对象。从 COM 客户端中使用该对象,将说明如何从.NET 程序集中创建类型库,如何使用不同的.NET 特性指定 COM 交互行为,如何把.NET 程序集注册为 COM 组件。接着用 C++ 创建一个 COM 客户端,以使用 CCW。最后扩展.NET 组件,以提供 COM 连接点。

26.4.1 CCM

如果要使用.NET 客户端访问 COM 组件,就必须使用 RCW。要从 COM 客户端应用程序中访问.NET 组件,就必须使用 CCW。图 26-11 显示了包装.NET 类的 CCW,并提供了 COM 客户端希望使用的 COM 接口。CCW 提供了 IUnknown、IDispatch 及其他接口,它还为事件提供了 IConnectionPointContainer 和 IConnectionPoint 接口。当然,CCW 还提供了.NET 类定义的自定义接口,如 IWelcome 和 IMath 接口。COM 客户端可以从 COM 对象中得到它需要的信息,尽管.NET 组件在后台发挥作用。包装器处理 IUnknown 接口中的 AddRef()、Release()、QueryInterface()方法。而在.NET 对象中,可以依赖垃圾收集器,不需要处理引用计数。

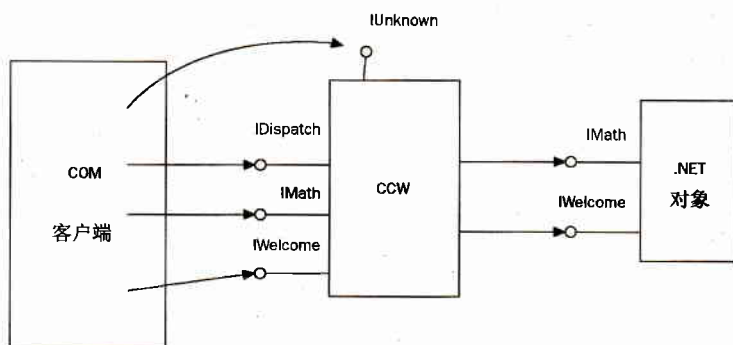


图 26-11

26.4.2 创建.NET 组件

在下面的示例中,要在.NET 类中构建与 COM 组件相同的功能。首先创建一个 C#类库,命名它为 DotNetServer。接着添加 IWelcome 和 IMath 接口,以及实现这些接口的 DotNetComponent 类。`[ComVisible(true)]`特性使类和接口可用于 COM:



可从
wrox.com
下载源代码

```

using System;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.Interop.Server
{
    [ComVisible(true)]
    public interface IWelcome
    {
        string Greeting(string name);
    }

    [ComVisible(true)]
    public interface IMath
    {
        int Add(int val1, int val2);
    }
}
  
```

```

        int Sub(int val1, int val2);
    }

    [ComVisible(true)]
    public class DotnetComponent: IWelcome, IMath
    {
        public DotnetComponent()
        {
        }

        public string Greeting(string name)
        {
            return "Hello " + name;
        }

        public int Add(int val1, int val2)
        {
            return val1 + val2;
        }

        public int Sub(int val1, int val2)
        {
            return val1 - val2;
        }
    }
}

```

代码段 DotnetServer/DotnetServer.cs

构建了项目后，就可以创建类型库了。

26.4.3 创建类型库

类型库可以用命令行实用程序 `tlbexp` 创建。命令：

```
tlbexp DotnetServer.dll
```

会创建类型库 `DotNetComponent.tlb`。使用实用程序 OLE/COM 对象查看器 `oleview32.exe` 可以查看类型库，该工具包含在 Microsoft SDK 中。可以在 Visual Studio 2010 的命令提示符中访问这个实用程序，选择 `File | View TypeLib` 命令，打开类型库。现在可以看出，接口定义非常类似于前面用 COM 服务器创建的接口。

根据程序集的名称创建类型库的名称。类型库的头文件还在自定义特性中定义了程序集的全名，所有接口都在定义之前前置声明：

```

// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: DotnetServer.tlb
[
    uuid(14F7A17A-B97D-41DA-B3E1-B6025F188FAD),
    version(1.0),
    custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, "DotnetServer, Version=1.0.0.0,

```

```

        Culture=neutral, PublicKeyToken=null")
    ]
    library DotnetServer
    {
        // TLib : // TLib : mscorlib.dll : {BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
        importlib("mscorlib.tlb");
        // TLib : OLE Automation : {00020430-0000-0000-c260-000000000046}
        importlib("stdole2.tlb");

        // Forward declare all types defined in this typelib
        interface IWelcome;
        interface IMath;
        interface _DotnetComponent;
    }

```

在下面生成的代码中，可以看到 `IWelcome` 和 `IMath` 接口被定义为 COM 双重接口。在 C# 代码中声明的所有方法都列在类型库定义中。参数有所改变：.NET 类型映射到 COM 类型(如 `String` 类映射到 `BSTR` 类型)，签名也改变了，从而返回一个 `HRESULT`。因为接口是双重接口，所以还生成了调度 ID。

```

[
    odl,
    uuid(6AE7CB9C-7471-3B6A-9E13-51C2294266F0),
    version(1.0),
    dual,
    oleautomation,
    custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.Interop.Server.IWelcome")
]
interface IWelcome : IDispatch {
    [id(0x60020000)]
    HRESULT Greeting(
        [in] BSTR name,
        [out, retval] BSTR* pRetVal);
};

[
    odl,
    uuid(AED00E6F-3A60-3EB8-B974-1556096350CB),
    version(1.0),
    dual,
    oleautomation,
    custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.Interop.Server.IMath")
]
interface IMath : IDispatch {
    [id(0x60020000)]
    HRESULT Add(
        [in] long val1,
        [in] long val2,
        [out, retval] long* pRetVal);
};

```

```
[id(0x60020001)]
HRESULT Sub(
    [in] long val1,
    [in] long val2,
    [out, retval] long* pRetVal);
};
```

coclass 部分标记了 COM 对象本身。头文件中的 **uuid** 是用于实例化 COM 对象的 CLSID。**DotNetComponent** 类支持 **_DotNetComponent**、**_Object**、**IWelcome** 和 **IMath** 接口。**_Object** 在文件 **mscorlib.tlb** 中定义, 该文件包含在前面的代码段中, 提供了基类 **Object** 的方法。组件的默认接口是 **_DotNetComponent**, 它在 **coclass** 部分的后面定义为一个调度接口。在接口声明中, 它标记为双重接口, 但因为它不包含方法, 所以它是一个调度接口。在这个接口中, 可以使用后期绑定访问组件的所有方法。

```
[
    uuid(2F1E78D4-1147-33AC-9233-C0F51121DAAA),
    version(1.0),
    custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.Interop.Server.DotnetComponent")
]
coclass DotnetComponent {
    [default] interface _DotnetComponent;
    interface _Object;
    interface IWelcome;
    interface IMath;
};

[
    odl,
    uuid(2B36C1BF-61F7-3E84-87B2-EAB52144046D),
    hidden,
    dual,
    oleautomation,
    custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.Interop.Server.DotnetComponent")
]
interface _DotnetComponent : IDispatch {
};
};
```

生成类型库有许多默认选项。其优点是可以修改一些从 .NET 到 COM 的默认映射。这可以使用 **System.Runtime.InteropServices** 名称空间中的几个特性来实现。

26.4.4 COM 互操作特性

把 **System.Runtime.InteropServices** 名称空间中的特性应用于类、接口或方法, 可以修改 CCW 的实现方式。表 26-2 列出了这些特性和描述。

表 26-2

特 性	说 明
Guid	这个特性可以赋予程序集、接口和类。把 Guid 用作程序集特性，会定义类型库 ID；把它应用于接口，会定义接口 ID(IID)；把它设置为一个类，会定义类 ID(CLSID)。使用这个特性需要定义的唯一 ID，可以用实用程序 guidgen 来创建。在每次构建程序时，都会自动修改 CLSID 和类型库 ID。如果不希望在每次构建程序时都改变它们，就可以使用这个特性来固定它们。IID 只有在接口的签名改变时才修改。例如，添加或删除了方法，或者改变了某些参数。因为在 COM 中，IID 应在每个接口的新版本中改变，所以这是很好的默认行为，通常不需要对 IID 应用 Guid 特性。对接口应用固定 IID 的唯一情况是.NET 接口是已有 COM 接口的准确表示，而且 COM 客户端希望使用这个标识符
ProgId	这个特性可以应用于类，用于指定在注册表中配置对象时使用什么名称
ComVisible	使用项目属性的 Assembly Information 设置，可以配置程序集中的所有类型是否应通过 COM 可见。这个设置默认为 false，这是很好的默认方式，必须显式地使用 ComVisible 特性标记类、接口和委托，来创建 COM 表示。如果默认设置改为通过 COM 使所有类型可见，对于不应创建 COM 表示的类型，就可以把 ComVisible 特性设置为 false
InterfaceType	如果把把这个特性设置为 COMInterfaceType 枚举值，就允许修改为 .NET 接口创建的默认双重接口类型。COMInterfaceType 枚举值有 InterfaceIsDual、InterfaceIsIDispatch 和 InterfaceIsUnknown。如果要把自定义接口类型应用于 .NET 接口，就设置这个特性： [InterfaceType(ComInterfaceType.InterfaceIsUnknown)]
ClassInterface	这个特性允许修改为类创建的默认调度接口。ClassInterface 的参数是 ClassInterfaceType 枚举的值。该枚举值有 AutoDispatch、AutoDual 和 None。在前面的示例中默认值是 AutoDispatch，因为创建了一个调度接口。如果类只能由定义的接口访问，就应给这个类应用 ClassInterface (ClassInterfaceType.None)属性
DispId	这个特性可以与双重接口和调度接口一起使用，以定义方法和属性的 DispId
In Out	如果参数应发送给组件 In，或者参数从组件发送给客户端 out，或者参数是双向的 In,out，COM 就允许把特性指定为参数类型
Optional	COM 方法的参数可以是可选的。可选参数可以用 Optional 特性标记

现在可以修改 C#代码，为 IWelcome 接口指定双重接口类型，为 IMath 接口指定自定义接口类型，用参数 ClassInterfaceType.None 为 DotNetComponent 类指定 ClassInterface 特性，不生成单独的 COM 接口。progid 和 guid 属性指定 progID 和 GUID:



```
[InterfaceType(ComInterfaceType.InterfaceIsDual)]
[ComVisible(true)]
public interface IWelcome
{
    [DispId(60040)]
    string Greeting(string name);
}

[InterfaceType(ComInterfaceType.InterfaceIsUnknown)]
[ComVisible(true)]
public interface IMath
{
```

```

    int Add(int val1, int val2);
    int Sub(int val1, int val2);
}

[ClassInterface(ClassInterfaceType.None)]
[ProgId("Wrox.DotnetComponent")]
[Guid("77839717-40DD-4876-8297-35B98A8402C7")]
[ComVisible(true)]
public class DotnetComponent: IWelcome, IMath
{
    public DotnetComponent()
    {
    }
}

```

代码段 DotnetServer/DotnetServer.cs

重新构建类库和类型库，修改接口定义。使用 OleView.exe 可以验证这一点。IWelcome 接口仍是双重接口，而 IMath 现在是一个派生自 IUnknown 接口的自定义接口，不是派生自 IDispatch 接口。在 coclass 部分中不再有 _DotNetComponent 接口。

26.4.5 COM 注册

在 .NET 组件用作 COM 对象之前，需要在注册表中配置它。另外，如果不希望把程序集复制到客户端应用程序所在的目录下，就需要把程序集安装在全局程序集缓存中。全局程序集缓存详见第 18 章。

要把程序集安装在全局程序集缓存中，必须用强名标识它(使用 Visual Studio 2010，可以在解决方案的属性中定义强名)，然后在全局程序集缓存中注册程序集：

```
gacutil -i DotnetServer.dll
```

