

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

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

51.9 小结

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

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

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

第 52 章

目录服务

本章内容:

- Active Directory 的体系结构和概念
- 访问 Active Directory 的工具
- 如何读取和修改 Active Directory 中的数据
- 搜索 Active Directory 中的对象
- 以编程方式管理用户和组
- 使用 DSML(Directory Service Markup Language, 目录服务标记语言)访问 Active Directory

Microsoft 的 Active Directory 是一种目录服务, 它提供用户信息、网络资源和服务等中心分层存储器。还可以扩展这个目录服务中的信息, 同时存储企业感兴趣的自定义数据。例如, Microsoft Exchange Server 和 Microsoft Dynamics 广泛使用 Active Directory 来存储公共文件夹和其他项。

在 Active Directory 发布之前, Exchange Server 使用它自己的私有存储器来存储对象。系统管理员必须为一个人配置两个用户 ID: Windows NT 域中的用户账户(启用登录), 和 Exchange Directory 中的用户账户。这是必需的, 因为需要用户的其他信息(如电子邮件地址, 电话号码等), NT 域的用户信息不能扩展, 以添加需要的信息。

目前系统管理员只需要在 Active Directory 上为一个人配置一个用户账户; user 对象的信息可以扩展, 以便它满足 Exchange Server 的要求。我们也可以扩展这些信息。例如, 可以在 Active Directory 上用一个技能列表扩展用户信息。然后, 就可以搜索需要的 C#技能, 跟踪 C#开发人员。

本章介绍如何在 .NET Framework 中使用 System.DirectoryServices、System.DirectoryServices.AccountManagement 和 System.DirectoryServices.Protocols 名称空间中的类访问和处理目录服务中的数据。



本章使用兼具 Active Directory 配置的 Windows Server 2008 R2, 也可以使用 Windows 2003 Server 或其他目录服务。

在讨论了体系结构和如何对 Active Directory 编程之后, 就要创建一个 Windows 应用程序, 在该应用程序中指定一些属性和一个筛选器, 搜索 user 对象。与其他章节一样, 本章的示例代码也可以从 Wrox 网站 www.wrox.com 或随书附赠光盘中找到。

52.1 Active Directory 的体系结构

在开始对 Active Directory 编程前, 必须知道 Active Directory 的工作方式、用途, 以及什么数据可以存储在 Active Directory 中。

52.1.1 Active Directory 的功能

Active Directory 的功能可以总结为:

- Active Directory 中的数据以分层的方式组合。对象可以存储在其他容器对象中。用户并不是放在一个大型用户列表中, 而是组合到组织单元中。因为组织单元可以包含其他组织单元, 所以以这种方式可以构建一个树型视图。
- Active Directory 使用多主机复制方式(multimaster replication)。在 Active Directory 中, 每个域控制器(DC)都是主机。在多主机模型中, 更新可以应用于所有 DC。与单主机模型相比, 这个模型的伸缩性比较高, 因为可以同时在不同的服务器上更新。该模型的缺点是复制起来比较复杂。本章后面会讨论复制问题。
- 灵活的复制拓扑(replication topology), 这通过 WAN 中的慢速链接支持复制。数据复制的频率由域管理员配置。
- Active Directory 支持开放标准。LDAP(Lightweight Directory Access Protocol, 轻型目录访问协议)是一个 Internet 标准, 该标准用于访问许多不同目录服务, 包括 Active Directory 中的数据。在 LDAP 中, 也定义一个编程接口 LDAP API。LDAP API 可以使用 C 语言来访问 Active Directory。在 Active Directory 中使用的另一个标准是 Kerberos, 它用于身份验证。Windows Server Kerberos 服务也可用于验证 UNIX 客户端的身份。
- 活动目录服务接口(Active Directory Service Interface, ADSI)定义访问目录服务的 COM 接口。ADSI 可以访问 Active Directory 的所有功能。System.DirectoryServices 名称空间中的类包装 ADSI COM 对象, 用于从 .NET 应用程序中访问目录服务。
- 目录服务标记语言(Directory Service Markup Language, DSML)是另一个访问目录服务的标准, 它是独立于平台的方法, 得到 OASIS 组的支持。
- 细粒度安全性。对于 Active Directory, 细粒度安全性可用。存储在 Active Directory 中的每个对象都可以有一个关联的访问控制列表, 该列表确定谁可以对对象进行哪些处理。

目录中的对象都是强类型化的, 这说明, 对象的类型是精确定义的, 没有给对象添加未指定的属性。在架构中, 定义了对象类型和对象的各部分(特性)。特性可以是必选的, 也可以是可选的。

52.1.2 Active Directory 的概念

在编程之前, 必须了解 Active Directory 的一些基本术语和定义。

1. 对象

可以在 Active Directory 中存储对象。对象可以是用户、打印机或网络共享。对象包含描述它们的必选特性和可选特性。例如, user 对象的特性包含姓、名、电子邮件地址和电话号码等。

图 52-1 显示了一个容器对象，即 Wrox Press，它包含一些其他对象：两个用户对象、一个联系人对象、一台打印机对象和一个用户组对象。

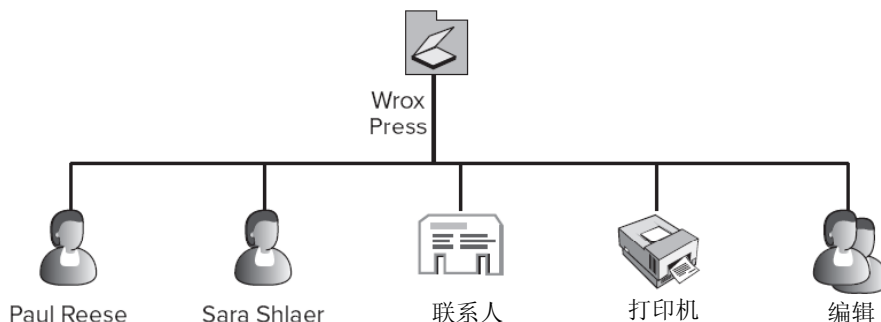


图 52-1

2. 架构

每个对象是类的一个实例，这个类在架构中定义。架构可以定义类型，架构本身存储在 Active Directory 的对象中。必须区分 classSchema 和 attributeSchema。对象的类型在 classSchema 中定义，其必选特性和可选特性也在 classSchema 中定义。attributeSchema 则定义特性的外观，以及特定特性的语法。

我们可以定义自定义类型和特性，并把它们添加到架构中。但要注意，不能从 Active Directory 中删除新架构类型。可以把它标记为未激活，这样就不能再创建新对象，但因为已有的对象可以是该类型，所以不能删除在架构中定义的类型或特性。

Administrator 用户组没有足够的权限创建新架构项，此时需要 Enterprise Admins 用户组。

3. 配置

除了对象和存储为对象的类定义之外，Active Directory 本身的配置也存储在 Active Directory 中。Active Directory 的配置存储所有站点的信息，如复制间隔等，这些都由系统管理员设置。因为配置本身存储在 Active Directory 中，所以可以像访问 Active Directory 中的所有其他对象那样访问配置信息。

4. Active Directory 域

域是 Windows 网络的安全边界。在 Active Directory 域中，对象以分层顺序进行存储。Active Directory 本身由一个或多个域组成。图 52-2 显示了域中对象的分层顺序，其中域用一个三角形表示。容器对象如 Users、Computers 和 Books 都可以存储其他对象。图 52-2 中的每个椭圆表示一个对象，对象之间的连线表示父-子关系。例如，Books 是 .NET 和 Java 的父对象，Pro C#、Beg C# 和 ASP.NET 是 .NET 对象的子对象。

5. 域控制器

一个域可以有多个域控制器，每个控制器存储域中的所有对象。没有主服务器，所有 DC 都一视同仁。这是一个多主机模型。可以在域中跨服务器复制对象。

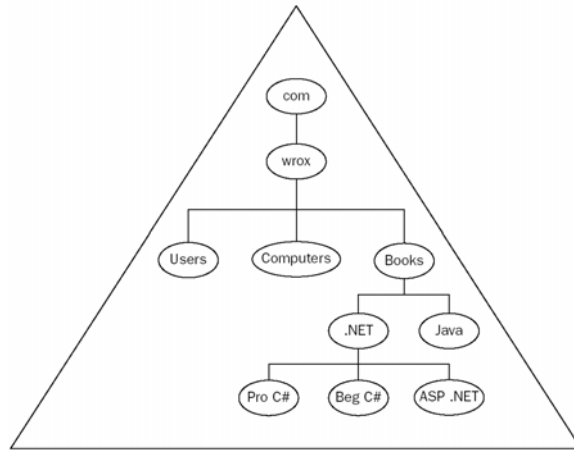


图 52-2

6. 站点

站点是网络中的一个位置，它至少拥有一个 DC。如果企业有多个办事处，办事处用慢速网络链接来连接，就可以在一个域中使用多个站点。由于备份或可伸缩性原因，每个站点都有一个或多个 DC 在运行。在一个站点的服务器之间复制数据，其间隔比较短，因为网络连接比较快。在跨站点的服务器之间进行复制的时间间隔可以配置得较长，这取决于网络的速度。当然，复制间隔由域管理员配置。

7. 域树

通过信任关系可以连接多个域。这些域共享公共架构、公共配置和全局目录(后面将介绍全局目录)。公共架构和公共配置表示可以跨域复制该数据。域树共享相同的类和属性架构。不能跨域复制对象自身。

以这种方式连接的域就构成了域树。域树中的域有一个连续和分层的名称空间。这表示，子域的域名是把子域的名称追加到父域名后。在这些域之间，建立使用 Kerberos 协议的信任关系。

例如，有一个根域 `wrox.com`，它是子域 `india.wrox.com` 和 `uk.wrox.com` 的父域。在父域和子域之间建立信任关系，这样域中的账户可以由其他域验证身份。

8. 森林

使用公共架构、公共配置和全局目录来连接多个域树，但没有使用连续的名称空间，该名称空间称为森林(forest)。森林是一组域树。如果公司有使用不同域名的子公司，就可以使用森林。例如，`wrox.com` 域应相对独立于 `wiley.com` 域，但应有一个公共管理，`wrox.com` 域的用户可以访问 `wiley.com` 域中的资源，反之亦然。使用森林，可以在多个域树之间建立信任关系。

9. 全局目录

对象的搜索可以跨越多个域。如果使用一些特性查找某个特定 `user` 对象，就必须搜索每个域。首先从 `wrox.com` 开始，接着搜索 `uk.wrox.com` 和 `india.wrox.com`，通过慢速链接，这样的搜索花费的时间会比较长。

要加快搜索速度，可以把所有对象都复制到全局目录 GC 中。GC 将被复制到森林的每个域中。

在每个域中至少有一个服务器包含 GC。出于性能和可伸缩性原因，在一个域中可以有多个 GC 服务器。使用 GC，可以在一个服务器上搜索所有对象。

GC 是所有对象的只读缓存，GC 只能用于搜索；必须使用域控制器更新。

并不是对象的所有特性都存储在 GC 中。可以定义特性是否和对象一起保存。特性是否存储在 GC 中，主要取决于该特性在搜索中使用的频率。如果特性在搜索中使用得很频繁，把它放在 GC 中，搜索就会比较快。用户的一幅图片在 GC 中不是很有用，因为我们从来不会搜索该图片。相反，在存储器中添加电话号码就比较有用。还可以定义该特性是否应被索引，如果进行索引，对该特性的查询就比较快。