现在可以使用 regasm 实用程序在注册表中配置组件。选项 /tlb 可以提取类型库，并在注册表中配置类型库：

```
regasm DotnetServer.dll /tlb
```

下面将列出写入注册表中的 .NET 组件信息。所有 COM 配置都在 HKEY_CLASSES_ROOT(HKCR)配置单位中。把 ProgId 的键(在本例中，它是 Wrox.DotNetComponent)和 CLSID 写入该配置单元中。

HKCR\CLSID\{CLSID}\InProcServer32 键有如下选项：

- **mscorlib.dll**——它表示 CCW。这是一个真正的 COM 对象，它负责存储 .NET 组件。这个 COM 对象访问 .NET 组件，为客户端提供 COM 作为。通过一般的 COM 实例化机制从客户端上加载和实例化 mscorlib.dll 文件。
- **ThreadingModel=Both**——这是 mscorlib.dll COM 对象的一个特性。这个组件以支持 STA 和 MTA 的方式编写。
- **Assembly=DotnetComponent, Version=1.0.0.0, Culture=neutral, PublicKeyToken=5cd57c93b4d9c41a:** Assembly 的值存储了程序集的全名，其中包括版本号和公钥标记，从而使可以唯一地标识程序集。这里注册的程序集将通过 mscorlib.dll 加载。

- `Class=Wrox.ProCSharp.COMInterop.Server.DotNetComponent`——类名也由 `mscorlib.dll` 使用。这是要实例化的类。
- `RuntimeVersion=v1.1.4322`——注册项 `RuntimeVersion` 指定要用于存储.NET 程序集的.NET 运行库的版本。

除了这里列出的配置之外，所有接口和类型库也用它们的标识符配置。

26.4.6 创建 COM 客户端应用程序

现在该创建 COM 客户端应用程序了。首先创建一个简单的 C++ Win32 控制台应用程序项目，命名它为 `COMClient`。可以在项目向导中选择默认选项，并单击 `Finish` 按钮。

在文件 `COMClient.cpp` 的开头，添加一条预处理器命令，以包含 `<iostream>` 头文件，和导入为 .NET 组件创建的类型库。导入语句创建了一个“智能指针”类，这样更容易处理 COM 对象。在构建过程中，导入语句会创建 `.tlh` 和 `.tli` 文件，它们位于项目的 `debug` 目录下，包含智能指针类。接着添加 `using namespace` 指令，打开 `std` 和 `DotNetComponent` 名称空间，`std` 名称空间用于把输出消息写入控制台中，`DotNetComponent` 名称空间在智能指针类中创建。



可从
wrox.com
下载源代码

```
// COMClient.cpp: Defines the entry point for the console application.
//
#include "stdafx.h"
#include <iostream>
#import "../DotNetComponent/bin/debug/DotnetServer.tlb" named_guids

using namespace std;
using namespace DotnetComponent;
```

代码段 COMClient/COMClient.cpp

在 `_tmain()` 方法中，在进行任何其他 COM 调用之前，要先用 API 调用 `CoInitialize()` 实例化 COM。`CoInitialize()` 方法会为线程创建 STA 并进入 STA。`spWelcome` 变量是 `IWelcome` 类型的智能指针，智能指针方法 `CreateInstance()` 的参数是 `ProgId`，它使用 COM API `CoCreateInstance()` 创建 COM 对象。用智能指针重写运算符 `->`，这样就可以调用 COM 对象的方法，如 `Greeting()`。

```
int _tmain(int argc, _TCHAR* argv[])
{
    HRESULT hr;
    hr = CoInitialize(NULL);

    try
    {
        IWelcomePtr spWelcome;

        // CoCreateInstance()
        hr = spWelcome.CreateInstance("Wrox.DotnetComponent");

        cout << spWelcome->Greeting("Bill") << endl;
    }
}
```

.NET 组件支持的第二个接口是 `IMath`，它也有一个包装 COM 接口的智能指针 `IMathPtr`。可以直接把一个智能指针赋予另一个智能指针，如 `spMath = spWelcome`；在智能指针的实现(重写了 `=` 运算符)中，调用了 `QueryInterface()` 方法。使用 `IMath` 接口的引用，可以调用 `Add()` 方法。

```

IMathPtr spMath;
spMath = spWelcome; // QueryInterface()

long result = spMath->Add(4, 5);
cout <<"result:" <<result <<endl;
}

```

如果 COM 对象返回一个 HRESULT 错误值(如果.NET 组件生成异常, 返回 HRESULT 错误的 CCW 就会返回一个错误值), 智能指针就包装 HRESULT 错误, 并生成 `_com_error` 异常。错误在 `catch` 块中处理。在程序的最后, 使用 `CoUninitialize()` 方法关闭和卸载 COM DLL。

```

catch (_com_error & e)
{
    cout <<e.ErrorMessage() <<endl;
}

CoUninitialize();
return 0;
}

```

现在可以运行应用程序, 在控制台上得到 `Greeting()` 和 `Add()` 方法的输出。还可以试着调试智能指针类, 在这里可以看到直接调用了 COM API。



如果得到一个找不到组件的异常, 就应检查是否在全局程序集缓存中安装了在注册表中配置的程序集的同版本。

26.4.7 添加连接点

在.NET 组件中添加对 COM 事件的支持, 需要对.NET 类的实现代码作一些修改。提供 COM 事件并不是简单地使用 `event` 和 `delegate` 关键字, 还需要添加更多 COM 互操作特性。

首先需要给.NET 项目添加一个 `IMathEvents` 接口。这个接口是组件的源或输出接口, 由客户端中的 `sink` 对象实现。源接口必须是调度接口或自定义接口。脚本客户端只支持调度接口。调度接口通常优先于源接口。



可从
wrox.com
下载源代码

```

[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
[ComVisible(true)]
public interface IMathEvents
{
    [DispId(46200)]
    void CalculationCompleted();
}

```

代码段 DotnetServer/DotnetServer.cs

在 `DotNetComponent` 类中, 必须指定源接口。这可以使用 `[ComSourceInterfaces]` 特性来实现。添加 `[ComSourceInterface]` 特性, 像前面那样指定输出接口。可以用特性类的不同构造函数添加多个接口, 但是支持多个源接口的唯一客户端语言是 C++。VB6 客户端仅支持一个源接口。

```

[ClassInterface(ClassInterfaceType.None)]
[ProgId("Wrox.DotnetComponent")]

```

```
[Guid("77839717-40DD-4876-8297-35B98A8402C7")]
[ComSourceInterfaces( typeof (IMathEvents))]
[ComVisible(true)]
public class DotnetComponent : IWelcome, IMath
{
    public DotnetComponent()
    {
    }
}
```

在 DotNetComponent 类中, 必须为源接口的每个方法声明一个事件。方法的类型必须是委托名, 事件名必须是源接口中的方法名。可以给 Add()方法和 Sub()方法添加事件调用。这一步是调用事件的正常.NET 方式, 详见第 8 章。

```
public event Action CalculationCompleted;

public int Add(int val1, int val2)
{
    int result = val1 + val2;
    if (CalculationCompleted != null)
        CalculationCompleted();
    return result;
}

public int Sub(int val1, int val2)
{
    int result = val1 - val2;
    if (CalculationCompleted != null)
        CalculationCompleted();
    return result;
}
}
```



事件名必须是源接口中的方法名, 否则就不能为 COM 客户端映射事件。

26.4.8 用 sink 对象创建客户端

在构建和注册.NET 程序集并把它安装到全局程序集缓存中后, 就可以使用事件源构建客户端应用程序了。这次使用 VB6 编写一个实现 IDispatch 接口的回调对象或 sink 对象。为此只需添加 With Events 关键字, 与目前 VB 处理.NET 事件的方式相似。使用 C++所需的工作较多, 但这里可以使用 ATL(Active Template Library, 活动模板库)。

打开 26.4.6 节创建的 C++控制台应用程序, 在 stdafx.h 文件中添加如下 include 语句:



可从
wrox.com
下载源代码

```
#include <atlbase.h>
extern CComModule _Module;
#include <atlcom.h>
```

代码段 COMClient/stdafx.h

文件 stdafx.cpp 需要包含 ATL 实现文件 atlimpl.cpp:



可从
wrox.com
下载源代码

```
#include <atlimpl.cpp>
```

代码段 COMClient/stdafx.cpp

给 COMClient.cpp 文件添加新的 CEventHandler 类, 这个类包含组件调用的 IDispatch 接口的实现代码。IDispatch 接口由基类 IDispatchImpl 实现。这个类读取类型库, 把方法和参数的调度 ID 匹配到类的方法上。IDispatchEventImpl 类的模板参数有 sink 对象的 ID(这里使用 ID 4)、实现回调方法的类(CEventHandler)、回调接口的接口 ID (DIID_IMathEvents)、类型库的 ID (LIBID_DotnetComponent) 和类型库的版本号。指定的 ID(DIID_IMathEvents 和 LIBID_DotnetComponent)在#import 语句创建的文件 dotnetcomponent.tlh 中。

BEGIN_SINK_MAP 和 END_SINK_MAP 封装的 sink 映射定义了由 sink 对象实现的方法。SINK_ENTRY_EX 把 OnCalcCompleted()方法映射到调度 ID 46200 上。这个调度 ID 用.NET 组件中 IMathEvents 接口的 CalculationCompleted()方法定义。



可从
wrox.com
下载源代码

```
class CEventHandler: public IDispatchImpl<4, CEventHandler,
    &DIID_IMathEvents, &LIBID_DotnetComponent, 1, 0>
{
public:
    BEGIN_SINK_MAP(CEventHandler)
        SINK_ENTRY_EX(4, DIID_IMathEvents, 46200, OnCalcCompleted)
    END_SINK_MAP()

    HRESULT __stdcall OnCalcCompleted()
    {
        cout <<"calculation completed" <<endl;
        return S_OK;
    }
};
```

代码段 COMClient/COMClient.cpp

现在, 主方法需要修改, 以告诉组件存在事件 sink 对象, 这样组件才能回调到 sink 中。为此需要使用 CEventHandler 类的 DispEventAdvise()方法, 并传递一个 IUnknown 接口指针。DispEventUnadvise()方法再注销 sink 对象。

```
int _tmain(int argc, _TCHAR* argv[])
{
    HRESULT hr;
    hr = CoInitialize(NULL);

    try
    {
        IWelcomePtr spWelcome;
        hr = spWelcome.CreateInstance("Wrox.DotnetComponent");

        IUnknownPtr spUnknown = spWelcome;

        cout<<spWelcome->Greeting("Bill") << endl;

        CEventHandler* eventHandler = new CEventHandler();
        hr = eventHandler->DispEventAdvise(spUnknown);
    }
}
```

```

IMathPtr spMath;
spMath = spWelcome; // QueryInterface()

long result = spMath->Add(4, 5);
cout << "result:" << result << endl;

eventHandler->DispEventUnadvise(spWelcome.GetInterfacePtr());
delete eventHandler;
}
catch (_com_error & e)
{
    cout << e.ErrorMessage() << endl;
}

CoUninitialize();
return 0;
}

```

26.5 平台调用

并不是 Windows API 调用的所有特性都可用于 .NET Framework。不仅旧 Win32 API 调用是这样，Windows 7 和 Windows Server 2008 R2 中的新特性也是如此。也许读者编写过一些导出非托管方法的 DLL，希望在 C# 中使用它们。



附录 A 介绍了一些 Windows 7 和 Windows Server 2008 R2 的专用特性。

要重用一個非托管的库，它不包含 COM 对象，只包含导出的函数，就可以使用平台调用服务。通过这个服务，CLR 会加载包含所需调用的函数的 DLL，并编组参数。

要使用非托管函数，首先必须找到要导出的函数名。为此，可以使用 `dumpbin` 工具及其 `/exports` 选项。

例如，下面的命令：

```
dumpbin /exports c:\windows\system32\kernel32.dll | more
```

列出了 DLL 文件 `kernel32.dll` 中的所有导出函数。在这个例子中，使用 Win32 API 函数 `CreateHardLink()` 创建与一个已有文件的硬链接。在这个 API 调用中，可以有几个引用同一个文件的文件名，只要这些文件名在同一个硬盘上即可。因为这个 API 调用不能用于 .NET Framework 4，所以必须使用平台调用服务。