10. 复制

程序员不喜欢配置复制，但因为会影响存储在 Active Directory 中的数据，所以必须了解它如何工作。Active Directory 使用了多主机服务器体系结构。域中的每个域控制器都可以进行更新。复制延迟时间定义了更新开始之前等待的时间。

- 如果某些特性发生变化，默认情况下，站点中每隔 5 分钟就发布一次变更通知，通知的内容是可配置的。因为发生改变的 DC 每隔 30 秒就通知另一个服务器，所以第 4 个 DC 可以在 7 分钟后得到变更通知。在默认情况下，把跨站点的变更通知设置为 180 分钟。站点之间和内部的复制可以配置为其他值。
- 如果没有发生改变，在站点内，每隔 60 分钟就进行一次预定复制。这将确保不遗漏一个变更通知。
- 对于敏感的安全信息，如账户被锁，会立即发出通知。

进行复制后，就只把改动的内容复制到 DC 中。在每次修改特性后，就会记录版本号(USN，更新序列号)和时间戳。如果更新不同服务器上的同一个特性，这就可以用于解决冲突。

下面看一个示例。用户 John Doe 的移动电话特性的 USN 号为 47。这个值已经复制到所有 DC 中。一个系统管理员改变电话号码。在服务器 DC1 上发生改变后；在服务器 DC1 上这个特性的新 USN 现在是 48，而其他域控制器的 USN 仍是 47。如果有人读取该特性，就会读取旧值，直到所有域控制器都进行复制为止。

下面的情况很少发生：另一个管理员改变电话号码特性，选择另一个域控制器，因为这个管理员从服务器 DC2 中接收到一个比较快的响应。在服务器 DC2 上，这个特性的 USN 也改为 48。

在通知的间隔期间，发出通知的原因是特性的 USN 改变，上次进行复制时，USN 的值是 47。使用复制机制可以检测到服务器 DC1 和 DC2 的电话号码特性的 USN 都是 48。虽然使用哪个服务器上的特性值并不重要，但必须使用其中一个服务器上的特性值。要解决这个冲突，就要使用改变的时间戳。因为 DC2 上的改变比较迟，所以会复制存储在 DC2 域控制器中的值。



在读取对象时，必须知道数据不一定是最新的。数据是否最新取决于复制延迟时间。在更新对象时，另一个用户可以在更新后仍获取旧值，同时还可能进行另一个更新操作。

52.1.3 Active Directory 数据的特征

假如 Active Directory 没有替代关系数据库或注册表，那么什么数据可以存储在 Active Directory 中？

- Active Directory 可以存储分层数据，容器也可以存储其他容器和对象。容器本身也是对象。
- 数据应主要用于读取。因为在一定的时间间隔中会进行复制，所以不能确定可以读取到最新的数据。在应用程序中，必须注意读取的信息有可能不是最新的信息。

- 数据应是企业普遍感兴趣的数据。因为给架构添加一个新数据类型，会把该数据类型复制到企业的所有服务器上。如果只有一小部分用户对该数据类型感兴趣，企业的域管理员就不会安装新的架构类型。
- 存储的数据量应合适，因为这些数据是要被复制的。如果存储数据量是 100KB，而且每星期仅修改一次数据，把它存储在目录中就不会出问题。但如果每小时修改一次数据，这个数据量就太大了。总是要考虑到数据复制到不同的服务器上、数据要传输到什么地方、复制的时间间隔等。如果数据量比较大，就要链接到 Active Directory 中，并把数据存储到另一个地方。总之，存储在 Active Directory 中的数据应分层组织，且数据量应合理，这对企业非常重要。

52.1.4 指定架构

Active Directory 对象是强类型化的。架构定义对象的类型、必选特性和可选特性，特性的语法和约束。如前所述，在架构中，必须区分架构对象和特性架构的对象。

类是特性的集合。通过类，就可以支持单一继承。从图 52-3 中可以看出，user 类派生自 organizationalPerson 类，而 organizationalPerson 类是 person 类的一个子类，它们的基类都是 top。classSchema 定义一个类，它用 systemMayContain 特性描述该特性。

图 52-3 只列出了几个 systemMayContain 值。使用 ADSI Edit 工具可以查看所有值。下一节将介绍这个工具。在 top 根类中，每个对象都有公共名称(cn)、displayName、objectGUID、whenChanged 和 whenCreated 特性。person 类派生自 top 类。person 对象也有 userPassword 和 telephoneNumber。organizationalPerson 派生自 person。除了 person 类的特性外，它还有 manager、department、company 特性，以及 user 登录系统所必需的其他特性。

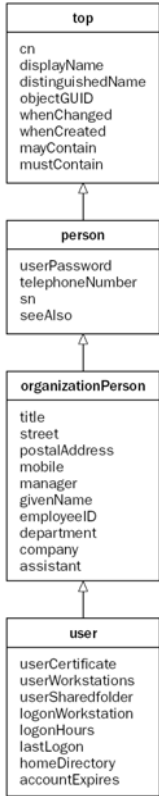


图 52-3

52.2 Active Directory 的管理工具

学习 Active Directory 的一些管理工具有助于理解 Active Directory、其中包含的数据，以及以编程方式可以完成的任务。

系统管理员可以用许多工具输入新数据，更新数据和配置 Active Directory。

- Active Directory Users and Computers MMC 插件用于更新用户数据、输入新用户。
- Active Directory Sites and Services MMC 插件用于配置域中的站点，在这些站点之间复制数据。
- Active Directory Domains and Trusts MMC 插件可以在树的域之间建立信任关系。
- ADSI Edit 是 Active Directory 的编辑器，可以在其中查看和编辑所有对象。



要在 Windows 7 上运行这些工具，需要安装 Windows 7 Remote Administration Tools。对于其他 Windows 版本，也需要单独下载这些系统管理工具。

下面几节介绍 Active Directory Users and Computers 和 ADSI Edit 工具的功能，因为这些工具对使用 Active Directory 创建应用程序非常重要。

52.2.1 Active Directory Users and Computers 工具

Active Directory Users and Computers 插件系统管理员用来管理用户的工具。选择 Start | Programs | Administrative | Tools | Active Directory Users and Computers 命令，就会启动这个程序，如图 52-4 所示。

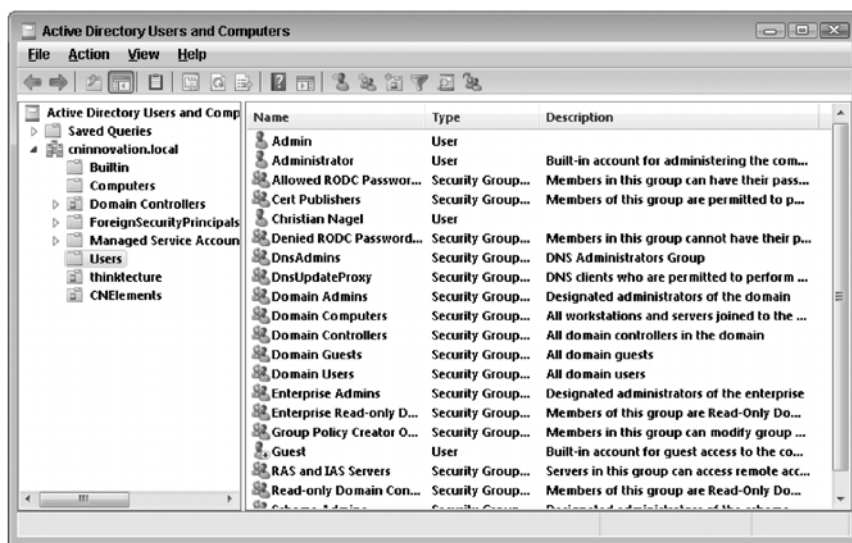


图 52-4

使用这个工具可以添加新用户、组、联系人、组织单元、打印机、共享文件夹或计算机，修改已有的项。在图 52-5 中，可以为 user 对象输入属性：办公室、电话号码、电子邮件地址、网页、公司信息、地址和组等。

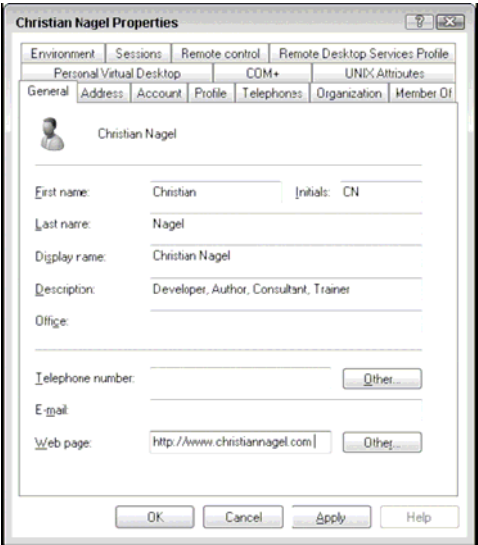


图 52-5

Active Directory Users and Computers 还可以用于有上百万对象的大公司。我们不必在有 1000 个对象的列表中查找，因为可以使用一个自定义筛选器，只显示某些对象。也可以执行 LDAP 查询，搜索公司中的对象。本章后面将讨论这些问题。

52.2.2 ADSI Edit 工具

ADSI Edit 是 Active Directory 的编辑器。如图 52-6 所示，ADSI Edit 提供的控制比 Active Directory Users and Computers 工具更多。使用 ADSI Edit，可以配置所有内容，也可以查看架构和对应配置。但使用这个工具并不是很容易，而且很容易输入错误的数据。

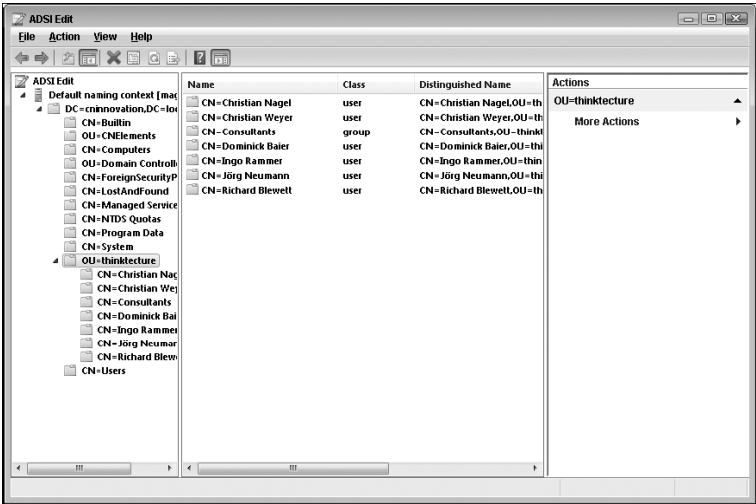


图 52-6

打开对象的 Properties 窗口，可以查看和修改 Active Directory 中对象的每个特性。利用该工具，可以查看必选特性、可选特性、特性的类型和值等，如图 52-7 所示。

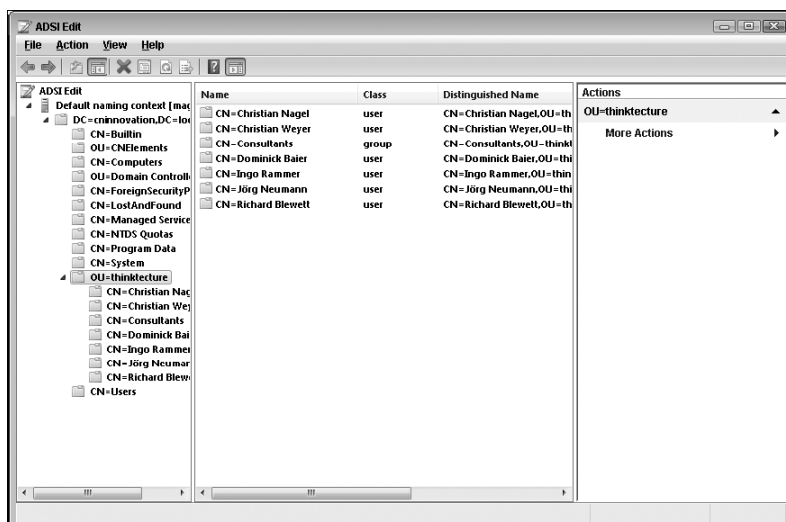


图 52-7

52.3 Active Directory 编程

要为 Active Director 开发程序, 可以使用 `System.DirectoryServices` 或 `System.Directory Services.Protocols` 名称空间中的类。在 `System.DirectoryServices` 名称空间中, 可以找到包装活动目录服务接口(Active Directory Service Interfaces, ADSI) COM 对象的类, 它们可以访问 Active Directory。

ADSI 是目录服务的一个编程接口。ADSI 定义一些由 ADSI 提供程序实现的 COM 接口。客户端可以在相同编程接口中使用不同目录服务。`System.DirectoryServices` 名称空间中的 .NET Framework 类可以使用 ADSI。

在图 52-8 中, 有一些实现 COM 接口(如 IADs 和 IUnknown)的 ADSI 提供程序(LDAP、IIS 和 NDS)。`System.DirectoryServices` 程序集使用 ADSI 提供程序。

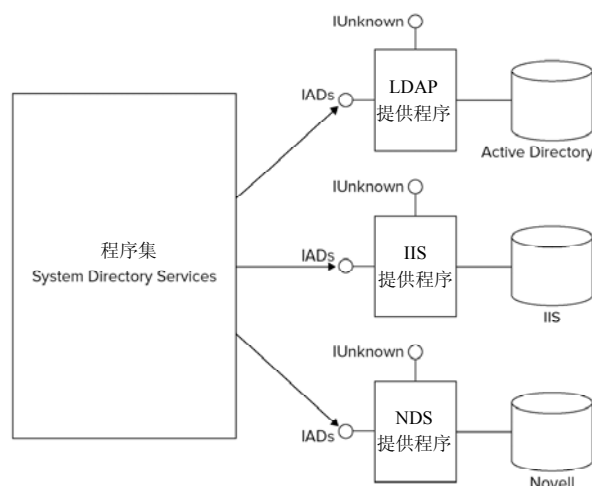


图 52-8

System.DirectoryServices.Protocols 名称空间中的类使用 Windows 的 DSML(Directory Services Markup Language, 目录服务标记语言)服务。OASIS 组(<http://www.oasis-open.org/committees/dsml>) 利用 DSML 定义了标准化的 Web 服务接口。

要使用 System.DirectoryServices 名称空间中的类, 必须引用 System.DirectoryServices 程序集。使用这个程序集中的类可以查询对象、查看和更新属性、搜索对象, 以及把对象移动到其他容器对象中。在本节后面的代码段中, 将使用一个简单的 C#控制台应用程序, 来说明 System.DirectoryServices 名称空间中的类的功能。

本节将介绍:

- System.DirectoryServices 名称空间中的类
- 连接 Active Directory 的处理方式: 绑定
- 获取目录项, 创建新对象, 并更新已有项
- 搜索 Active Directory

52.3.1 System.DirectoryServices 名称空间中的类

表 52-1 列出了 System.DirectoryServices 名称空间中的主要类。

表 52-1	
类	说 明
DirectoryEntry	这个类是 System.DirectoryServices 名称空间中的主类。这个类的对象表示 Active Directory 存储器中的一个对象。使用这个类可以绑定对象, 查看和更新属性。对象的属性都显示在 PropertyCollection 中。PropertyCollection 中的每一项都有一个 PropertyValueCollection
DirectoryEntries	DirectoryEntries 是 DirectoryEntry 对象的一个集合。DirectoryEntry 对象的 Children 属性返回 DirectoryEntries 集合中的一个对象列表
DirectorySearcher	这个类主要用于用指定的属性搜索对象。要定义该搜索, 可以使用 SortOption 类和 SearchScope、SortDirection 和 ReferralChasingOption 枚举。搜索的结果是一个 SearchResult 或 SearchResultCollection。也可以得到 ResultProperty Collection 和 ResultPropertyValueCollection 对象

52.3.2 绑定到 Directory Services

要获得 Active Directory 中一个对象的值, 必须连接到 Active Directory 服务。这个连接进程称为绑定。绑定路径如下:

LDAP://dc01.thinktecture.com/OU=Development, DC=Thinktecture, DC=Com

在绑定进程中, 可以指定如下项:

- 协议(protocol)指定要使用的提供程序
- 域控制器的服务器名(server name)
- 服务器进程的端口号(port number)
- 对象的可分辨名称(distinguished name), 这标识要访问的对象
- 如果允许访问 Active Directory 的用户不是当前登录的账户, 则提供用户名和密码

- 如果需要加密，应指定 authentication 类型。

下面详细介绍这些内容。

1. 协议

绑定路径(即协议)的第一部分指定 ADSI 提供程序。该提供程序作为一个 COM 服务器实现；对于标记，progID 可在注册表的 HKEY_CLASSES_ROOT 下。Windows 7 附带的提供程序如表 52-2 所示。

表 52-2

提 供 程 序	说 明
LDAP	LDAP 服务器，如 Exchange 目录和 Windows 2000 及以后版本的 Active Directory 服务器
GC	GC 用于访问 Active Directory 中的全局目录。它也可以用于快速查询
IIS	使用 IIS 的 ADSI 提供程序，可以在 IIS 目录中创建和管理新网站
NDS	这个 progID 用于和 Novell Directory Services 通信

2. 服务器名

在绑定路径中，服务器名在协议的后面。如果用户登录到 Active Directory 域上，服务器名就是可选的。如果不提供服务器名，就会发生无服务器绑定操作，此时 Windows Server 2008 会在域中查找与用户绑定过程相关的“最好的”域控制器。如果站点中没有服务器，就使用查找到的第一个域控制器。

无服务器绑定如下所示：

```
LDAP://OU=Sales, DC=Thinktecture, DC=Local
```

3. 端口号

在服务器名的后面，可以指定服务器进程的端口号，其语法是 xxx。LDAP 服务器的默认端口号是 389：

```
LDAP://dc01.sentinel.net:389
```

Exchange 服务器使用的端口号同 LDAP 服务器的端口号。如果在同一个系统上安装 Exchange 服务器，例如，用作 Active Directory 的域控制器，就可以配置另一个端口。

4. 可分辨名称

在路径中指定的第 4 部分是可分辨名称(Distinguished Name, DN)。DN 是一个唯一的名称，它标识要访问的对象。在 Active Directory 中，可以使用基于 X.500 的 LDAP 语法，指定对象的名称。

这是有一个可分辨名称：

```
CN=Christian Nagel, OU=Consultants, DC= Thinktecture, DC=local
```

这个 DN 指定 thinktecture.local 域中 thinktecture 域组件(Domain Component, DC)的 Consultants 组织单元(Organizational Unit, OU)的公共名称(Common Name, CN) Christian Nagel。最右边指定的

部分是域的根对象。该名称必须符合对象树中的层次结构。

可分辨名称的字符串表示的 LDAP 规范在 RFC 2253(<http://www.ietf.org/rfc/rfc2253.txt>)上。

(1) 相对可分辨名称

相对可分辨名称(RDN)用于引用容器对象中的对象。使用 RDN 时,不需要指定 OU 和 DC,因为一个公共名称就足够了。CN=Christian Nagel 就是组织单元中的一个相对可分辨名称。如果已经引用一个容器对象,并且要访问其子对象,就可以使用相对可分辨名称。

(2) 默认的命名上下文

如果在路径中没有指定可分辨名称,绑定进程就会使用默认的命名上下文。使用 rootDSE 可以读取默认命名上下文。LDAP 3.0 把 rootDSE 定义为目录服务器中目录树的根。例如

```
LDAP://rootDSE
```

或

```
LDAP://servername/rootDSE
```

通过枚举 rootDSE 的所有属性,将获得没有指定名称时使用的 defaultNamingContext 信息。schemaNamingContext 和 configurationNamingContext 指定用于访问 Active Directory 存储器中架构和配置所需要的名称。

通过下面的代码可获得 rootDSE 的所有属性:

```
try
{
    using (var de = new DirectoryEntry())
    {
        de.Path = "LDAP://magellan/rootDSE";
        de.Username = @"cninnovation\christian";
        de.Password = "Pa$$w0rd";

        PropertyCollection props = de.Properties;
        foreach (string prop in props.PropertyNames)
        {
            PropertyValueCollection values = props[prop];
            foreach (string val in values)
            {
                Console.WriteLine("{0}: ", prop);
                Console.WriteLine(val);
            }
        }
    }
}
catch (COMException ex)
{
    Console.WriteLine(ex.Message);
}
```

代码段 DirectoryServicesSamples/Program.cs



要在自己的计算机上运行这段代码,必须更改要访问的对象的`路径`,使之包含服务器名。

这个程序显示默认的命名上下文(defaultNamingContextDC=explorer、DC=local), 用于访问架构的上下文(CN=Schema、CN=Configuration、DC=cninnovation、DC=local)和上下文的命名环境(CN=Configuration、DC=cninnovation、DC=local), 如下所示。

```
currentTime: 20090925131508.0Z
subschemaSubentry: CN=Aggregate,CN=Schema,CN=Configuration,DC=cninnovation,
DC=local
dsServiceName: CN=NTDS Settings,CN=MAGELLAN,CN=Servers,
CN=Default-First-Site-Name,CN= Sites,CN=Configuration,DC=cninnovation,DC=local
namingContexts: DC=cninnovation,DC=local
namingContexts: CN=Configuration,DC=cninnovation,DC=local
namingContexts: CN=Schema,CN=Configuration,DC=cninnovation,DC=local
namingContexts: DC=DomainDnsZones,DC=cninnovation,DC=local
namingContexts: DC=ForestDnsZones,DC=cninnovation,DC=local
defaultNamingContext: DC=cninnovation,DC=local
schemaNamingContext: CN=Schema,CN=Configuration,DC=cninnovation,DC=local
configurationNamingContext: CN=Configuration,DC=cninnovation,DC=local
rootDomainNamingContext: DC=cninnovation,DC=local
supportedControl: 1.2.840.113556.1.4.319
supportedControl: 1.2.840.113556.1.4.801
```

(3) 对象标识符

每个对象都有一个全局唯一的标识符 GUID。GUID 是一个唯一的 128 位数字, 您可能已经从 COM 开发中了解了它。使用 GUID 可以绑定一个对象。这样, 不管对象是否移动到另一个容器中, 也可以得到同一个对象。GUID 在创建对象时生成, 且总是保持不变。

使用 DirectoryEntry.NativeGuid 可以得到 GUID 的字符串表示。这个字符串表示就可以用于绑定对象。

下面的示例显示一个无服务器绑定的路径名, 它绑定到 GUID 表示的一个特定对象上:

```
LDAP://<GUID=14abbd652aae1a47abc60782dcfc78ea>
```