要调用本地函数，必须定义一个 C# 外部方法，它的参数个数必须与本地函数相同，用非托管方法定义的类型必须对应于用托管代码映射的类型。

Windows API 调用 `CreateHardLink()` 在 C++ 中的定义如下所示：

```

BOOL CreateHardLink(
    LPCTSTR lpFileName,
    LPCTSTR lpExistingFileName,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes);

```

现在, 这个定义必须映射到 .NET 数据类型。非托管代码的返回类型是 `BOOL`, 它会映射到 `bool` 数据类型。`LPCTSTR` 定义一个指向 `const` 字符串的 `long` 指针。Windows API 给数据类型使用 Hungarian 命名约定。LP 是一个 `long` 指针, C 是常量, STR 是以 `null` 终止的字符串。T 把类型标记为泛型类型, 根据编译器的设置, 类型解析为 `LPCSTR` (ANSI 字符串) 或 `LPWSTR` (宽 Unicode 字符串)。C 字符串映射到 .NET 类型 `String`。`LPSECURITY_ATTRIBUTES` 是一个指向 `SECURITY_ATTRIBUTES` 类型的结构的 `long` 指针。因为我们把 `NULL` 传递给这个参数, 所以可以把这个类型映射到 `IntPtr`。这个方法的 C# 声明必须用 `extern` 修饰符来标记, 因为它在 C# 代码中没有该方法的实现。该方法的实现在 DLL 文件 `kernel32.dll` 中, 这是用 `[DllImport]` 特性引用的文件。 .NET 声明 `CreateHardLink()` 的返回类型是 `bool` 布尔, 本地方法 `CreateHardLink()` 返回一个 `BOOL` 值, 所以需要额外说明一下。C++ 有不同的 `Boolean` 数据类型, 例如本地 `bool` 和 Windows 定义的 `BOOL`, 它们有不同的值, 所以特性 `[MarshalAs]` 指定 .NET 类型 `bool` 应映射到什么本地类型上。

```
[DllImport("kernel32.dll", SetLastError="true",
    EntryPoint="CreateHardLink", CharSet=CharSet.Unicode)]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool CreateHardLink(string newFileName,
    string existingFilename,
    IntPtr securityAttributes);
```



网站 <http://www.pinvoke.net> 和可以从 <http://www.codeplex.com> 上下载的 P/Invoke Interop Assistant 工具非常适用于从本地代码到托管代码的转换。

表 26-3 列出了可以用 `[DllImport]` 特性指定的设置。

表 26-3

DllImport 属性或字段	说 明
EntryPoint	可以给函数的 C# 声明指定一个不同于非托管库中的名称。在非托管库中, 方法的名称在 EntryPoint 字段中定义
CallingConvention	根据编译器和用于编译非托管函数的编译器设置, 可以使用不同的调用约定。调用约定定义了参数的处理方式和它们在栈中的位置。可以通过设置一个可枚举的值, 来定义调用约定。Windows API 通常在 Windows 操作系统上使用 StdCall 调用约定, 在 Windows CE 上使用 Cdecl 调用约定。把该值设置为 CallingConvention.Winapi, 可以使 Windows API 在 Windows 和 Windows CE 环境下工作
CharSet	字符串参数可以是 ANSI 或 Unicode。使用 CharSet 设置, 可以确定如何管理字符串。用 CharSet 枚举定义的值有 Ansi、Unicode 和 Auto。CharSet.Auto 在 Windows NT 平台上使用 Unicode, 在 Windows 98 和 Windows ME 上使用 ANSI
SetLastError	如果非托管函数使用 Windows API SetLastError 设置一个错误, 就可以把 SetLastError 字段设置为 true。这样, 就可以在以后使用 Marshal.GetLastWin32Error() 方法读取错误号

为了在 .NET 环境中更方便地使用 `CreateHardLink()` 方法, 应遵循如下规则:

- 创建一个内部类 `NativeMethods`, 它包装了平台调用服务的方法调用

- 创建一个公共类，为.NET 应用程序提供本地方法功能
- 使用安全属性标记必要的安全性

在示例代码中，FileUtility 类中的公共方法 CreateHardLink()就是.NET 应用程序使用的方法。与本地 Windows API 方法 CreateHardLink()相比，公共方法 CreateHardLink()将文件名参数的顺序倒转了。第一个参数是已有文件的名称，第二个参数是新文件的名称。这类似于 Framework 中的其他类，如 File.Copy()。因为第 3 个参数给新文件名传送安全特性，但在这里的实现方式没有使用它，所以公共方法只有两个参数。返回类型也修改了，不是通过返回值 false 来返回一个错误，而是抛出一个异常。在出现错误时，非托管方法 CreateHardLink()会使用非托管方法 API SetLastError()设置错误号。要从.NET 中读取这个值，[DllImport]字段 SetLastError 应设置为 true。在托管方法 CreateHardLink()中，调用 Marshal.GetLastWin32Error()方法来读取错误号。要从这个号码中创建错误消息，应使用 System.ComponentModel 名称空间中的 Win32Exception 类。这个类通过构造函数来接收错误号，并返回一条本地化的错误消息。在出现错误时，抛出一个 IOException 类型的异常，它有一个 Win32Exception 类型的内部异常。公共方法 CreateHardLink()使用了属性 FileIOPermission，来确定调用者是否有必要的权限。NET 安全性的内容详见第 21 章。



可从
wrox.com
下载源代码

```
using System;
using System.ComponentModel;
using System.IO;
using System.Runtime.InteropServices;
using System.Security;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Interop
{
    [SecurityCritical]
    internal static class NativeMethods
    {
        [DllImport("kernel32.dll", SetLastError = true,
            EntryPoint = "CreateHardLinkW", CharSet = CharSet.Unicode)]
        [return: MarshalAs(UnmanagedType.Bool)]
        private static extern bool CreateHardLink(
            [In, MarshalAs(UnmanagedType.LPWStr)] string newFileName,
            [In, MarshalAs(UnmanagedType.LPWStr)] string existingFileName,
            IntPtr securityAttributes);

        internal static void CreateHardLink(string oldFileName,
            string newFileName)
        {
            if (!CreateHardLink(newFileName, oldFileName, IntPtr.Zero))
            {
                var ex = new Win32Exception(Marshal.GetLastWin32Error());
                throw new IOException(ex.Message, ex);
            }
        }
    }

    public static class FileUtility
    {
        [FileIOPermission(SecurityAction.LinkDemand, Unrestricted = true)]
        public static void CreateHardLink(string oldFileName,
```

```

        string newFileName)
    {
        NativeMethods.CreateHardLink(oldFileName, newFileName);
    }
}

```

代码段 PInvokeSample/NativeMethods.cs

这个类现在可以轻松地用于创建硬链接。如果通过程序的第1个参数传递的文件不存在，就会得到一个异常和一条消息“系统找不到指定的文件”。如果该文件存在，就会得到一个新的文件名，它引用原始文件。很容易验证这一点：修改一个文件中的文本，该文本也会出现在另一个文件中。

```

using System;
using System.IO;

namespace Wrox.ProCSharp.Interop
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length != 2)
            {
                Console.WriteLine("usage: PInvokeSample " +
                                   "existingfilename newfilename");
                return;
            }
            try
            {
                FileUtility.CreateHardLink(args[0], args[1]);
            }
            catch (IOException ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}

```

在本地方法调用中，常常需要使用 Windows 句柄。Windows 句柄是一个 32 位的值，它取决于句柄类型，一些值不允许使用。在 .NET 1.0 中，句柄通常使用 `IntPtr` 结构，因为可以用这个结构设置任意可能的 32 位值。但是，在一些句柄类型中，这会导致安全问题，以及线程争用条件，在最后的清理阶段泄露句柄。因此 .NET 2.0 引入了 `SafeHandle` 类。这是每个 Windows 句柄的抽象基类。Microsoft.Win32.SafeHandles 名称空间中的派生类有 `SafeHandleZeroOrMinusOneIsInvalid` 和 `SafeHandleMinusOneIsInvalid`。顾名思义，这些类不接受无效的 0 或 -1 值。更进一步的派生句柄类型有 `SafeFileHandle`、`SafeWaitHandle`、`SafeNCryptHandle` 和 `SafePipeHandle`，它们可以由特定的 Windows API 调用使用。

例如，要映射 Windows API `CreateFile()`，可以使用下面的声明返回一个 `SafeFileHandle`。当然，也可以使用 .NET 类 `File` 和 `FileInfo`。

```
[DllImport("Kernel32.dll", SetLastError = true,
```

```
        CharSet = CharSet.Unicode)]  
internal static extern SafeFileHandle CreateFile(  
    string fileName,  
    [MarshalAs(UnmanagedType.U4)] FileAccess fileAccess,  
    [MarshalAs(UnmanagedType.U4)] FileShare fileShare,  
    IntPtr securityAttributes,  
    [MarshalAs(UnmanagedType.U4)] FileMode creationDisposition,  
    int flags,  
    SafeFileHandle template);
```



第 23 章介绍了如何创建自定义 `SafeHandle` 类，来处理 Windows 7 中的事务文件 API。

26.6 小结

本章介绍了 COM 和 .NET 应用程序交互的不同实现方式。无需重写应用程序和组件，就可以从 .NET 应用程序中像使用 .NET 类那样使用 COM 组件。实现这个功能的工具是 `tlbimp`，它创建了一个 RCW，该 RCW 在 .NET 外观的后面隐藏了 COM 对象。

同样，`tlbexp` 在 .NET 组件中创建了一个由 CCW 使用的类型库，CCW 将 .NET 组件隐藏在 COM 外观的后面。要把 .NET 类用作 COM 组件，就必须使用 `System.Runtime.InteropServices` 名称空间中的一些特性，以定义 COM 客户端需要的具体 COM 特征。

利用平台调用服务，可以使用 C# 调用本地方法。平台调用服务需要用 C# 和 .NET 数据类型重新定义本地方法。定义了映射后，就可以像 C# 方法那样调用本地方法。进行交互操作的另一个方法是使用 It Just Works (IJW) 技术和 C++/CLI。C++/CLI 详见第 53 章。

第 27 章

核 心 XAML

本章内容:

- XAML 语法
- 依赖属性
- 标记扩展
- 动态加载 XAML

编写 .NET 应用程序时, 需要了解的通常不仅仅是 C# 语法。如果编写 WPF 应用程序, 使用 WF, 创建 XPS 文档, 或者编写 Silverlight 应用程序, 就还需要 XAML。XAML(eXtensible Application Markup Language, 可扩展应用程序标记语言)是一种声明性的 XML 语法, 上述这些应用程序通常需要 XAML。

本章详细介绍 XAML 的语法, 以及可用于这种标记语言的扩展机制。

27.1 概述

XAML 代码使用文本 XML 来声明。XAML 代码可以使用设计器创建, 也可以手工编写。Visual Studio 包含的设计器可给 WPF、Silverlight 或 WF 编写 XAML 代码。也可以使用其他工具创建 XAML, 如 Microsoft Expression Design 和 Microsoft Expression Blend。

XAML 和几种技术一起使用, 但这些技术是有区别的。利用 XML 名称空间 <http://schemas.microsoft.com/winfx/2006/xaml/presentation>(通过 WPF 应用程序映射为默认名称空间), 可以定义 WPF 对 XAML 的扩展。WPF 使用依赖属性、附加属性和几个 WPF 特有的标记扩展。WF 4 使用 XML 名称空间 <http://schemas.microsoft.com/netfx/2009/xaml/activities> 来定义 Workflow 活动。XML 名称空间 <http://schemas.microsoft.com/winfx/2009/xaml> 通常映射到 x 前缀上, 并定义对所有 XAML 词汇通用的功能。