5. 用户名

如果必须使用另一个用户名访问目录(也许这个用户没有访问 Active Directory 所必需的权限), 此时必须为绑定进程指定显式用户证书(user credential)。Active Directory 提供许多方式来指定用户名。

(1) Downlevel 登录

使用 downlevel 登录, 用户名可以用 Windows 2000 以前的域名指定:

```
domain\username
```

(2) 可分辨名称

也可以用 user 对象的可分辨名称来指定用户, 例如:

```
CN=Administrator, CN=Users, DC=thinkecture, DC=local
```

(3) 用户主体名 (UPN)

对象的 UPN 用 userPrincipalName 属性定义。系统管理员可以在 Active Directory Users and Computers 工具中 User 属性的 Account 选项卡上, 用登录信息指定 UPN, 注意, 这不是用户的电子邮件地址。

这些信息也唯一地标识了用户，可以用于登录：

```
Nagel@ thinktecture.local
```

6. 身份验证

对于安全加密身份验证，也可以指定身份验证(authentication)类型。身份验证可以用 `DirectoryEntry` 类的 `AuthenticationType` 属性设置。可以指定其值为 `AuthenticationTypes` 枚举中的一个值。因为枚举用 [Flags] 特性标记，所以可以指定多个值。发送的数据进行加密的可能值有 `ReadonlyServer` 和 `Secure`，`ReadonlyServer` 指定只需要读取访问；`Secure` 表示安全身份验证。

7. 用 DirectoryEntry 类绑定

`System.DirectoryServices.DirectoryEntry` 类可以用于指定所有绑定信息。可以使用默认的构造函数，并用 `Path`、`Username`、`Password` 和 `AuthenticationType` 属性定义绑定信息，或者把这些信息传递给构造函数：

```
DirectoryEntry de = new DirectoryEntry();
var de = new DirectoryEntry();
de.Path = "LDAP://platinum/DC=thinktecture, DC=local";
de.Username = "Christian.Nagel@thinktecture.local";
de.Password = "password";

// use the current user credentials
var de2 = new DirectoryEntry("LDAP://DC=thinktecture, DC=local");
```

即使构造 `DirectoryEntry` 对象成功，也并不意味着绑定成功。在第一次读取属性时进行绑定，可以避免不必要的网络流量。对象是否存在，或者指定的用户证书是否正确，都可以在第一次访问该对象时确定。

52.3.3 获取目录项

既然知道了如何指定 Active Directory 中对象的绑定特性，就可以转向读取对象的特性。在下面的示例中，读取用户对象的属性。

`DirectoryEntry` 类的一些属性可以提供对象的信息，即 `Name`、`Guid` 和 `SchemaClassName` 属性。第一次访问 `DirectoryEntry` 对象的属性时，会执行绑定操作，并填充基础 ADSI 对象的缓存。后面将详细讨论这些。其他属性可以从缓存中读取，同一对象的数据不需要通过与服务器的通信来获得。

在本例中，用 `thinktecture` 组织单元中的公共名称 `Christian Nagel` 访问 `user` 对象：



可从
wrox.com
下载源代码

```
using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/CN=Christian Nagel, " +
              "OU=thinktecture, DC=cninnovation, DC=local";
    Console.WriteLine("Name: {0}", de.Name);
    Console.WriteLine("GUID: {0}", de.Guid);
    Console.WriteLine("Type: {0}", de.SchemaClassName);
    Console.WriteLine();

    //...
}
```

代码段 `DirectoryServicesSamples/Program.cs`

Active Directory 对象包含许多信息，可用信息取决于对象的类型。Properties 属性返回一个 PropertyCollection。每个属性本身就是一个集合，因为一个属性可以有多个值；例如，user 对象可以有多个电话号码。在本例中，用一个内层 foreach 循环查看这些值。从 properties[name]返回的集合是一个 object 数组。特性值可以是字符串、数字或其他类型。使用 ToString()方法就可以显示这些值：

```
Console.WriteLine("Properties: ");
PropertyCollection properties = de.Properties;
foreach (string name in properties.PropertyNames)
{
    foreach (object o in properties[name])
    {
        Console.WriteLine("{0}: {1}", name, o.ToString());
    }
}
```

在最终输出中，可以查看 user 对象指定的所有特性。一些属性(如 otherTelephone)有多个值，通过该属性，可以定义许多电话号码。一些属性值只显示 System.__ComObject 对象的类型，例如，lastLogoff、lastLogon 和 nTSecurityDescriptor。要得到这些特性的值，必须直接使用 System.DirectoryServices 名称空间的类中的 ADSI COM 接口。

```
Name: CN=Christian Nagel
GUID: 0238fd5c-7e67-48bc-985f-c2f1ccf0f86c
Type: user

Properties:
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: user
cn: Christian Nagel
sn: Nagel
givenName: Christian
distinguishedName: CN=Christian Nagel,OU=thinktecture,DC=cninnovation,DC=local
instanceType: 4
whenCreated: 9/25/2009 12:42:05 PM
whenChanged: 9/25/2009 12:42:05 PM
displayName: Christian Nagel
uSNCreated: System.__ComObject
uSNChanged: System.__ComObject
name: Christian Nagel
objectGUID: System.Byte[]
userAccountControl: 66048
badPwdCount: 0
codePage: 0
countryCode: 0
badPasswordTime: System.__ComObject
lastLogoff: System.__ComObject
lastLogon: System.__ComObject
pwdLastSet: System.__ComObject
primaryGroupID: 513
objectSid: System.Byte[]
accountExpires: System.__ComObject
logonCount: 0
```

```

sAMAccountName: christian.nagel
sAMAccountType: 805306368
userPrincipalName: christian.nagel@cninnovation.local
objectCategory: CN=Person,CN=Schema,CN=Configuration,DC=cninnovation,DC=local
dSCorePropagationData: 1/1/1601 12:00:00 AM
nTSecurityDescriptor: System.__ComObject

```

使用 `DirectoryEntry.Properties` 可以访问所有属性。如果已知属性名，就可以直接访问其值：

```

foreach (string homePage in de.Properties["WWWHomePage"])
    Console.WriteLine("Home page: " + homePage);

```

52.3.4 对象集合

对象分层存储在 Active Directory 中。容器对象包含子对象。使用 `DirectoryEntry` 类的 `Children` 属性可以枚举容器中的子对象。另一方面，使用 `Parent` 属性可以得到对象的容器。

因为 `user` 对象没有子对象，所以下面的示例使用一个组织单元。非容器对象用 `Children` 属性返回一个空集合。下面从 `explore.local` 域 `thinktecture` 组织单元中获得所有 `user` 对象。`Children` 属性返回一个 `DirectoryEntries` 集合，其中包含 `DirectoryEntry` 对象。遍历所有 `DirectoryEntry` 对象，显示子对象的名称。



```

using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/OU=thinktecture, DC=cninnovation, DC=local";

    Console.WriteLine("Children of {0}", de.Name);
    foreach (DirectoryEntry obj in de.Children)
    {
        Console.WriteLine(obj.Name);
    }
}

```

代码段 `DirectoryServicesSamples/Program.cs`

运行程序，会显示对象的公共名称：

```

Children of OU=thinktecture
OU=Admin
CN=Buddhike de Silva
CN=Christian Nagel
CN=Christian Weyer
CN=Consultants
CN=demos
CN=Dominick Baier
CN=Ingo Rammer
CN=Neno Loye

```

本例显示了组织单元中的所有对象：`users`、`contacts`、`printers`、`shares` 和其他对象。如果只需要查看某些对象类型，就可以使用 `DirectoryEntries` 类的 `SchemaFilter` 属性。`SchemaFilter` 属性返回一个 `SchemaNameCollection`。通过这个 `SchemaNameCollection`，就可以使用 `Add()` 方法定义要查看的对象类型。在本例中，因为只需要查看 `user` 对象，所以把 `user` 添加到这个集合中：

```

using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/OU=thinktecture, DC=cninnovation, DC=local";

    Console.WriteLine("Children of {0}", de.Name);
    de.Children.SchemaFilter.Add("user");
    foreach (DirectoryEntry obj in de.Children)
    {
        Console.WriteLine(obj.Name);
    }
}

```

结果, 只显示组织单元中的 user 对象, 如下所示。

```

Children of OU=thinktecture
CN=Christian Nagel
CN=Christian Weyer
CN=Dominick Baier
CN=Ingo Rammer
CN=Jörg Neumann
CN=Richard Blewett

```

52.3.5 缓存

为了减少网络流量, ADSI 使用缓存来存储对象属性。如前所述, 在创建 DirectoryEntry 对象时不访问服务器。只要从目录存储器中读取第一个属性, 所有属性都会写入缓存中, 这样, 在访问下一个属性时, 就不需要往返服务器。

给对象写入任何更改内容时, 只会改变已缓存的对象。设置属性不会产生网络流量。必须使用 DirectoryEntry.CommitChanges() 方法刷新缓存, 把任何已更改的数据传输到服务器。要从目录存储器中获取新写入的数据, 可以使用 DirectoryEntry.RefreshCache() 读取属性。当然, 如果没有调用 CommitChanges() 和 RefreshCache() 方法更改一些属性, 那么所有更改都会丢失, 因为我们将再次使用 RefreshCache() 方法读取目录服务中的值。

把 DirectoryEntry.UsePropertyCache 属性设置为 false, 就可以关闭这个属性缓存。但除非正在调试代码, 否则最好不要关闭缓存, 因为这会与服务器间产生额外的往返。

52.3.6 创建新对象

创建新 Active Directory 对象时, 如 users、computers、printers 和 contacts 等, 可以使用 DirectoryEntries 类以编程方式完成创建工作。

要给目录添加新对象, 首先必须绑定一个容器对象, 如组织单元, 可以在其中插入新对象。不能使用不包含其他对象的对象。下面的示例使用一个容器对象, 其可分辨名称为 CN=Users, DC=thinktecture, DC=local:



可从
wrox.com
下载源代码

```

var de = new DirectoryEntry();
de.Path = "LDAP://magellan/CN=Users, DC=cninnovation, DC=local";

```

代码段 DirectoryServicesSamples/Program.cs

使用 DirectoryEntry 类的 Children 属性，可以得到 DirectoryEntries 对象：

```
DirectoryEntries users = de.Children;
```

DirectoryEntries 类提供的方法可以添加、删除和查找集合中的对象。下面创建一个新的 user 对象。使用 Add()方法时，需要一个对象名和一个类型名。使用 ADSI Edit 可以直接获得类型名：

```
DirectoryEntry user = users.Add("CN=John Doe", "user");
```

对象现在有默认属性值。要指定特定属性值，可以使用 Properties 属性的 Add()方法添加属性。当然，所有属性都必须存在于 user 对象的架构中。如果指定的属性不存在，就得到一个 COMException 异常：“指定的目录服务属性或值不存在”：

```
user.Properties["company"].Add("Some Company");
user.Properties["department"].Add("Sales");
user.Properties["employeeID"].Add("4711");
user.Properties["samAccountName"].Add("JDoe");
user.Properties["userPrincipalName"].Add("JDoe@explorer.local");
user.Properties["givenName"].Add("John");
user.Properties["sn"].Add("Doe");
user.Properties["userPassword"].Add("someSecret");
```

最后，为了把数据写入 Active Directory 中，必须刷新缓存：

```
user.CommitChanges();
```

52.3.7 更新目录项

Active Directory 服务中对象的更新和读取一样简单。可以在读取对象后更改它们的值。要删除一个属性的所有值，可以调用 PropertyValueCollection.Clear()方法。使用 Add()，可以把新值添加到属性中。Remove()和 RemoveAt()方法可以从属性集合中删除指定的值。

要更改一个值，只需把这个值设置为指定的值。下面的示例通过 PropertyValueCollection 类的一个索引器把移动电话号码设置为一个新值。使用该索引器只能改变已存在的值。因此，应总是使用 DirectoryEntry.Properties.Contains()方法来确定该特性是否可用。

```
using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/CN=Christian Nagel, " +
              "OU=thinkecture, DC=cninnovation, DC=local";

    if (de.Properties.Contains("mobile"))
    {
        de.Properties["mobile"][0] = "+43(664)3434343434";
    }
    else
    {
        de.Properties["mobile"].Add("+43(664)3434343434");
    }

    de.CommitChanges();
}
```

在本例的 `else` 部分，如果移动电话号码不存在新属性，就用 `PropertyValue Collection.Add()` 方法添加它。如果对已有的属性使用 `Add()` 方法，那么最终结果将取决于属性的类型(单值属性或多值属性)。对已有的单值属性使用 `Add()` 方法，会得到一个 `COMException` 异常：“违反了一个约束。”对已有的多值属性使用 `Add()` 方法，则会成功地把另一个值添加到属性中。

因为把 `user` 对象的属性 `mobile` 定义为单值属性，所以不能添加其他移动电话号码。但是用户可以有多个移动电话号码。对于多个移动电话号码，`otherMobile` 属性可用。因为它是一个多值属性，允许设置多个移动电话号码，所以可以多次调用 `Add()` 方法。注意，对多值属性要检查其唯一性。如果把第二个电话号码添加到同一个 `user` 对象中，那么也会得到一个 `COMException` 异常：“指定的目录服务属性或值已经存在。”



在创建或更新目录对象后，应调用 `DirectoryEntry.CommitChanges()` 方法。否则，只能更新缓存中的信息，更改的信息不会发送到目录服务中。

52.3.8 访问本地 ADSI 对象

调用预定义的 ADSI 接口的方法常常比搜索对象的属性名容易得多。一些 ADSI 对象还支持不能从 `DirectoryEntry` 类中直接使用的方法。例如，`IADsServiceOperations` 接口有启动和停止 Windows 服务的方法。Windows 服务的详细内容请参见第 25 章。

如前所述，`System.DirectoryServices` 名称空间的类使用基础 ADSI COM 对象。`DirectoryEntry` 类支持直接使用 `Invoke()` 方法调用基础对象的主调方法。

`Invoke()` 方法的第一个参数是应在 ADSI 对象中调用的方法名，第二个参数的 `params` 关键字允许把数量可变的其他参数传递给 ADSI 方法：

```
public object Invoke(string methodName, params object[] args);
```

ADSI 文档介绍了可以用 `Invoke()` 方法调用的方法。域中的每个对象都支持 `IADs` 接口的方法。前面创建的 `user` 对象也支持 `IADsUser` 接口的方法。

在下面的代码示例中，使用 `IADsUser.SetPassword()` 方法改变前面创建的 `user` 对象的密码：

```
using (var de = new DirectoryEntry())
{
    de.Path = "LDAP://magellan/CN=John Doe, CN=Users, DC=cninnovation, DC=local";

    de.Invoke("SetPassword", "anotherSecret");
    de.CommitChanges();
}
```

如果不使用 `Invoke()` 方法，那么还可以直接使用基础 ADSI 对象。要使用这些对象，必须选择 `Project | Add Reference` 命令添加对 `Active DS Type Library` 的引用，如图 52-9 所示。这会创建一个包装器类，在该类中可以访问 `ActiveDs` 名称空间中的这些对象。

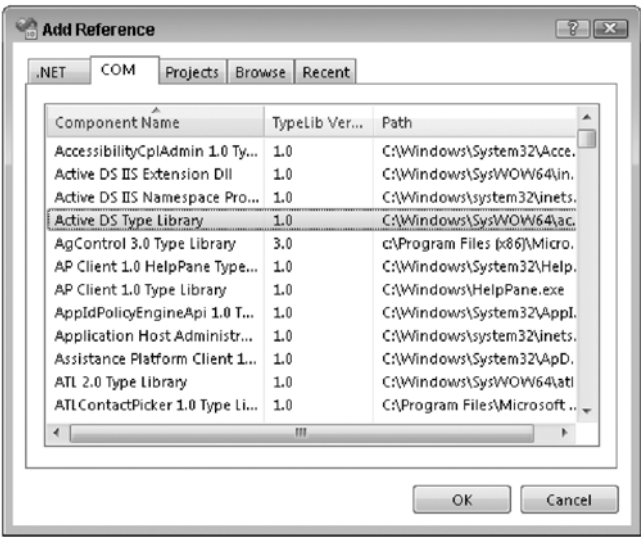


图 52-9

本地对象可以使用 `DirectoryEntry` 类的 `NativeObject` 属性访问。在下面的示例中，因为 `de` 对象是一个 `user` 对象，所以可以把它强制转换为 `ActiveDs.IADsUser` 类型。`SetPassword()` 方法是在 `IADsUser` 接口中归档的方法，所以可以直接调用它，而不是使用 `Invoke()` 方法调用它。把 `IADsUser` 接口的 `AccountDisabled` 属性设置为 `false`，就可以启用账户。与前面的示例一样，调用 `CommitChanges()` 方法和 `DirectoryEntry` 对象，把更改的内容写入目录服务中：

```
ActiveDs.IADsUser user = (ActiveDs.IADsUser)de.NativeObject;
user.SetPassword("someSecret");
user.AccountDisabled = false;
de.CommitChanges();
```



从 .NET 3.5 开始，不需要在 .NET 类 `DirectoryEntry` 的后面调用本地对象。可以使用 `System.DirectoryServices.AccountManagement` 名称空间中的类管理用户。这个名称空间中的类参见本章后面的内容。

52.3.9 在 Active Directory 中搜索

因为 Active Directory 是一个数据存储器，它为大多数读取访问进行了优化，所以一般用来搜索一些值。要在 Active Directory 中搜索，可以使用 .NET Framework 中的 `DirectorySearcher` 类。



`DirectorySearcher` 类只能和 LDAP 提供程序一起使用；它不能和 NDS 或 IIS 等提供程序一起使用。

在 `DirectorySearcher` 类的构造函数中，可以定义搜索的 4 个重要部分：搜索开始的根目录、筛选、应加载的属性和搜索的范围。也可以使用默认构造函数，并用属性定义搜索选项。

1. SearchRoot

搜索根目录指定搜索应从什么地方开始。SearchRoot 的默认值是当前使用的域的根。SearchRoot 用 DirectoryEntry 对象的 Path 指定。

2. 筛选器

筛选器定义希望获得的值。筛选器是一个必须放在圆括号中的字符串。

表达式中可以使用关系运算符，如 <=、=、>=。(objectClass=contact)搜索所有为 contact 类型的对象；(lastName>=Nagel)按字母顺序搜索 lastName 属性等于或大于 Nagel 的所有对象。

表达式可以和前缀运算符“&”和“|”组合使用。例如，(&(objectClass=user)(description=Auth*))搜索类型为 user，其 description 属性以字符串 Auth 开头的所有对象。因为“&”和“|”运算符在表达式的开头，所以可以把两个以上的表达式用一个前缀运算符组合在一起。

因为默认筛选器是(objectClass=*)，所以所有对象有效。



筛选器语法在 RFC 2254 “LDAP 搜索筛选器的字符串表示”中定义。这个 RFC 在 <http://www.ietf.org/rfc/rfc2254.txt> 中。

3. PropertiesToLoad

使用 PropertiesToLoad，可以定义需要的所有属性的一个 StringCollection。对象可以有許多属性，其中大多数对于搜索请求都不太重要。我们定义了应加载到缓存中的属性。如果没有指定属性返回的，默认属性就是对象的 Path 和 Name 属性。

4. SearchScope

SearchScope 是一个枚举，它定义搜索应扩展的深度：

- 因为 SearchScope.Base 只搜索开始搜索的对象中的特性，所以至多可以得到一个对象。
- SearchScope.OneLevel 表示在基对象的子集合中继续搜索。不搜索基对象本身。
- SearchScope.Subtree 定义在整个树中向下搜索。

SearchScope 属性的默认值是 SearchScope.Subtree。

5. 搜索的限制

在目录服务中搜索特定的对象可以跨几个域进行。要限制搜索对象的个数或搜索时间，可以定义其他几个属性(如表 52-3 所示)。

表 52-3

属 性	说 明
ClientTimeout	客户端等待服务器返回结果的最长时间。如果服务器没有响应，就不返回记录
PageSize	使用 paged search，服务器会返回用 PageSize 定义的许多对象，而不是所有对象。这会减少客户端获得第一个应答的时间和需要的内存。服务器把一个 cookie 发送给客户端，客户端再把下一个搜索请求发送回服务器，这样搜索就可以在上一次结束的地方继续进行

(续表)

属 性	说 明
ServerPageTimeLimit	对于 paged search, 这个值定义了一个搜索时间范围, 在该时间范围内应继续返回用 PageSize 值定义的许多对象。如果这段时间在 PageSize 值之前到达, 就会把此时查找到的对象返回给客户端。默认值为 - 1, 表示时间为无限长
SizeLimit	定义搜索返回的最大对象个数。如果把它设置为比服务器定义的值(1000)还大, 就使用服务器定义的限制
ServerTimeLimit	定义服务器搜索对象的最长时间。过了这段时间后, 就会把此时查找到的所有对象返回给客户端。默认值是 120 秒, 不能把搜索时间设置为较高的值
ReferralChasing	搜索可以跨几个域进行。如果用 SearchRoot 指定的根目录是一个父域或没有指定根目录, 就会继续在子域中搜索。使用这个属性可以指定是否应在不同的服务器上继续搜索。 ReferralChasingOption.None 表示不在不同的服务器上继续搜索。 ReferralChasingOption.Subordinate 指定继续在子域中搜索。当搜索从 DC=Wrox、DC=COM 开始时, 服务器可以返回一个结果集, 并引用 DC=France、DC=Wrox、DC=COM。客户端可以继续在于域中搜索。 ReferralChasingOption.External 表示服务器可以使客户机搜索不在子域中的另一个独立服务器, 这是默认选项。 ReferralChasingOption.All 表示返回外部引用和从属引用
Tombstone	如果将属性 Tombstone 设置为 true, 就返回所有匹配搜索的已删除对象
VirtualListView	如果搜索结果比较多, 就可以使用 VirtualListView 属性定义一个应从搜索中返回的子集。这个子集用 DirectoryVirtualListView 类定义

在搜索示例中, 要搜索 thinktecture 组织单元中 description 属性值为 Author 的所有 user 对象。

首先, 绑定 thinktecture 组织单元, 从该组织单元中开始搜索。创建一个 DirectorySearcher 对象, 在其中设置 SearchRoot。把筛选器定义为(&(objectClass=user)(description=Auth*)), 这样就可以搜索所有类型为 user、description 属性以 Auth 开头的对象。搜索的范围应在一个子树中, 这样在 thinktecture 的子组织单元中搜索:



可从
wrox.com
下载源代码

```
using (var de = new DirectoryEntry("LDAP://OU=thinktecture, DC=cninnovation, DC=local"))
using (var searcher = new DirectorySearcher())
{
    searcher.SearchRoot = de;
    searcher.Filter = "( &(objectClass=user)(description=Auth*))";
    searcher.SearchScope = SearchScope.Subtree;
```