XAML 元素通常映射到 .NET 类。这并不是一个严格的要求, 但通常都是如此。在 Silverlight 1.0 中, .NET 不能用于插件, 只能使用 JavaScript 解释和通过编程方法访问 XAML 代码。这种情况自 Silverlight 2.0 以来有了改变, .NET Framework 的一个小型版本是 Silverlight 插件的一部分。对于 WPF, 每个 XAML 元素都对应一个类, WF 也是这样, 例如, XAML 元素 DoWhile 是一个循环活动, 名称空间 System.Activities 中的 DoWhile 类支持它。XAML 元素 Button 与名称空间 System.Windows.Controls 中的 Button 类相同。

把 .NET 名称空间映射到 XML 别名上, 也可以在 XML 中使用自定义 .NET 类, 参见后面的内容。

在.NET 4 中, XAML 的语法得到了增强, 这个版本称为 XAML 2009。XAML 的第一个版本是 XAML 2006, 它在 XML 名称空间 <http://schemas.microsoft.com/winfx/2006/xaml/presentation> 中定义。XAML 的新版本支持增强语法, 如 XAML 代码中的泛型。但 Visual Studio 2010 版本中的 WPF 和 WF 设计器仍基于 XAML 2006。可以在应用程序中直接使用 XAML 2009 加载 XAML。本章将介绍 XAML 2009 的变化。

XAML 代码在构建过程中会发生什么? 为了编译 WPF 项目, 应在程序集 `PresentationBuildTasks` 中定义 MSBuild 任务 `MarkupCompilePass1` 和 `MarkupCompilePass2`。这些 MSBuild 任务会创建标记代码的二进制表示 BAML(Binary Application Markup Language, 二进制应用程序标记语言), 并添加到程序集的.NET 资源中。在运行期间, 会使用该二进制表示。

读写 XAML 和 BAML 可以通过读取器和写入器来完成。在 `System.Xaml` 名称空间中, 包含用于核心 XAML 特性的类, 如抽象的 `XamlReader` 类和 `XamlWriter` 类, 以及读写对象和 XAML XML 格式的具体实现方式。`System.Windows.Markup` 名称空间中的一些特性可用于使用程序集 `System.Xaml` 中的 XAML 的所有技术。在这个名称空间中, 位于 `PresentationFramework` 程序集类是 WPF 专用的扩展。例如, 为 WPF 特性进行了优化的其他 `XamlReader` 类和 `XamlWriter` 类就在该程序集中。

下面讨论 XAML 的语法。

27.1.1 元素映射到.NET 对象上

如上一节所述, XAML 元素通常映射到.NET 类上。下面利用 C#控制台项目在 Window 中通过编程方式创建一个 `Button` 对象。如下面的代码所示, 实例化一个 `Button` 对象, 并将其 `Content` 属性设置为一个字符串, 定义一个 `Window`, 设置其 `Title` 和 `Content` 属性。要编译这段代码, 需要引用程序集 `PresentationFramework`、`PresentationCore`、`WindowBase` 和 `System.Xaml`。



可从
wrox.com
下载源代码

```
using System;
using System.Windows;
using System.Windows.Controls;

namespace Wrox.ProCSharp.XAML
{
    class Program
    {
        [STAThread]
        static void Main(string[] args)
        {
            var b = new Button
            {
                Content = "Click Me!"
            };
            var w = new Window
            {
                Title = "Code Demo",
                Content = b
            };

            var app = new Application();
            app.Run(w);
        }
    }
}
```

代码段 CodeIntro/Program.cs

使用 XAML 代码可以创建类似的 UI。与以前一样,这段代码创建了一个包含 Button 元素的 Window 元素。Window 元素设置了其内容和 Title 特性。



```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="http://www.wrox.com/Schemas/2010"
        Title="XAML Demo" Height="350" Width="525">
    <StackPanel>
        <Button Content="Click Me!" />
    </StackPanel>
</Window>
```

代码段 XAMLIntro/MainWindow.xaml

当然,上面没有定义 Application 实例,它也可以用 XAML 定义。在 Application 元素中,设置了 StartupUri 特性,它链接到包含主窗口的 XAML 文件上。



```
<Application x:Class="Wrox.ProCSharp.XAML.App"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        StartupUri="MainWindow.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

代码段 XAMLIntro/App.xaml

27.1.2 使用自定义.NET 类

要在 XAML 代码中使用自定义.NET 类,只需在 XAML 中声明.NET 名称空间,并定义一个 XML 别名。为了说明这个过程,下面定义了一个简单的 Person 类及其 FirstName 和 LastName 属性。



```
namespace Wrox.ProCSharp.XAML
{
    public class Person
    {
        public Person()
        {
            FirstName = "one";
            LastName = "two";
        }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public override string ToString()
        {
            return string.Format("{0} {1}", FirstName, LastName);
        }
    }
}
```

代码段 DemoLib/Person.cs

在 XAML 中,定义一个 XML 名称空间别名 local,它映射到.NET 名称空间 Wrox.ProCSharp.XAML

上。现在可以通过该别名使用这个名称空间中的所有类。在 XAML 代码中,添加了一个列表框,其中包含 **Person** 类型的项。使用 XAML 特性,设置 **FirstName** 和 **LastName** 属性的值。运行该应用程序时, **ToString()** 方法的输出会显示在列表框中。



可从
wrox.com
下载源代码

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.XAML"
        Title="XAML Demo" Height="350" Width="525">
    <StackPanel>
        <Button Content="Click Me!" />
        <ListBox>
            <local:Person FirstName="Stephanie" LastName="Nagel" />
            <local:Person FirstName="Angela" LastName="Schoeberl" />
        </ListBox>
    </StackPanel>
</Window>
```

代码段 XAMLIntro/MainWindow.xaml



如果 .NET 名称空间与 XAML 代码不在同一个程序集中,就必须在 XAML 名称空间别名中包含该程序集名。例如 `xmlns:local="clr-namespace:Wrox.ProCSharp.XAML; assembly=XAMLIntro"`。

要把 .NET 名称空间映射到 XML 名称空间上,可以使用程序集特性 **XmlnsDefinition**。这个特性的一个变量定义了 XML 名称空间,另一个变量定义了 .NET 名称空间。使用这个特性,也可以把多个 .NET 名称空间映射到一个 XML 名称空间上。



可从
wrox.com
下载源代码

```
[assembly: XmlnsDefinition("http://www.wrox.com/Schemas/2010",
    "Wrox.ProCSharp.XAML")]
```

代码段 DemoLib/AssemblyInfo.cs

有了这个特性, XAML 代码中的名称空间声明就可以改为映射到 XML 名称空间上。

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="http://www.wrox.com/Schemas/2010"
        Title="XAML Demo" Height="350" Width="525">
    <StackPanel>
        <Button Content="Click Me!" />
        <ListBox>
            <local:Person FirstName="Stephanie" LastName="Nagel" />
            <local:Person FirstName="Angela" LastName="Schoeberl" />
        </ListBox>
    </StackPanel>
</Window>
```

27.1.3 把属性用作特性

只要属性的类型可以表示为字符串, 或者可以把字符串转换为特属性型, 就可以把属性设置为属性。下面的代码段用特性设置了 `Button` 元素的 `Content` 和 `Background` 属性。因为 `Content` 属性的类型是 `object`, 所以可以接受字符串。`Background` 属性的类型是 `Brush`, `Brush` 类型把 `BrushConverter` 类定义为一个转换器类型, 这个类用 `TypeConverter` 特性进行注解。`BrushConverter` 使用一个颜色列表, 从 `ConvertFromString()` 方法中返回一个 `SolidColorBrush`。



可从
wrox.com
下载源代码

```
<Button Content="Click Me!" Background="LightGoldenrodYellow" />
```

代码段 XAMLSyntax/MainWindow.xaml



类型转换器派生自 `System.ComponentModel` 名称空间中的基类 `TypeConverter`。需要转换的类的类型用 `TypeConverter` 特性定义了类型转换器。WPF 使用许多类型转换器把 XML 特性转换为特定的类型。`ColorConverter`、`FontFamilyConverter`、`PathFigureCollectionConverter`、`ThicknessConverter`、`GeometryConverter` 是大量类型转换器中的几个。

27.1.4 把属性用作元素

总是可以使用元素语法给属性提供值。`Button` 类的 `Background` 属性可以用子元素 `Button.Background` 设置。这样, 可以把比较复杂的画笔应用于这个属性, 如 `LinearGradientBrush`, 如下面的示例所示。

下面的示例在设置内容时, 既没有使用 `Content` 特性也没有使用 `Button.Content` 元素来写入内容, 而是直接把内容写入为 `Button` 元素的一个子值中。这是可行的, 因为 `Button` 类的一个基类是 `ContentControl`, 所以应用了 `ContentProperty` 特性, 它把 `Content` 属性标记为 `ContentProperty: [ContentProperty ("Content")]`。通过这个标记属性, 就可以把属性的值写入为子元素。



可从
wrox.com
下载源代码

```
<Button Click="OnButtonClick">
    Click Me!
    <Button.Background>
        <LinearGradientBrush StartPoint="0.5,0.0" EndPoint="0.5, 1.0">
            <GradientStop Offset="0" Color="Yellow" />
            <GradientStop Offset="0.3" Color="Orange" />
            <GradientStop Offset="0.7" Color="Red" />
            <GradientStop Offset="1" Color="DarkRed" />
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

代码段 XAMLSyntax/MainWindow.xaml

27.1.5 基本的.NET 类型

在 XAML 2006 中,核心.NET 类型需要从 XML 名称空间中引用,这与所有其他的.NET 类相同,例如, `String` 用 `sys` 别名来引用,如下所示:

```
<sys:String xmlns:sys="clr-namespace:System;assembly=mscorlib">Simple String</sys:String>
```

XAML 2009 用别名 `x` 定义了 `String`、`Boolean`、`Object`、`Decimal`、`Double`、`Int32` 等类型。

```
<x:String>Simple String</x:String>
```

27.1.6 集合

在包含 `Person` 元素的 `ListBox` 中,介绍过 XAML 中的集合。在 `ListBox` 中,列表项直接定义为子元素。另外, `LinearGradientBrush` 包含了一个 `GradientStop` 元素集合。这是可行的,因为基类 `ItemsControl` 把 `ContentProperty` 特性设置为该类的 `Items` 属性, `GradientBrush` 基类把 `ContentProperty` 特性设置为 `GradientStops`。

下面的长版本代码在定义背景时,直接设置了 `GradientStops` 属性,并把 `GradientStopCollection` 元素设置为它的子元素:



可从
wrox.com
下载源代码

```
<Button Click="OnClick">
  Click Me!
  <Button.Background>
    <LinearGradientBrush StartPoint="0.5,0.0" EndPoint="0.5, 1.0">
      <LinearGradientBrush.GradientStops>
        <GradientStopCollection>
          <GradientStop Offset="0" Color="Yellow" />
          <GradientStop Offset="0.3" Color="Orange" />
          <GradientStop Offset="0.7" Color="Red" />
          <GradientStop Offset="1" Color="DarkRed" />
        </GradientStopCollection>
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

代码段 XAMLSyntax/MainWindow.xaml

要定义数组,可以使用 `x:Array` 扩展。`x:Array` 扩展有一个 `Type` 属性,其中可以用于指定数组元素的类型。

```
<x:Array Type="local:Person">
  <local:Person FirstName="Stephanie" LastName="Nagel" />
  <local:Person FirstName="Angela" LastName="Schoeberl" />
</x:Array>
```

因为 XAML 2006 不支持泛型,所以要在 XAML 中使用泛型集合类,就需要定义一个派生自泛型类的非泛型类,再使用这个非泛型类。XAML 2009 直接支持泛型,允许通过 `x:TypeArguments` 定义泛型类型,如下面的 `ObservableCollection<T>` 所示:

```
<ObservableCollection x:TypeArguments="local:Person">
  <local:Person FirstName="Stephanie" LastName="Nagel" />
  <local:Person FirstName="Angela" LastName="Schoeberl" />
</ObservableCollection>
```

27.1.7 构造函数

如果类没有默认的构造函数，就不能在 XAML 2006 中使用它。在 XAML 2009 中，可以使用 `x:Arguments`，以通过参数调用构造函数。下面的 `Person` 类用构造函数进行实例化，该构造函数需要两个 `String` 参数。

```
<local:Person>
  <x:Arguments>
    <x:String>Stephanie</x:String>
    <x:String>Nagel</x:String>
  </x:Arguments>
</local:Person>
```

27.2 依赖属性

WPF 使用依赖属性完成数据绑定、动画、属性变更通知、样式化等。对于数据绑定，绑定到 .NET 属性源上的 UI 元素的属性必须是依赖属性。

从外部来看，依赖属性像是正常的 .NET 属性。但是，正常的 .NET 属性通常还定义了由该属性的 `get` 和 `set` 访问器访问的数据成员。

```
private int val;
public int Value
{
    get
    {
        return val;
    }
    set
    {
        val = value;
    }
}
```

依赖属性不是这样。依赖属性通常也有 `get` 和 `set` 访问器。在 `get` 和 `set` 访问器的实现代码中，调用了 `GetValue()` 和 `SetValue()` 方法。`GetValue()` 和 `SetValue()` 方法是基类 `DependencyObject` 的成员，依赖对象需要使用这个该类——它们必须在派生自 `DependencyObject` 基类的类中实现。

有了依赖属性，数据成员就放在由基类管理的内部集合中，仅在值发生变化时分配数据。对于没有变化的值，数据可以在不同的实例或基类之间共享。`GetValue()` 和 `SetValue()` 方法需要一个 `DependencyProperty` 参数。这个参数由类的一个静态成员定义，该静态成员与属性同名，并在该属性名的后面追加加上 `Property` 术语。对于 `Value` 属性，静态成员的名称是 `ValueProperty`。`DependencyProperty.Register()` 是一个帮助方法，可在依赖属性系统中注册属性。在下面的代码段中，使用 `Register()` 方法和 3 个参数定义了属性名、属性的类型和拥有者（即 `MyDependencyObject` 类）的类型。



可从
wrox.com
下载源代码

```
public int Value
{
    get { return (int)GetValue(ValueProperty); }
    set { SetValue(ValueProperty, value); }
}
```

```
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register("Value", typeof(int), typeof(MyDependencyObject));
```

代码段 DependencyObjectDemo/MyDependencyObject.cs

27.2.1 创建依赖类型

下面的示例定义的不是一个依赖属性，而是 3 个依赖属性。MyDependencyObject 类定义了依赖属性 Value、Minimum 和 Maximum。所有这些属性都是用 DependencyProperty.Register() 方法注册的依赖属性。GetValue() 和 SetValue() 方法是基类 DependencyObject 的成员。对于 Minimum 和 Maximum 属性，定义了默认值，用 DependencyProperty.Register() 方法设置该默认值时，可以把第 4 个参数设置为 ProperMetadata。使用带一个参数 ProperMetadata 的构造函数，把 Minimum 属性设置为 0，把 Maximum 属性设置为 100。



可从
wrox.com
下载源代码

```
using System;
using System.Windows;

namespace Wrox.ProCSharp.XAML
{
    class MyDependencyObject : DependencyObject
    {
        public int Value
        {
            get { return (int)GetValue(ValueProperty); }
            set { SetValue(ValueProperty, value); }
        }

        public static readonly DependencyProperty ValueProperty =
            DependencyProperty.Register("Value", typeof(int), typeof(MyDependencyObject));

        public int Minimum
        {
            get { return (int)GetValue(MinimumProperty); }
            set { SetValue(MinimumProperty, value); }
        }

        public static readonly DependencyProperty MinimumProperty =
            DependencyProperty.Register("Minimum", typeof(int), typeof(MyDependencyObject),
                new PropertyMetadata(0));

        public int Maximum
        {
            get { return (int)GetValue(MaximumProperty); }
            set { SetValue(MaximumProperty, value); }
        }

        public static readonly DependencyProperty MaximumProperty =
            DependencyProperty.Register("Maximum", typeof(int), typeof(MyDependencyObject),
                new PropertyMetadata(100));
    }
}
```

代码段 DependencyObjectDemo/MyDependencyObject.cs



在 `get` 和 `set` 属性访问器中, 只能调用 `GetValue()` 和 `SetValue()` 方法。使用依赖属性, 可以通过 `GetValue()` 和 `SetValue()` 方法从外部访问属性的值, WPF 也是这样做的, 因此, 强类型化的属性访问器可能根本就不调用, 包含它们仅为了方便自定义代码的编写。

27.2.2 强制值回调

依赖属性支持强制检查。通过强制检查, 可以检查属性的值是否有效, 例如, 该值是否在某个有效范围之内。因此本例包含了 `Minimum` 和 `Maximum` 属性。现在 `Value` 属性的注册变更为把事件处理方法 `CoerceValue()` 传递给 `ProperMetadata` 对象的构造函数, 再把 `ProperMetadata` 对象传递为 `DependencyProperty.Register()` 方法的一个参数。现在, 在 `SetValue()` 方法的实现代码中, 对属性值的每次变更都调用 `CoerceValue()` 方法。在 `CoerceValue()` 方法中, 检查 `set` 值是否在最大值和最小值之间, 如果不在, 就设置该值。



可从
wrox.com
下载源代码

```
using System;
using System.Windows;

namespace Wrox.ProCSharp.XAML
{
    class MyDependencyObject : DependencyObject
    {
        public int Value
        {
            get { return (int)GetValue(ValueProperty); }
            set { SetValue(ValueProperty, value); }
        }

        public static readonly DependencyProperty ValueProperty = DependencyProperty.Register(
            "Value", typeof(int), typeof(MyDependencyObject), new PropertyMetadata(
                0, null, CoerceValue));

        public int Minimum
        {
            get { return (int)GetValue(MinimumProperty); }
            set { SetValue(MinimumProperty, value); }
        }

        public static readonly DependencyProperty MinimumProperty =
            DependencyProperty.Register("Minimum", typeof(int), typeof(MyDependencyObject),
                new PropertyMetadata(0));

        public int Maximum
        {
            get { return (int)GetValue(MaximumProperty); }
            set { SetValue(MaximumProperty, value); }
        }

        public static readonly DependencyProperty MaximumProperty =
            DependencyProperty.Register("Maximum", typeof(int), typeof(MyDependencyObject),
                new PropertyMetadata(100));

        private static object CoerceValue(DependencyObject element, object value)
        {
            int newValue = (int)value;
            MyDependencyObject control = (MyDependencyObject)element;
```

```
newValue = Math.Max(control.Minimum, Math.Min(control.Maximum, newValue));
return newValue;
}
```

代码段 DependencyObjectDemo/MyDependencyObject.cs

27.2.3 值变更回调和事件

为了获得值变更的信息，依赖属性还支持值变更回调。在属性值发生变化时调用的 `DependencyProperty.Register()` 方法中，可以添加一个 `DependencyPropertyChanged` 事件处理程序。在示例代码中，把 `OnValueChanged()` 处理方法赋予 `ProperMetadata` 对象的 `PropertyChangedCallback` 属性。在 `OnValueChanged()` 方法中，可以用 `DependencyPropertyChangedEventArgs()` 参数访问属性的新旧值。



可从
wrox.com
下载源代码

```
using System;
using System.Windows;

namespace Wrox.ProCSharp.XAML
{
    class MyDependencyObject : DependencyObject
    {
        public int Value
        {
            get { return (int)GetValue(ValueProperty); }
            set { SetValue(ValueProperty, value); }
        }

        public static readonly DependencyProperty ValueProperty =
            DependencyProperty.Register("Value", typeof(int), typeof(MyDependencyObject),
                new PropertyMetadata(0, OnValueChanged, CoerceValue));
        //...

        private static void OnValueChanged(DependencyObject obj,
            DependencyPropertyChangedEventArgs args)
        {
            int oldValue = (int)args.OldValue;
            int newValue = (int)args.NewValue;
            //...
        }
    }
}
```

代码段 DependencyObjectDemo/MyDependencyObject.cs

27.2.4 事件的冒泡和隧道

元素可以包含在元素中。通过 XAML 和 WPF，可以定义 `Button` 包含一个 `Listbox`，该 `Listbox` 又包含 `Button` 控件选项。单击一个内层控件时，事件就应在传递到外部。WPF 支持事件的冒泡和隧道。这些事件常常成对使用。`PreviewMouseMove` 事件是一个隧道事件，它从外部向内部移动，`MouseMove` 事件跟在 `PreviewMouseMove` 事件的后面，是一个冒泡事件，从内部向外部移动。



.NET 事件的核心信息参见第 8 章。

为了说明冒泡过程, 下面的 XAML 代码包含 4 个 Button 控件, 其中外部的 StackPanel 给 Button.Click 事件定义了一个事件处理程序 OnOuterButtonClick()。button2 包含一个 Listbox, 该 Listbox 又包含两个 Button 控件作为其子项, 还包含 Click 事件处理程序 OnButton2()。这两个内部按钮也有与 Click 事件关联的事件处理程序。



可从
wrox.com
下载源代码

```
<Window x:Class=" Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel x:Name="stackPanel1" Button.Click="OnOuterButtonClick">
        <Button x:Name="button1" Content="Button 1" Margin="5" />
        <Button x:Name="button2" Margin="5" Click="OnButton2">
            <ListBox x:Name="listBox1">
                <Button x:Name="innerButton1" Content="Inner Button 1" Margin="4"
                    Padding="4" Click="OnInner1" />
                <Button x:Name="innerButton2" Content="Inner Button 2" Margin="4"
                    Padding="4" Click="OnInner2" />
            </ListBox>
        </Button>
        <ListBox ItemsSource="{Binding}" />
    </StackPanel>
</Window>
```

代码段 BubbleDemo/MainWindow.xaml

事件处理方法在代码隐藏中实现。该处理方法的第二个参数是 RoutedEventArgs 类型, 它提供了事件的 Source 和 OriginalSource 信息。尽管这个按钮并没有直接关联的 Click 事件, 但单击 Button1 按钮时, 会调用处理程序方法 OnOuterButtonClick(); 该事件会向上冒泡到容器元素。这里, Source 和 OriginalSource 是 Button1。如果第一次单击 Button2, 就调用事件处理程序 OnButton2(), 之后调用 OnOuterButtonClick()。因为处理程序 OnButton2() 改变了 Source 属性, 所以通过 OnOuterButtonClick(), 会看到与前面处理程序不同的 Source。事件的 Source 属性可以改变, 但 OriginalSource 是只读的。单击 innerButton1 按钮, 会调用 OnInner1() 事件处理程序, 之后调用 OnButton2() 和 OnOuterButtonClick()。事件向上冒泡了。单击 innerButton2 按钮, 仅调用处理程序 OnInner2(), 因为其 Handled 属性设置为 true。在这里事件的向上冒泡停止了。



可从
wrox.com
下载源代码

```
using System;
using System.Collections.ObjectModel;
using System.Windows;

namespace BubbleDemo
{
    public partial class MainWindow : Window
    {
        private ObservableCollection<string> messages = new ObservableCollection<string>();

        public MainWindow()
        {
            InitializeComponent();
            this.DataContext = messages;
        }

        private void AddMessage(string message, object sender, RoutedEventArgs e)
        {
            messages.Add(String.Format("{0}, sender: {1}; source: {2}; original source: {3}",
                message, (sender as FrameworkElement).Name,
```