代码段 DirectoryServicesSamples/Program.cs

要在搜索结果中包含的属性有 name、description、givenName 和 wWWHomePage。

```
searcher.PropertiesToLoad.Add("name");
searcher.PropertiesToLoad.Add("description");
searcher.PropertiesToLoad.Add("givenName");
searcher.PropertiesToLoad.Add("wWWHomePage");
```

下面准备开始搜索。还应对结果进行排序。DirectorySearcher 类有一个 Sort 属性，它可以设置一个 SortOption。SortOption 类的构造函数的第一个参数定义用于排序的属性，第二个参数定义排序的方向。SortDirection 枚举的值有 Ascending 和 Descending。

要开始搜索，可以使用 FindOne()方法查找第一个对象，或者使用 FindAll()方法。FindOne()方法返回一个简单的 SearchResult，而 FindAll()方法返回一个 SearchResultCollection。因为要返回所有作者，所以应使用 FindAll()方法：

```
searcher.Sort = new SortOption("givenName", SortDirection.Ascending);
SearchResultCollection results = searcher.FindAll();
```

使用一个 foreach 循环，可以访问 SearchResultCollection 中的每个 SearchResult。SearchResult 表示搜索缓存中的一个对象。Properties 属性返回一个 ResultPropertyCollection，其中使用属性名和索引器可以访问该集合中的所有属性：

```
SearchResultCollection results = searcher.FindAll();

foreach (SearchResult result in results)
{
    ResultPropertyCollection props = result.Properties;
    foreach (string propName in props.PropertyNames)
    {
        Console.WriteLine("{0}: ", propName);
        Console.WriteLine(props[propName][0]);
    }
    Console.WriteLine();
}
```

也可以在搜索之后获得完整的对象：SearchResult 有一个 GetDirectoryEntry()方法，它返回查找到的对象的相应 DirectoryEntry。

最终的输出应显示所有 thinktecture 对应的列表的开始，以及已选择的属性。

```
givenname: Christian
adspath: LDAP://magellan/CN=Christian Weyer,OU=thinktecture,DC=cninnovation,
DC=local
description: Author
name: Christian Weyer

givenname: Christian
adspath: LDAP://magellan/CN=Christian Nagel,OU=thinktecture,DC=cninnovation,
DC=local
description: Author
name: Christian Nagel

givenname: Dominick
adspath: LDAP://magellan/CN=Dominick Baier,OU=thinktecture,DC=cninnovation,
DC=local
description: Author
name: Dominick Baier

givenname: Ingo
adspath: LDAP://magellan/CN=Ingo Rammer,OU=thinktecture,DC=cninnovation,
DC=local
```

```
description: Author
name: Ingo Rammer

givenname: Jörg
adspath: LDAP://magellan/CN=Jörg Neumann,OU=thinktecture,DC=cninnovation,
DC=local
description: Author
name: Jörg Neumann
```

52.4 搜索用户对象

本节要创建一个 WPF 应用程序 UserSearch。这个应用程序非常灵活，因为可以输入特定域控制器、用户名和密码，访问 Active Directory 或者使用正在运行的进程的用户。在这个应用程序中，我们将访问 Active Directory 服务的架构，获得 user 对象的属性。该用户可以输入一个筛选器字符串，搜索域中的所有 user 对象。还可以设置应显示的 user 对象的属性。

52.4.1 用户界面

用户界面显示一些已编号的步骤，说明如何使用该应用程序(如图 52-10 所示)。

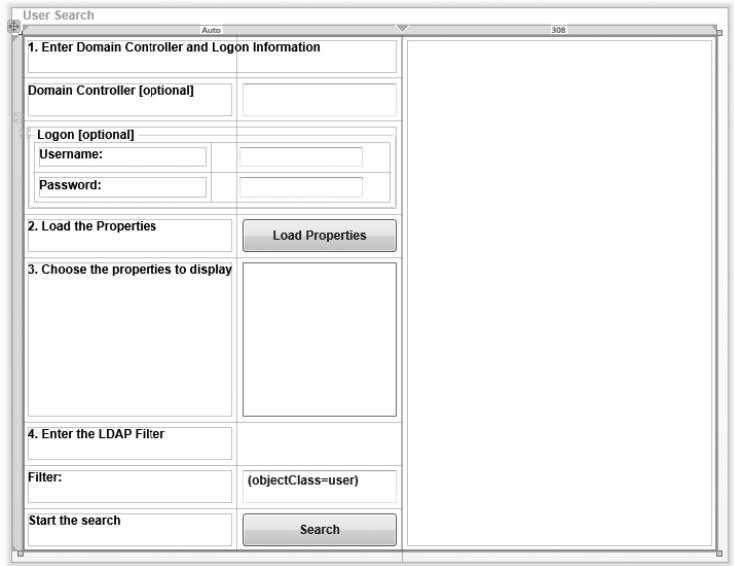


图 52-10

1. 输入用户名、密码和域控制器。所有这些信息都是可选的。如果没有输入域控制器，就使用无服务器绑定进行连接。如果没有输入用户名，就使用当前用户的安全上下文。
2. 使用一个按钮，就可以把 user 对象的所有属性名动态地加载到 listBoxProperties 列表框中。
3. 选择要显示的属性。把列表框的 SelectionMode 设置为 MultiSimple。
4. 可以输入限制搜索的筛选器，在该对话框中设置的默认值搜索所有 user 对象：(objectClass=user)。
5. 现在开始搜索。

52.4.2 获取架构命名上下文

这个应用程序只有两个处理程序方法：按钮的第一个方法加载属性，第二个方法则在域中开始搜索操作。首先，从架构中动态地读取 `user` 类的属性，在用户界面中显示它。

在 `buttonLoadProperties_Click()` 方法中，使用 `SetLogonInformation()` 方法从对话框中读取用户名、密码和主机名，并把它们存储在类的成员中。接着，`SetNamingContext()` 方法设置架构的 LDAP 名称和默认上下文的 LDAP 名称。这个架构的 LDAP 名称用于 `SetUserProperties()` 调用中，设置列表框中的属性。



```
private void OnLoadProperties(object sender, RoutedEventArgs e)
{
    try
    {
        SetLogonInformation();

        SetNamingContext();

        SetUserProperties(schemaNamingContext);
    }
    catch (Exception ex)
    {
        MessageBox.Show(String.Format("check your input! {0}", ex.Message));
    }
}
```

代码段 UserSearch/MainWindow.xaml.cs

在辅助方法 `SetNamingContext()` 中，使用目录树的根目录来获得服务器的属性。这里只考虑两个属性的值：`schemaNamingContext` 和 `defaultNamingContext`。

```
private void SetNamingContext()
{
    using (var de = new DirectoryEntry())
    {
        string path = "LDAP://" + hostname + "rootDSE";
        de.Username = username;
        de.Password = password;
        de.Path = path;
        schemaNamingContext = de.Properties["schemaNamingContext"][0].ToString();
        defaultNamingContext = de.Properties["defaultNamingContext"][0].ToString();
    }
}
```

52.4.3 获取 User 类的属性名

使用 LDAP 名称可以访问架构。使用它可以访问目录，并读取属性。我们不仅要介绍 `user` 类的属性，还将介绍 `user` 类的基类：`Organizational-Person`、`Person` 和 `Top`。在这个程序中，基类的名称是硬编码的。还可以使用 `subClassOf` 属性动态地读取基类。

`GetSchemaProperties()` 方法返回 `IEnumerable<string>`，其中包含指定对象类型的所有属性名。把所有属性名都添加到列表框中：



可从
wrox.com
下载源代码

```
private void SetUserProperties(string schemaNamingContext)
{
    var properties = from p in GetSchemaProperties(schemaNamingContext,
                                                    "User").Concat(
                                                    GetSchemaProperties(schemaNamingContext,
                                                    "Organizational-Person")).Concat(
                                                    GetSchemaProperties(schemaNamingContext, "Top"))
                    orderby p
                    select p;
    listBoxProperties.DataContext = properties;
}
```

代码段 UserSearch/MainWindow.xaml.cs

在 GetSchemaProperties()方法中, 再次访问 Active Directory 服务。这次不使用 rootDSE, 而使用前面介绍的架构的 LDAP 名称。systemMayContain 属性包含 objectType 类中的所有属性的一个集合:

```
private IEnumerable<string>GetSchemaProperties(string schemaNamingContext,
                                              string objectType)
{
    IEnumerable<string>data;
    using (var de = new DirectoryEntry())
    {
        de.Username = username;
        de.Password = password;

        de.Path = String.Format("LDAP://{0}CN={1},{2}", hostname, objectType,
                                schemaNamingContext);

        PropertyValueCollection values = de.Properties["systemMayContain"];
        data = from s in values.Cast<string>()
              orderby s
              select s;
    }
    return data;
}
```

这样就完成了应用程序的第二步。Listbox 控件包含 user 对象的所有属性名。

52.4.4 搜索用户对象

搜索按钮的处理程序只调用辅助方法 FillResult();



可从
wrox.com
下载源代码

```
private void OnSearch(object sender, RoutedEventArgs e)
{
    try
    {
        FillResult();
    }
    catch (Exception ex)
    {
        MessageBox.Show(String.Format("check your input! {0}", ex.Message));
    }
}
```

代码段 UserSearch/MainWindow.xaml.cs

在 FillResult()方法中,在整个 Active Directory 域中进行与前面一样的正常搜索。把 SearchScope 设置为 Subtree,把 Filter 设置为从 TextBox 获取中的字符串,应加载到缓存中的属性由用户在列表框中选择的值设置。DirectorySearcher 类的 PropertiesToLoad 属性是 StringCollection 类型,其中应加载的属性可以使用 AddRange()方法添加,该方法的参数是一个字符串数组。应加载的属性用 SelectedItems 属性从 listBoxProperties 列表框中读取。在设置 DirectorySearcher 对象的属性后,就调用 SearchAll()方法搜索属性。SearchResultCollection 中的搜索结果用于生成写入 textBoxResults 文本框中的汇总信息。

```
private void FillResult()
{
    using (var root = new DirectoryEntry())
    {
        root.Username = username;
        root.Password = password;
        root.Path = String.Format("LDAP://{0}/{1}",
                                hostname, defaultNamingContext);
        using (var searcher = new DirectorySearcher())
        {
            searcher.SearchRoot = root;
            searcher.SearchScope = SearchScope.Subtree;
            searcher.Filter = textFilter.Text;
            searcher.PropertiesToLoad.AddRange(
                listBoxProperties.SelectedItems.Cast < string > ().ToArray());

            SearchResultCollection results = searcher.FindAll();
            var summary = new StringBuilder();
            foreach (SearchResult result in results)
            {
                foreach (string propName in result.Properties.PropertyNames)
                {
                    foreach (object p in result.Properties[propName])
                    {
                        summary.AppendFormat(" {0}: {1}", propName, p);
                        summary.AppendLine();
                    }
                }
                summary.AppendLine();
            }
            textResult.Text = summary.ToString();
        }
    }
}
```