```

        (e.Source as FrameworkElement).Name,
        (e.OriginalSource as FrameworkElement).Name));
    }

    private void OnOuterButtonClick(object sender, RoutedEventArgs e)
    {
        AddMessage("outer event", sender, e);
    }

    private void OnInner1(object sender, RoutedEventArgs e)
    {
        AddMessage("inner1", sender, e);
    }

    private void OnInner2(object sender, RoutedEventArgs e)
    {
        AddMessage("inner2", sender, e);
        e.Handled = true;
    }

    private void OnButton2(object sender, RoutedEventArgs e)
    {
        AddMessage("button2", sender, e);
        e.Source = sender;
    }
}

```

代码段 BubbleDemo/MainWindow.xaml.cs



改变源同时也改变事件类型很常见。例如，Button 类响应鼠标的按下和释放事件，向上冒泡它们，而不是创建按钮单击事件。



如果不同处理程序的实现代码非常类似(例如，容器中的多个按钮)，则只编写一个事件处理程序非常有益。在实现代码中，只需区分 sender 或 source。

如何在自定义类中定义事件的冒泡和隧道？MyDependencyObject 改为支持值变更时发生的事件。为了支持事件的冒泡和隧道，该类必须派生自 UIElement，而不是 DependencyObject，因为这个类要为事件定义 AddHandler() 和 RemoveHandler() 方法。

为了让 MyDependencyObject 的调用者接收到值变更的信息，该类定义了 ValueChanged 事件。这个事件用显式的 add 和 remove 处理程序来声明，其中调用了基类的 AddHandler() 和 RemoveHandler() 方法。AddHandler() 和 RemoveHandler() 方法需要一个 RoutedEventArgs 类型和委托作为参数。路由事件 ValueChangedEvent 的声明与依赖属性非常类似，它也声明为一个静态成员，并调用 EventManager.RegisterRoutedEvent() 方法来注册。这个方法需要事件名、路由策略(可以是 Bubble、Tunnel 和 Direct)、处理程序的类型，以及所有者类的类型。EventManager 类还可以注册静态事件，获取所注册事件的信息。



可从
wrox.com
下载源代码

```
using System;
using System.Windows;

namespace Wrox.ProCSharp.XAML
{
    class MyDependencyObject : UIElement
    {
        public int Value
        {
            get { return (int)GetValue(ValueProperty); }
            set { SetValue(ValueProperty, value); }
        }

        public static readonly DependencyProperty ValueProperty =
            DependencyProperty.Register("Value", typeof(int), typeof(MyDependencyObject),
            new PropertyMetadata(0, OnValueChanged, CoerceValue));

        //...

        private static void OnValueChanged(DependencyObject obj,
            DependencyPropertyChangedEventArgs args)
        {
            MyDependencyObject control = (MyDependencyObject)obj;

            RoutedPropertyChangedEventArgs<int> e = new RoutedPropertyChangedEventArgs<int>(
                (int)args.OldValue, (int)args.NewValue, ValueChangedEvent);
            control.OnValueChanged(e);
        }

        public static readonly RoutedEvent ValueChangedEvent =
           EventManager.RegisterRoutedEvent("ValueChanged", RoutingStrategy.Bubble,
            typeof(RoutedPropertyChangedEventHandler<int>), typeof(MyDependencyObject));
        public event RoutedPropertyChangedEventHandler<int> ValueChanged
        {
            add
            {
                AddHandler(ValueChangedEvent, value);
            }
            remove
            {
                RemoveHandler(ValueChangedEvent, value);
            }
        }

        protected virtual void OnValueChanged(RoutedPropertyChangedEventArgs<int> args)
        {
            RaiseEvent(args);
        }
    }
}
```

代码段 DependencyObjectDemo/MyDependencyObject.cs

现在就可以使用冒泡功能了，其方式与以前使用按钮单击事件一样。

27.3 附加属性

依赖属性是可用于特定类型的属性。而通过附加属性，可以为其他类型定义属性。一些容器控件为其子控件定义了附加属性，例如，如果使用 DockPanel 控件，就可以为其子控件使用 Dock 属

性。Grid 控件定义了 Row 和 Column 属性。

下面的代码段说明了附加属性在 XAML 中的情况。Button 类没有 Dock 属性,但它是从 DockPanel 控件附加的。

```
<DockPanel>
  <Button Content="Top" DockPanel.Dock="Top" Background="Yellow" />
  <Button Content="Left" DockPanel.Dock="Left" Background="Blue" />
</DockPanel>
```

附加属性的定义可以与依赖属性非常类似,如下面的示例所示。定义附加属性的类必须派生自基类 DependencyObject,并定义一个正常的属性, get 和 set 访问器访问基类的 GetValue()和 SetValue()方法。这些都是类似之处。接着不调用 DependencyProperty 类的 Register()方法,而是调用 RegisterAttached()方法。RegisterAttached()方法注册一个附加属性,现在它可用于每个元素。



可从
wrox.com
下载源代码

```
using System.Windows;

namespace Wrox.ProCSharp.XAML
{
    class MyAttachedPropertyProvider : DependencyObject
    {
        public int MyProperty
        {
            get { return (int)GetValue(MyPropertyProperty); }
            set { SetValue(MyPropertyProperty, value); }
        }

        public static readonly DependencyProperty MyPropertyProperty =
            DependencyProperty.RegisterAttached("MyProperty", typeof(int),
                typeof(MyAttachedPropertyProvider));
        public static void SetMyProperty(UIElement element, int value)
        {
            element.SetValue(MyPropertyProperty, value);
        }

        public static int GetMyProperty(UIElement element)
        {
            return (int)element.GetValue(MyPropertyProperty);
        }
    }
}
```

代码段 AttachedPropertyDemo/MyAttachedPropertyProvider.cs



似乎 DockPanel.Dock 属性只能添加到 DockPanel 控件中的元素上。实际上,附加属性可以添加到任何元素上。但无法使用这个属性值。DockPanel 控件能够识别这个属性,并从其子元素中读取它,以安排其子元素。

在 XAML 代码中,附加属性现在可以附加到任何元素上。第二个 Button 控件 button2 为自身附加了属性 MyAttachedPropertyProvider.MyProperty, 其值指定为 5。



可从
wrox.com
下载源代码

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:Wrox.ProCSharp.XAML"
  Title="MainWindow" Height="350" Width="525">
```

```

<Grid x:Name="grid1">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Button Grid.Row="0" x:Name="button1" Content="Button 1" />
    <Button Grid.Row="1" x:Name="button2" Content="Button 2"
        local:MyAttachedPropertyProvider.MyProperty="5" />

    <ListBox Grid.Row="2" x:Name="list1" />
</Grid>
</Window>

```

代码段 AttachedPropertyDemo/MainWindow.xaml

在代码隐藏中执行相同的操作时，必须调用 `MyAttachedPropertyProvider` 类的静态方法 `SetMyProperty()`。不能扩展 `Button` 类，使其包含某个属性。`SetMyProperty()` 方法获取一个应由该属性及其值扩展的 `UIElement` 实例。在代码段中，将该属性附加到 `button1` 中，其值设置为 44。

属性设置之后的 `foreach` 循环从 `Grid` 元素 `grid1` 的所有子元素中检索附加属性的值。检索这个值使用了 `MyAttachedPropertyProvider` 类的 `GetProperty()` 方法。它从 `DockPanel` 和 `Grid` 控件中检索其子控件的设置，以安排这些子控件。



可从
wrox.com
下载源代码

```

using System;
using System.Windows;

namespace Wrox.ProCSharp.XAML
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            MyAttachedPropertyProvider.SetMyProperty(button1, 44);

            foreach (object item in LogicalTreeHelper.GetChildren(grid1))
            {
                FrameworkElement e = item as FrameworkElement;
                if (e != null)
                    list1.Items.Add(String.Format("{0}: {1}", e.Name,
                        MyAttachedPropertyProvider.GetProperty(e)));
            }
        }
    }
}

```

代码段 AttachedPropertyDemo/MainWindow.xaml.cs



有一些机制可用于以后再扩展类。扩展方法可以用于扩展带方法的任何类。扩展方法只能扩展带方法但不带属性的类。扩展方法参见第 3 章。 `ExpandableObject` 类允许扩展带方法和属性的类。要使用这个功能，类必须派生自 `ExpandableObject` 类。 `ExpandableObject` 类和动态类型参见第 12 章。



第 35 章和第 36 章介绍了许多不同的关联属性,例如容器控件 Canvas、DockPanel、Grid 的附加属性,以及 Validation 类的 ErrorTemplate 属性。

27.4 标记扩展

通过标记扩展,可以扩展 XAML 的元素或特性语法。如果 XML 特性包含花括号,就表示这是标记扩展的一个符号。特性的标记扩展常常用作简写记号,而不再使用元素。

这种标记扩展的示例是 StaticResourceExtension, 它可查找资源。下面是带有 gradientBrush1 键的线性渐变笔刷的资源:



可从
wrox.com
下载源代码

```
<Window.Resources>
  <LinearGradientBrush x:Key="gradientBrush1" StartPoint="0.5,0.0" EndPoint="0.5, 1.0">
    <GradientStop Offset="0" Color="Yellow" />
    <GradientStop Offset="0.3" Color="Orange" />
    <GradientStop Offset="0.7" Color="Red" />
    <GradientStop Offset="1" Color="DarkRed" />
  </LinearGradientBrush>
</Window.Resources>
```

代码段 MarkupExtensionDemo/MainWindow.xaml

使用 StaticResourceExtension, 通过特性语法来设置 TextBlock 的 Background 属性, 就可以引用这个资源。特性语法通过花括号和没有 Extension 后缀的扩展类名来定义。

```
<TextBlock Text="Test" Background="{StaticResource gradientBrush1}" />
```

该特性简写记号的较长形式使用元素语法, 如下面的代码段所示。StaticResourceExtension 定义为 TextBlock.Background 元素的一个子元素。通过一个特性把 ResourceKey 属性设置为 gradientBrush1。在上面的示例中, 没有用 ResourceKey 属性设置资源键(这也是可行的), 但使用一个构造函数重载来设置资源键。

```
<TextBlock Text="Test">
  <TextBlock.Background>
    <StaticResourceExtension ResourceKey="gradientBrush1" />
  </TextBlock.Background>
</TextBlock>
```

27.5 创建自定义标记扩展

要创建标记扩展, 可以定义一个派生自基类 MarkupExtension 的类。大多数标记扩展名都有 Extension 后缀(这个命名约定类似于特性的 Attribute 后缀, 参见第 14 章)。有了自定义标记扩展后, 就只需重写 ProvideValue() 方法, 它返回扩展的值。返回的类型用 MarkupExtensionReturnType 特性注解了类。对于 ProvideValue() 方法, 需要传递一个 IServiceProvider 对象。通过这个接口, 可以查询不同的服务, 如 IProvideValueTarget 或 IXamlTypeResolver。IProvideValueTarget 可以用于访问通过 TargetObject 和 TargetProperty 属性应用标记扩展的控件和属性。IXamlTypeResolver 可用于把 XAML

元素名解析为 CLR 对象。自定义标记扩展类 `CalculatorExtension` 定义了 `double` 类型的属性 `X` 和 `Y`，并通过枚举定义了一个 `Operation` 属性。根据 `Operation` 属性的值，在 `X` 和 `Y` 输入属性上执行不同的计算，并返回一个字符串。