启动该应用程序,给出其中筛选器有效的所有对象对应的一个列表,如图 52-11 所示。

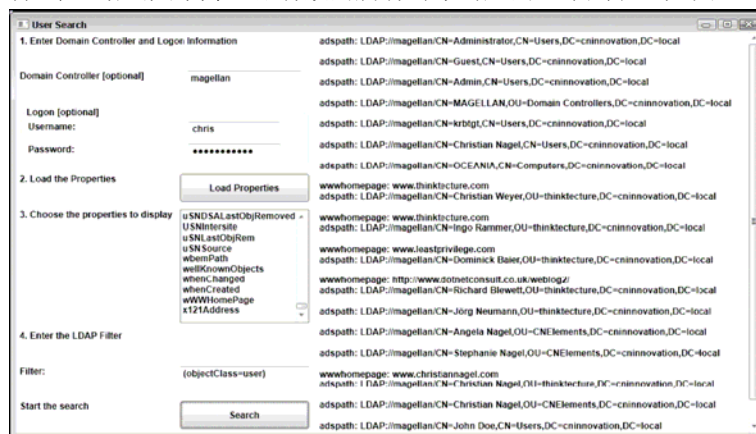


图 52-11

52.5 账户管理

在.NET 3.5 推出之前，很难创建和修改用户账户和组账户。以前使用的一种方式是利用 System.DirectoryServices 名称空间中的类或强类型化的本地 COM 接口。从.NET 3.5 开始新增了 System.DirectoryServices.AccountManagement 程序集，它提供 System.DirectoryServices 类的抽象，方法是使用特定的方法和属性搜索、修改、创建、更新用户和组。

这些类及其功能如表 52-4 所示。

表 52-4

类	说 明
PrincipalContext	使用 PrincipalContext 类可以配置账户管理的上下文。这里可以确定是否使用 Active Directory 域、本地系统中的账户或应用程序目录。为此，应把 ContextType 枚举设置为 Domain、Machine 或 ApplicationDirectory。根据上下文类型，还可以定义域名，指定用于访问的用户名和密码
Principal	Principal 类是所有主体的基类。使用静态方法 FindByIdentity()，可以获得一个 Principal 标识对象。通过主体对象可以访问各种属性，如架构中的名称、描述、可分辨的名称，以及对象类型。如果除了这个类中可用的方法和属性之外，还需要更多地控制主体，就可以使用 GetUnderlyingType()方法，它返回基础 DirectoryEntry 对象
AuthenticablePrincipal	AuthenticablePrincipal 类派生自 Principal 类，是所有可以验证身份的主体的基类。有几个查找主体的静态方法，如按登录时间或锁定时间、按不正确的密码尝试次数或按密码设置时间来查找。使用实例方法可以修改密码，释放账户
UserPrincipal ComputerPrincipal	UserPrincipal 类和 ComputerPrincipal 类派生自 AuthenticablePrincipal 基类，因此拥有这个基类的所有属性和方法。UserPrincipal 对象映射到用户账户，ComputerPrincipal 对象映射到计算机账户。UserPrincipal 类有许多属性可以获取和设置用户信息，如 EmployeeId、EmailAddress、GivenName 和 VoiceTelephoneNumber
GroupPrincipal	组不能进行身份验证，因此 GroupPrincipal 类直接派生自 Principal 类。在 GroupPrincipal 类中，可以使用 Members 属性和 GetMembers()方法获得组的成员
PrincipalCollection	PrincipalCollection 类包含一组 Principal 对象；例如，GroupPrincipal 类的 Members 属性返回一个 PrincipalCollection 对象
PrincipalSearcher	PrincipalSearcher 类是 DirectorySearcher 类的一个抽象，专门用于账户管理。使用 PrincipalSearcher 类不需要了解 LDAP 查询语法，因为这是自动创建的
PrincipalSearchResult<T>	PrincipalSearcher 类和 Principal 类的搜索方法返回一个 PrincipalSearchResult<T>

下面几节介绍可以使用 System.DirectoryServices.AccountManagement 名称空间中的类的场合。

52.5.1 显示用户信息

UserPrincipal 类的 Current 静态属性返回一个 UserPrincipal 对象和当前登录用户的信息：



```
using (var user = UserPrincipal.Current)
{
    Console.WriteLine("Context Server: {0}", user.Context.ConnectedServer);
    Console.WriteLine(user.Description);
    Console.WriteLine(user.DisplayName);
    Console.WriteLine(user.EmailAddress);
    Console.WriteLine(user.GivenName);
    Console.WriteLine("{0:d}", user.LastLogon);
}
```

代码段 AccountManagementSamples/Program.cs

运行这个应用程序，会显示用户的信息：

```
Context Server: Magellan.cninnovation.local
Developer, Author, Trainer, Consultant
Christian Nagel
Christian@ChristianNagel.com
Christian
2009/09/25
```

52.5.2 创建用户

使用 `UserPrincipal` 类可以创建新用户。首先需要有一个 `PrincipalContext` 类定义创建用户的位置。在 `PrincipalContext` 类中，根据应使用目录服务、计算机的本地账户还是应用程序目录，把 `ContextType` 设置为 `Domain`、`Machine` 或 `ApplicationDirectory` 的一个枚举值。如果当前用户不能在 Active Directory 中添加账户，那么还可以使用 `PrincipalContext` 类设置用户和密码，用于访问服务器。

接着，传递该 `PrincipalContext`，设置需要的属性，创建 `UserPrincipal` 类的一个实例。这里设置 `GivenName` 和 `EmailAddress` 属性。最后，必须调用 `UserPrincipal` 类的 `Save()` 方法，把新用户写入存储器中：



```
using (var context = new PrincipalContext(ContextType.Domain, "cninnovation"))
using (var user = new UserPrincipal(context, "Tom", "P@ssw0rd", true)
{
    GivenName = "Tom",
    EmailAddress = "test@test.com"
}))
{
    user.Save();
}
```

代码段 AccountManagementSamples/Program.cs

52.5.3 重置密码

要给已有用户重置密码，可以使用 `UserPrincipal` 对象的 `SetPassword()` 方法：

```
using (var context = new PrincipalContext(ContextType.Domain, "cninnovation"))
using (var user = UserPrincipal.FindByIdentity(context, IdentityType.Name,
    "Tom"))
{
    user.SetPassword("Pa$$w0rd");
    user.Save();
}
```

代码段 AccountManagementSamples/Program.cs

运行上述代码的用户需要有重置密码的权限。要把旧密码改为新密码，可以使用 `ChangePassword()` 方法。

52.5.4 创建组

新组的创建方式与新用户的创建相同。这里仅使用 `GroupPrincipal` 类替代 `UserPrincipal` 类。与创建新用户一样，也要设置属性，并调用 `Save()` 方法：



```
using (var ctx = new PrincipalContext(ContextType.Domain, "cninnovation"))
using (var group = new GroupPrincipal(ctx)
{
    Description = "Sample group",
    DisplayName = "Wrox Authors",
    Name = "WroxAuthors"
})
{
    group.Save();
}
```

代码段 AccountManagementSamples/Program.cs

52.5.5 在组中添加用户

要在组中添加用户，可以使用 `GroupPrincipal`，并把 `UserPrincipal` 添加到组的 `Members` 属性中。要获得已有用户和组，可以使用静态方法 `FindByIdentity()`：



```
using (var context = new PrincipalContext(ContextType.Domain))
using (var group = GroupPrincipal.FindByIdentity(
    context, IdentityType.Name, "WroxAuthors"))
using (var user = UserPrincipal.FindByIdentity(
    context, IdentityType.Name, "Stephanie Nagel"))
{
    group.Members.Add(user);
    group.Save();
}
```

代码段 AccountManagementSamples/Program.cs

52.5.6 查找用户

`UserPrincipal` 对象的静态方法可以根据某些预定义的条件查找用户。这里的示例说明了如何使用 `FindPasswordSetTime()` 方法查找在最近 30 天内未修变密码的用户。这个方法返回一个 `PrincipalSearchResult<UserPrincipal>` 集合，迭代它就可以显示用户名、最后一次登录时间和重置密码的时间：



```
using (var context = new PrincipalContext(ContextType.Domain, "explorer"))
using (var users = UserPrincipal.FindByPasswordSetTime(context,
    DateTime.Today - TimeSpan.FromDays(30), MatchType.LessThan))
{
    foreach (var user in users)
    {
        Console.WriteLine("{0}, last login: {1}, " +
            "last password change: {2}", user.Name, user.LastLogon,

```

```

        user.LastPasswordSet);
    }
}

```

代码段 AccountManagementSamples/Program.cs

UserPrincipal 类中用于查找用户的其他方法有 FindByBadPasswordAttempt()、FindByExpirationTime()、FindByLockoutTime() 和 FindByLogonTime()。

使用 PrincipalSearcher 类可以更灵活地查找用户。这个类是 DirectorySearcher 类的一个抽象，且在后台使用 DirectorySearcher 类。在 PrincipalSearcher 类中，可以把任意 Principal 对象赋予 QueryFilter 属性。

在下面的例子中，把带有 Surname 和 Enabled 属性的 UserPrincipal 对象赋予 QueryFilter。这样，就用 PrincipalSearchResult 集合返回所有姓氏为 Nag 且启用的用户对象。PrincipalSearcher 类创建一个 LDAP 查询字符串，进行搜索。



```

var context = new PrincipalContext(ContextType.Domain);
var userFilter = new UserPrincipal(context);
userFilter.Surname = "Nag*";
userFilter.Enabled = true;

using (var searcher = new PrincipalSearcher())
{
    searcher.QueryFilter = userFilter;
    var searchResult = searcher.FindAll();
    foreach (var user in searchResult)
    {
        Console.WriteLine(user.Name);
    }
}

```

代码段 AccountManagementSamples/Program.cs

52.6 DSML

在 System.DirectoryServices.Protocols 名称空间中，可以通过 DSML(目录服务标记语言)访问 Active Directory。DSML 是 OASIS 组(<http://www.oasis-open.org>)定义的一个标准，它允许通过 Web 服务访问目录服务。要通过 DSML 访问 Active Directory，至少要安装 Windows Server 2003 R2。

图 52-12 显示了 DSML 的一个配置。提供 DSML 服务的系统通过 LDAP 访问 Active Directory。在客户端系统上，使用 System.DirectoryServices.Protocols 名称空间中的 DSML 类向 DSML 服务发出 SOAP 请求。

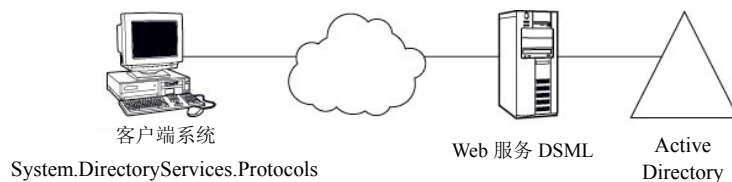


图 52-12

52.6.1 System.DirectoryServices.Protocols 名称空间中的类

表 52-5 列出了 System.DirectoryServices.Protocols 名称空间中的主要类。

表 52-5

类	说 明
DirectoryConnection	DirectoryConnection 类是可以定义到目录服务的连接的所有连接类的基类。派生自 DirectoryConnection 类的类有 LdapConnection(为使用 LDAP 协议)、DsmSoapConnection 和 DsmSoapHttpConnection。使用 SendRequest()方法可以把消息发送给目录服务
DirectoryRequest	派生自 DirectoryRequest 基类的类可以把请求发送给目录服务。根据请求的类型, 使用 SearchRequest、AddRequest、DeleteRequest 和 ModifyRequest 等类发送请求
DirectoryResponse	用 SendRequest 返回的结果的类型派生自 DirectoryResponse 基类。这种派生类的示例有 SearchResponse、AddResponse、DeleteResponse 和 ModifyResponse