可从
wrox.com
下载源代码

```
using System;
using System.Windows;
using System.Windows.Markup;

namespace Wrox.ProCSharp.XAML
{
    public enum Operation
    {
        Add,
        Subtract,
        Multiply,
        Divide
    }

    [MarkupExtensionReturnType(typeof(string))]
    public class CalculatorExtension : MarkupExtension
    {
        public CalculatorExtension()
        {
        }

        public double X { get; set; }
        public double Y { get; set; }

        public Operation Operation { get; set; }
        public override object ProvideValue(IServiceProvider serviceProvider)
        {
            IProvideValueTarget provideValue =
                serviceProvider.GetService(typeof(IProvideValueTarget))
                    as IProvideValueTarget;

            if (provideValue != null)
            {
                var host = provideValue.TargetObject as FrameworkElement;
                var prop = provideValue.TargetProperty as DependencyProperty;
            }

            double result = 0;
            switch (Operation)
            {
                case Operation.Add:
                    result = X + Y;
                    break;
                case Operation.Subtract:
                    result = X - Y;
                    break;
                case Operation.Multiply:
                    result = X * Y;
                    break;
                case Operation.Divide:
                    result = X / Y;
                    break;
                default:
                    throw new ArgumentException("invalid operation");
            }

            return result.ToString();
        }
    }
}
```

代码段 MarkupExtensionDemo/CalculatorExtension.cs

标记扩展现在可以在第一个 TextBlock 上与特性语法一起使用，把值 3 和 4 加在一起，或者在第二个 TextBlock 上与元素语法一起使用。



可从
wrox.com
下载源代码

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.XAML"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel>
        <TextBlock Text="{local:Calculator Operation=Add, X=3, Y=4}" />
        <TextBlock>
            <TextBlock.Text>
                <local:CalculatorExtension>
                    <local:CalculatorExtension.Operation>
                        <local:Operation> Multiply </local:Operation>
                    </local:CalculatorExtension.Operation>
                    <local:CalculatorExtension.X> 7 </local:CalculatorExtension.X>
                    <local:CalculatorExtension.Y> 11 </local:CalculatorExtension.Y>
                </local:CalculatorExtension>
            </TextBlock.Text>
        </TextBlock>
    </StackPanel>
</Window>
```

代码段 MarkupExtensionDemo/MainWindow.xaml

27.6 XAML 定义的标记扩展

标记扩展提供了许多功能。实际上本章已经使用了 XAML 定义的标记扩展。21.6 节中的 `x:Array` 就定义为标记扩展类 `ArrayExtension`。有了这个标记扩展，就可以不使用特性语法，因为使用特性语法很难定义元素列表。

用 XAML 定义的其他标记扩展有 `TypeExtension(x:Type)`，它根据字符串输入返回类型；`NullExtension(x:Null)`，它可以用于在 XAML 中把值设置为空；`StaticExtension(x:static)`，它可调用类的静态成员。

WPF、WF 和 WCF 都定义了专用于这些技术的标记扩展。WPF 使用标记扩展访问资源，以用于数据绑定和颜色转换。WF 联合使用了标记扩展和活动；WCF 给端点定义指定了标记扩展。

27.7 读写 XAML

有几个 API 可用于读写 XAML。高级 API 易于使用，但功能较少，而低级 API 的功能较多。还存在专用技术的 API，它们使用专门的 WPF 或 WF 特性。XAML 可以从文本 XML 形式、BAML 或对象树中读取，还可以写入 XML 或对象树。

一般，高级 API 在 `System.Xaml` 名称空间中。`XamlService` 类允许加载、解析、保存和转换 XAML。`XamlService.Load()` 方法可以从文件、流或读取器中加载 XAML 代码，或者使用 `XamlReader` 对象来

加载。XamlReader(在 System.Xaml 名称空间中)是一个抽象基类,它有几中具体的实现方式。XamlObjectReader 读取对象树,XamlXmlReader 从 XML 文件中读取 XAML,Baml2006Reader 读取二进制形式的 XAML。System.Activities.Debugger 名称空间中的 XamlDebuggerXmlReader 是一个专用于 WF 的读取器,并具备特殊的调试支持。

把字符串中的 XAML 代码传递给 XamlService 时,可以使用 Parse()方法。XamlService.Save()方法可用于保存 XAML 代码。通过 Save()方法可以使用与 Load()方法类似的数据源。所传递的对象可以保存为字符串、流、TextWriter、XamlWriter 或 XmlWriter。XamlWriter 是一个抽象基类。派生自 XamlWriter 的类是 XamlObjectWriter 和 XamlXmlWriter。在 .NET 4 版本中,BAML 代码没有写入器,但以后会有 BAML 写入器。

把 XAML 从一种格式转换为另一种格式时,可以使用 XamlService.Transform()方法。给 Transform()方法传递 XamlReader 和 XamlWriter,就可以转换特定读取器和写入器支持的任意格式。

除了使用高级 API XamlService 类之外,还可以直接使用一般的低级 API,也就是说,使用特定的 XamlReader 和 XamlWriter 类。使用读取器,可以通过 Read()方法从 XAML 树中逐个节点地读取。

一般的 XamlService 类不支持特定的 WPF 特性,如依赖属性或可冻结的对象。要读取 WPF XAML,可以使用 System.Windows.Markup 名称空间中的 XamlReader 和 XamlWriter 类,该名称空间在 PresentationFramework 程序集中定义,因此可以访问 WPF 特性。这些类的名称可能令人迷惑,因为不同名称空间的类使用相同的名称。System.Xaml.XamlReader 是读取器的抽象基类, System.Windows.Markup.XamlReader 是读取 XAML 的 WPF 类。在使用 WPF XamlReader 类的 Load()方法时,它接受 System.Xaml.XamlReader 作为参数,所以可能更容易混淆。

给 WF 读取 XAML 的一个优化版本是 System.Activities 名称空间中的 WorkflowXamlService,这个类用于在运行期间创建动态活动。



动态活动和 Windows Workflow Foundation 4 参见第 44 章。

下面的简单示例从文件中动态加载 XAML,以创建一个对象树,并把该对象树附加到一个容器元素中,如 StackPanel:

```
FileStream stream = File.OpenRead("Demo1.xaml");
object tree = System.Windows.Markup.XamlReader.Load(stream);

container1.Children.Add(tree as UIElement);
```

27.8 小结

本章介绍了 XAML 的核心功能和一些专门的特性,如依赖属性、附加属性和标记扩展。讨论了 XAML 的可扩展性,这是其他技术如 WPF 和 WF 的基础。在其他使用 XAML 的章节中,将列举这些 XAML 特性的大量不同用法。

本书的许多章节都涉及 XAML,尤其是介绍 WPF 的第 35 章和第 36 章、介绍 XPS 的第 37 章、介绍 Silverlight 的第 38 章和介绍 Windows Workflow Foundation 的第 44 章。

下一章讨论 MEF,即 Managed Extensibility Framework。

第28章

Managed Extensibility Framework

本章内容:

- Managed Extensibility Framework 的体系结构
- 合同
- 部件的导出和导入
- 宿主应用程序使用的容器
- 查找部件的分类

插件可以在以后给应用程序添加功能。我们可以创建一个宿主应用程序，它随着时间的推移会获得越来越多的功能——这些功能可能是开发团队编写的，不同的开发商也可以创建插件，来扩展自己的应用程序。

目前，插件用于许多不同的应用程序，如 Internet Explorer 和 Visual Studio。Internet Explorer 是一个宿主应用程序，它提供了一个插件架构，供许多公司提供浏览网页时的扩展。Shockwave Flash Object 允许浏览包含 Flash 内容的网页。Google 工具栏提供了可以从 Internet Explorer 上快速访问的特定 Google 功能。Visual Studio 也有一个插件模型，它允许用不同级别的扩展来扩展 Visual Studio。

对于自定义应用程序，总是可以创建一个插件模型，来动态加载和使用程序集中的功能。但需要解决查找和使用插件的问题。这个任务可以使用 Managed Extensibility Framework 自动完成。

本章介绍的主要名称空间是 `System.ComponentModel.Composition`。

28.1 MEF 的体系结构

.NET 4 Framework 提供了两个技术，以编写动态加载插件的灵活的应用程序。一种技术是本章介绍的 Managed Extensibility Framework(MEF)；另一个技术自从.NET 3.5 以来就有，即 Managed Add-in Framework(MAF)，参见可下载的第 50 章和光盘中的对应内容。MAF 使用一个管道在插件和宿主应用程序之间通信，使开发过程比较复杂，但通过应用程序域甚或不同的进程使插件彼此分开。在这方面，MEF 是两种技术中比较简单的。MAF 和 MEF 可以合并起来，发挥各自的长处(而且完成了两倍的工作)。

MEF 通过部件和容器来构建，如图 28-1 所示。容器在类别中查找部件，类别在程序集或目录中查找部件。容器把入口连接到出口上，因此使部件可用于宿主应用程序。

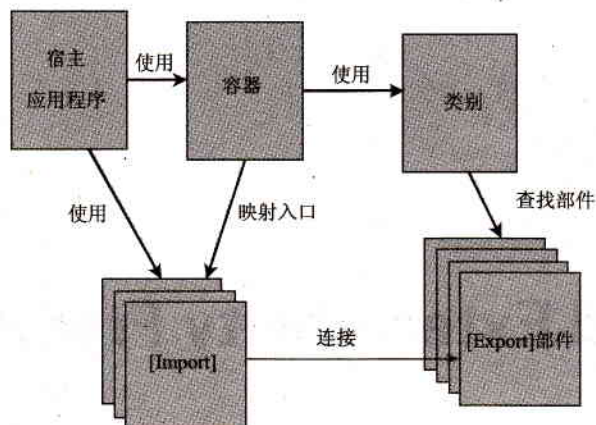


图 28-1

下面是部件加载的完整过程。在类别中查找部件。类别使用出口来查找其部件。出口提供程序访问类别，提供类别中的出口。多个出口提供程序可以连接成链，以定制出口，例如，使自定义出口提供程序只允许部件用于特定的用户或角色。容器使用出口提供程序把入口连接到出口上，该容器自己就是一个出口提供程序。

MEF 包含 3 个大类别：用于宿主的类、基元类和基于特性机制的类。宿主类包含类别和容器。基元类可以用作基类，来扩展 MEF 体系结构，以使用其他技术连接出口和入口。当然，通过反射构成基于特性机制的实现方式的类，如 `Export` 和 `Import` 特性，和提供扩展方法、更便于使用基于特性的部件的类，这些也是 MEF 的一部分。



MEF 实现方式基于特性，它标记出了哪些部件应导出，并把它们映射到入口上。但是，这个技术很灵活，允许使用抽象基类 `ComposablePart`、扩展方法以及 `ReflectionModelServices` 类中基于反射的机制，来实现其他技术。这个体系结构可以扩展，以从 `ComposablePart` 中派生出类，并提供其他扩展方法，通过其他机制从插件中检索信息。

下面用一个简单的例子来说明 MEF 体系结构。宿主应用程序可以动态地加载插件。通过 MEF，把插件表示为部件。部件定义为出口，并加载到一个导入部件的容器中。容器使用类别来查找部件。类别列出部件。

在这个例子中，创建了一个简单的控制台应用程序作为宿主，以包含库中的计算器插件。为了使宿主和计算器插件彼此独立，需要 3 个程序集。其中一个程序集 `SimpleContract` 包含了插件程序集和宿主可执行文件都使用的合同。插件程序集 `SimpleCalculator` 实现了合同程序集定义的合同。宿主使用合同程序集调用插件。

`SimpleContract` 程序集中的合同通过两个接口 `ICalculator` 和 `IOperation` 来定义。`ICalculator` 接口定义了 `GetOperations()` 和 `Operate()` 方法。`GetOperations()` 方法返回插件计算器支持的所有操作对应的列表，`Operate()` 方法可调用操作。这个接口很灵活，因为计算器可以支持不同的操作。如果该接口定义了 `Add()` 和 `Subtract()` 方法，而不是灵活的 `Operate()` 方法，就需要一个新版本的接口来支持 `Divide()`

和 `Multiply()` 方法。而使用本例定义的 `ICalculator` 接口, 计算器就可以提供任意多个操作, 且有任意多个操作数。



```
using System.Collections.Generic;

namespace Wrox.ProCSharp.MEF
{
    public interface ICalculator
    {
        IList<IOperation> GetOperations();
        double Operate(IOperation operation, double[] operands);
    }
}
```

代码段 SimpleContract/ICalculator.cs

`ICalculator` 接口使用 `IOperation` 接口返回操作列表, 并调用一个操作。`IOperation` 接口定义了只读属性 `Name` 和 `NumberOperands`。



```
namespace Wrox.ProCSharp.MEF
{
    public interface IOperation
    {
        string Name { get; }
        int NumberOperands { get; }
    }
}
```

代码段 SimpleContract/IOperation.cs

`SimpleContract` 程序集不需要引用 `MEF` 程序集, 只要 .NET 接口包含在其中即可。

插件程序集 `SimpleCalculator` 包含的类实现了合同定义的接口。`Operation` 类实现了 `IOperation` 接口。这个类仅包含接口定义的两个属性。该接口定义了属性的 `get` 访问器; 内部的 `set` 访问器用于在程序集内部设置属性。



```
namespace Wrox.ProCSharp.MEF
{
    public class Operation : IOperation
    {
        public string Name { get; internal set; }
        public int NumberOperands { get; internal set; }
    }
}
```

代码段 SimpleCalculator/Operation.cs

`Calculator` 类实现了 `ICalculator` 接口, 从而提供了这个插件的功能。按照 `Export` 特性的定义, `Calculator` 类导出为部件。这个特性在 `System.ComponentModel.Composition` 程序集的 `System.ComponentModel.Composition` 名称空间中定义。



```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;

namespace Wrox.ProCSharp.MEF
```

```

[Export(typeof(ICalculator))]
public class Calculator : ICalculator
{
    public IList<IOperation> GetOperations()
    {
        return new List<IOperation>()
        {
            new Operation { Name="+", NumberOperands=2},
            new Operation { Name="-", NumberOperands=2},
            new Operation { Name="/", NumberOperands=2},
            new Operation { Name="*", NumberOperands=2}
        };
    }

    public double Operate(IOperation operation, double[] operands)
    {
        double result = 0;
        switch (operation.Name)
        {
            case "+":
                result = operands[0] + operands[1];
                break;
            case "-":
                result = operands[0] - operands[1];
                break;
            case "/":
                result = operands[0] / operands[1];
                break;
            case "*":
                result = operands[0] * operands[1];
                break;
            default:
                throw new InvalidOperationException(
                    String.Format("invalid operation {0}", operation.Name));
        }
        return result;
    }
}

```

代码段 SimpleCalculator/Calculator.cs

宿主应用程序是一个简单的控制台应用程序。插件使用 **Export** 特性定义导出的内容，对于宿主应用程序，**Import** 特性定义了所使用的信息。在本例中，**Import** 特性注解了 **Calculator** 属性，以设置和获取实现 **ICalculator** 接口的对象。所以实现了这个接口的任意计算器插件都可以在这里使用。



可从
wrox.com
下载源代码

```

using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;

```

```

using Wrox.ProCSharp.MEF.Properties;

namespace Wrox.ProCSharp.MEF
{
    class Program
    {
        [Import]
        public ICalculator Calculator { get; set; }
    }
}

```

代码段 SimpleHost/Program.cs

在控制台应用程序的入口方法 `Main()` 中, 创建了 `Program` 类的一个新实例, 接着调用 `Run()` 方法。在 `Run()` 方法中, 创建了一个 `DirectoryCatalog`, 并用 `AddInDirectory` 进行初始化, `AddInDirectory` 在应用程序配置文件中配置。Settings.Default.AddInDirectory 通过项目属性 Settings, 使用一个强类型化的类来访问自定义配置。

`CompositionContainer` 类是一个部件存储库。这个容器用 `DirectoryCatalog` 初始化, 从这个类别指定的目录中获取部件。`ComposeParts()` 是一个扩展了 `CompositionContainer` 类的扩展方法, 它用 `AttributeModelServices` 类定义。这个方法需要把带 `Import` 特性的部件和参数一起传递。因为 `Program` 类有 `Import` 特性和 `Calculator` 属性, 所以 `Program` 类的实例可以传递给这个方法。在用于入口的实现代码中, 搜索并映射出口。成功调用了这个方法后, 就可以使用映射到入口的出口了。如果不是所有入口都可以映射到出口上, 就抛出一个 `ChangeRejectedException` 类型的异常, 捕获该异常以输出错误消息, 并从 `Run()` 方法退出。

```

static void Main()
{
    var p = new Program();
    p.Run();
}

public void Run()
{
    var catalog = new DirectoryCatalog(Settings.Default.AddInDirectory);
    var container = new CompositionContainer(catalog);
    try
    {
        container.ComposeParts(this);
    }
    catch (ChangeRejectedException ex)
    {
        Console.WriteLine(ex.Message);
        return;
    }
}

```

通过 `Calculator` 属性, 可以使用 `ICalculator` 接口中的方法。`GetOperations()` 方法调用前面创建的插件的方法, 它返回 4 个操作。要求用户指定应调用的操作, 并输入操作数的值后, 就调用插件方法 `Operate()`。

```

var operations = Calculator.GetOperations();
var operationsDict = new SortedList<string, IOperation>();
{
    Console.WriteLine("Name: {0}, number operands: {1}", item.Name,

```

```

        item.NumberOperands);
    operationsDict.Add(item.Name, item);
}
Console.WriteLine();
string selectedOp = null;
do
{
    try
    {
        Console.Write("Operation? ");
        selectedOp = Console.ReadLine();
        if (selectedOp.ToLower() == "exit" ||
            !operationsDict.ContainsKey(selectedOp))
            continue;

        var operation = operationsDict[selectedOp];
        double[] operands = new double[operation.NumberOperands];
        for (int i = 0; i < operation.NumberOperands; i++)
        {
            Console.Write("\t operand {0}? ", i + 1);
            string selectedOperand = Console.ReadLine();
            operands[i] = double.Parse(selectedOperand);
        }

        Console.WriteLine("calling calculator");
        double result = Calculator.Operate(operation, operands);
        Console.WriteLine("result: {0}", result);
    }
    catch (FormatException ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine();
        continue;
    }
} while (selectedOp != "exit");
}
}

```

运行应用程序的一个示例结果如下:

```

Name: +, number operands: 2
Name: -, number operands: 2
Name: /, number operands: 2
Name: *, number operands: 2

```

```

Operation? +
    operand 1? 3
    operand 2? 5
calling calculator
result: 8
Operation? -
    operand 1? 7
    operand 2? 2
calling calculator
result: 5
Operation? exit

```

无须重编译宿主应用程序，就可以使用另一个完全不同的插件库。AdvCalculator 程序集定义了 Calculator 类的另一种实现方式，以提供更多的操作。通过把程序集复制到宿主应用程序中 DirectoryCatalog 指定的目录下，就可以用这个计算器替代另一个计算器。



可从
wrox.com
下载源代码

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;

namespace Wrox.ProCSharp.MEF
{
    [Export(typeof(ICalculator))]
    public class Calculator : ICalculator
    {
        public IList<IOperation> GetOperations()
        {
            return new List<IOperation>()
            {
                new Operation { Name="+", NumberOperands=2},
                new Operation { Name="-", NumberOperands=2},
                new Operation { Name="/", NumberOperands=2},
                new Operation { Name="*", NumberOperands=2},
                new Operation { Name="%", NumberOperands=2},
                new Operation { Name="++", NumberOperands=1},
                new Operation { Name="--", NumberOperands=1}
            };
        }

        public double Operate(IOperation operation, double[] operands)
        {
            double result = 0;
            switch (operation.Name)
            {
                case "+":
                    result = operands[0] + operands[1];
                    break;
                case "-":
                    result = operands[0] - operands[1];
                    break;
                case "/":
                    result = operands[0] / operands[1];
                    break;
                case "*":
                    result = operands[0] * operands[1];
                    break;
                case "%":
                    result = operands[0] % operands[1];
                    break;
                case "++":
                    result = ++operands[0];
                    break;
                case "--":
                    result = --operands[0];
                    break;
                default:
                    throw new InvalidOperationException(
```

```

        String.Format("invalid operation {0}", operation.Name));
    }
    return result;
}
}
}

```

代码段 AdvCalculator/Calculator.cs

前面讨论了 MEF 体系结构中的入口、出口和类别。下面使用一个 WPF 应用程序来包含插件，详细论述 MEF 的不同选项。

28.2 协定

下面的示例应用程序扩展了第一个应用程序。宿主应用程序是一个 WPF 应用程序，它加载计算器插件，以实现计算功能，还加载其他插件，把它们自己的用户界面引入宿主。



编写 WPF 应用程序的更多信息，可参见第 35 章和第 36 章。

对于计算，使用前面定义的不同协定：ICalculator 和 IOperation 接口。另一个协定是 ICalculatorExtension。这个接口定义了可由宿主应用程序使用的 Title 和 Description 属性。GetUI() 方法返回一个 FrameworkElement，它允许插件返回任意派生自 FrameworkElement 的 WPF 元素，并在宿主程序中显示为用户界面。



可从
wrox.com
下载源代码

```

using System.Windows;

namespace Wrox.ProCSharp.MEF
{
    public interface ICalculatorExtension
    {
        string Title { get; }
        string Description { get; }

        FrameworkElement GetUI();
    }
}

```

代码段 CalculatorContract/ICalculatorExtension.cs

.NET 接口在宿主应用程序和插件之间建立了一个很好的协定。如果接口在一个单独的程序集中定义，与 CalculatorContract 程序集一样，宿主应用程序和插件就没有直接的依赖关系。然而，宿主应用程序和插件仅引用协定程序集。

从 MEF 的角度来看，根本不需要接口协定。协定可以是一个简单的字符串。为了避免与其他协定冲突，字符串名应包含名称空间名，例如 Wrox.ProCSharp.MEF.SampleContract，如下面的代码段所示。这里的 Foo 类使用 Export 特性导出，并把一个字符串传递给该特性，而不是传递给接口。

```
[Export("Wrox.ProCSharp.MEF.SampleContract")]
```

```
public class Foo
{
    public string Bar()
    {
        return "Foo.Bar";
    }
}
```

把协定用作字符串的问题是，类型提供的方法、属性和事件都不是强定义的。调用者或者需要引用类型 `Foo`，才能使用它，或者使用 .NET 反射来访问其成员。C# 4 的 `dynamic` 关键字使反射更便于使用，在这种情况下非常有帮助。

宿主应用程序可以使用 `dynamic` 类型导入 `Wrox.ProCSharp.MEF.SampleContract` 协定：

```
[Import("Wrox.ProCSharp.MEF.SampleContract")]
public dynamic Foo { get; set; }
```

有了 `dynamic` 关键字，`Foo` 属性就可以用于直接访问 `Bar()` 方法。这个方法的调用会在运行期间解析：

```
string s = Foo.Bar();
```



`dynamic` 类型参见第 12 章。

协定名和接口也可以联合使用，来定义只有接口和协定同名时才能使用的协定。这样，就可以对于不同的协定使用同一个接口。

28.3 导出

在前面的例子中，包含了 `SimpleCalculator` 部件，它导出了 `Calculator` 类型及其所有的方法和属性。下面的例子也包含 `SimpleCalculator`，其实现方式与前面的相同。这里要使用的另一个部件是一个 WPF 用户控件库 `TemperatureConversion`，它定义了如图 28-2 所示的用户界面。这个控件提供了摄氏、华氏和绝对温度之间的转换。在第一组合框和第二个组合框中，可以选择转换的源和目标。Calculate 按钮会启动执行转换的计算过程。

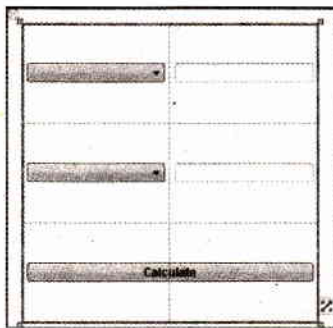


图 28-2