52.6.2 用 DSML 搜索 Active Directory 对象

本节用一个例子说明如何搜索目录服务对象。在下面的代码中, 首先实例化一个 DsmSoapHttpConnection 对象, 该对象定义与 DSML 服务的连接。该连接用包含 Uri 对象的 DsmDirectoryIdentifier 类定义, 还可以使用该连接设置用户证书:



可从
wrox.com
下载源代码

```
Uri uri = new Uri("http://dsmlserver/dsml");
var identifier = new DsmDirectoryIdentifier(uri);

var credentials = new NetworkCredential();
credentials.UserName = "cnagel";
credentials.Password = "password";
credentials.Domain = "explorer";

var dsmlConnection = new DsmSoapHttpConnection(identifier, credentials);
```

代码段 DsmSample/Program.cs

定义连接后, 就可以配置搜索请求。搜索请求包含开始搜索的目录项、LDAP 搜索筛选器和应从搜索中返回的属性值的定义。这里把筛选器设置为(objectClass=user), 以便从搜索中返回所有用户对象。把 attributesToReturn 属性设置为 null, 并且可以读取所有带值的属性。SearchScope 是 System.DirectoryServices.Protocols 名称空间中的一个枚举, 它类似于 System.DirectoryServices 名称空间中的 SearchScope 枚举, 它定义搜索的深度。这里把 SearchScope 属性设置为 Subtree, 搜索整个 Active Directory 树。

搜索筛选器可以用 LDAP 字符串定义, 或使用包含在 XmlDocument 类中的一个 XML 文档定义:

```
string distinguishedName = null;
string ldapFilter = "(objectClass=user)";
string[] attributesToReturn = null;// return all attributes

var searchRequest = new SearchRequest(distinguishedName,
    ldapFilter, SearchScope.Subtree, attributesToReturn);
```

用 `SearchRequest` 对象定义该搜索后, 就调用 `SendRequest()` 方法, 把该搜索发送给 Web 服务。`SendRequest()` 是 `DsmlSoapHttpConnection` 类的一个方法, 它返回一个 `SearchResponse` 对象, 从该对象中可以读取返回的对象。

除了调用同步的 `SendRequest()` 方法, `DsmlSoapHttpConnection` 类还提供了异步的 `BeginSendRequest()` 和 `EndSendRequest()` 方法, 这些异步方法遵循异步 .NET 架构。



异步架构可参见第 20 章。

```
SearchResponse searchResponse =
    (SearchResponse)dsmlConnection.SendRequest(searchRequest);
```

返回的 `Active Directory` 对象可以在 `SearchResponse` 中读取。`SearchResponse.Entries` 包含用 `SearchResultEntry` 类型包装的所有项的一个集合。`SearchResultEntry` 类的 `Attributes` 属性包含所有特性。每个属性都可以使用 `DirectoryAttribute` 类读取。

在代码示例中, 把每个对象的可分辨的名称都写入控制台中。接着, 访问组织单元的属性值, 把组织单元的名称写入控制台中。之后把 `DirectoryAttribute` 对象的所有值写入控制台中。

```
Console.WriteLine("\r\nSearch matched {0} entries:",
    searchResponse.Entries.Count);
foreach (SearchResultEntry entry in searchResponse.Entries)
{
    Console.WriteLine(entry.DistinguishedName);

    // retrieve a specific attribute
    DirectoryAttribute attribute = entry.Attributes["ou"];
    Console.WriteLine("{0} = {1}", attribute.Name, attribute[0]);
    // retrieve all attributes
    foreach (DirectoryAttribute attr in entry.Attributes.Values)
    {
        Console.WriteLine("{0}=", attr.Name);

        // retrieve all values for the attribute
        // the type of the value can be one of string, byte[] or Uri
        foreach (object value in attr)
        {
            Console.WriteLine("{0} ", value);
        }
        Console.WriteLine();
    }
}
```

添加、修改和删除对象的操作与搜索对象类似。根据要执行的操作, 使用相应的方法。

52.7 小结

本章介绍了 `Active Directory` 的体系结构, 域、树和森林的重要概念。利用它, 可以访问整个企业的信息。在编写访问 `Active Directory` 服务的应用程序时, 必须注意读取的数据可能不是最新的,

因为有复制延迟。

使用 `System.DirectoryServices` 名称空间中的类，可以很容易地访问包装到 ADSI 提供程序中的 Active Directory 服务。`DirectoryEntry` 类可以直接读写数据存储器中的对象。

使用 `DirectorySearcher` 类可以进行复杂的搜索，定义筛选器、超时、加载的属性和范围等。使用全局目录，可以加快对整个企业中的对象的搜索，因为它在森林中存储了所有对象的只读版本。

DSML 是另一个允许通过 Web 服务接口访问 Active Directory 的 API。

`System.DirectoryServices.AccountManagement` 名称空间中的类提供了一个抽象，很容易创建和修改用户、组和计算机账户。

第 53 章

C#、Visual Basic、C++/CLI 和 F#

本章内容:

- 名称空间
- 定义类型
- 方法
- 数组
- 控制语句
- 循环
- 异常处理
- 继承
- 资源管理
- 委托
- 事件
- 泛型
- LINQ 查询
- C++/CLI 混合本地代码和托管代码

C#是专为.NET 设计的编程语言。编写.NET 应用程序可以使用 50 多种语言,例如, Eiffel、Smalltalk、COBOL、Haskell、Pizza、Pascal、Delphi、Oberon、Prolog, 以及 Ruby 等。Microsoft 还发布了 C#、Visual Basic、C++/CLI、J#、JScript.NET 和 F#。

每种语言都有优缺点;一些任务很容易用一种语言完成,但用另一种语言完成就很复杂。.NET Framework 中的类总相同,但语言的语法从.NET Framework 中抽象出了各种功能。例如, C#的 using 语句很便于使用实现 IDisposable 接口的对象。其他语言要实现该功能就需要较多代码。

Microsoft 最常用的.NET 语言是 C#和 Visual Basic。C#是专为.NET 设计的新语言,其理念来自于 C++、Java、Pascal 和其他语言。Visual Basic 植根于 Visual Basic 6,用.NET 的面向对象功能进行了扩展。

C++/CLI 是 C++的一种扩展,基于 ECMA 标准(ECMA 372)。C++/CLI 的一大优点是可将本地代码与托管代码混合起来。我们可以扩展已有的本地 C++应用程序,添加.NET 功能,将.NET 类添加到本地库中,使它们能用于其他.NET 语言(如 C#)。还可以用 C++/CLI 编写完全托管的应用程序。

F#是 Visual Studio 中的一种新语言,它提供对函数编程的特殊支持。它可以与.NET 很好地合作,因为它也支持传统的面向对象编程。在 F#中,函数编程是指,函数可以用作值,这样,就很容易从

多个函数中构建函数，进行函数的通道化，其中函数逐个地链接起来。

本章介绍如何将.NET 应用程序从一种语言转换为另一种语言。Visual Basic 或 C++/CLI 示例代码可以映射到 C#，反之亦然。要进行这个比较，不需要学习 F#或 C++/CLI，因为本章主要介绍如何把 C#映射到其他语言上，而不讨论其他语言的核心概念和理念。



要理解本章的内容，读者应知道 C#，并已阅读了本书的前几章。但不必了解 Visual Basic、C++/CLI 和 F#。

53.1 名称空间

把.NET 类型组织到名称空间中。这 4 种语言定义和使用名称空间的语法完全不同。

为了导入名称空间，C#使用 `using` 关键字。C++/CLI 完全基于 C++语法和 `using namespace` 语句，Visual Basic 定义 `Imports` 关键字，F#定义 `open` 关键字，来导入名称空间。

在 C#和 Visual Basic 中，可以给类或其他名称空间定义别名。在 C++/CLI 中，别名只能引用其他名称空间，不能引用类。C++需要用 `namespace` 关键字定义别名，这个关键字也用于定义名称空间。Visual Basic 同样使用 `Imports` 关键字。

要定义名称空间，这 4 种语言都使用 `namespace` 关键字，但仍有一个区别。在 C++/CLI 中，不能用一条名称空间语句定义层次结构的名称空间，而必须嵌套名称空间。项目设置有一个重要的区别：在 C#的项目设置中定义名称空间，就是定义默认的名称空间，该名称空间会显示在添加到项目中的所有新项的代码中。在 Visual Basic 的项目设置中，定义项目中所有项使用的根名称空间。在源代码中声明的名称空间只定义根名称空间中的子名称空间。

```
// C#
using System;
using System.Collections.Generic;
using Assm = Wrox.ProCSharp.Assemblies;
namespace Wrox.ProCSharp.Languages
{
}

// C++/CLI
using namespace System;
using namespace System::Collections::Generic;
namespace Assm = Wrox::ProCSharp::Assemblies;
namespace Wrox
{
    namespace ProCSharp
    {
        namespace Languages
        {
        }
    }
}
' Visual Basic
Imports System
```

```
Imports System.Collections.Generic
Imports Assm = Wrox.ProCSharp.Assemblies
Namespace Wrox.ProCSharp.Languages

End Namespace

// F#
namespace Wrox.ProCSharp.Languages
open System
open System.Collections.Generic
```

53.2 定义类型

.NET 区分引用类型和值类型。在 C# 中，引用类型用类定义，值类型用结构定义。除了引用类型和值类型之外，本节还介绍如何定义接口(引用类型)和枚举(值类型)。

53.2.1 引用类型

要声明引用类型，C# 和 Visual Basic 使用 `class` 关键字。在 C++/CLI 中，类和结构基本相同，不能像 C# 和 Visual Basic 那样区分引用类型和值类型。C++/CLI 有一个 `ref` 关键字，它定义托管类。定义 `ref class` 或 `ref struct` 可以创建引用类型。

在 C# 和 C++/CLI 中，类用花括号括起来。C++/CLI 要求在类声明的最后加上分号。Visual Basic 在类的最后使用 `End Class` 语句。F# 创建类和其他类型的方式是：使用 `type` 关键字和后面的类型名以及可选的 `as` 标识符，`as` 标识符定义实例标识符的名称：

```
// C#
public class MyClass
{
}

// C++/CLI
public ref class MyClass
{
};
public ref struct MyClass2
{
};

' Visual Basic
Public Class MyClass2
End Class

// F# signature file
type MyClass() as this =
// ... members of the class
```

在使用引用类型时，需要声明一个变量，必须在托管堆上给该对象分配内存。在声明引用类型的句柄时，C++/CLI 会定义句柄操作符[^]，它有点类似于 C++ 指针*。`gcnew` 操作符分配托管堆上的内存。使用 C++/CLI 还可以在本地声明变量，但对于引用类型，仍在托管堆上给该对象分配内存。在 Visual Basic 中，变量声明以 `Dim` 语句开头，其后是变量名。对于 `new` 和对象类型，需要在托管

堆上分配内存。

```
// C#
MyClass obj = new MyClass();
var obj2 = new MyClass()

// C++/CLI
MyClass^ obj = gcnew MyClass();
MyClass obj2;

' Visual Basic
Dim obj as New MyClass2()
```

如果引用类型没有引用内存，这 4 种语言就使用其他关键字：C#定义 `null` 字面量，C++/CLI 定义 `nullptr`(`NULL` 仅对本地对象有效)，Visual Basic 定义 `Nothing`。F#一般不使用 `null` 值——用 F#定义的类型不允许使用这个值。使用不是 F#语言定义的类型时，可能得到 `null` 值。

表 53-1 列出了预定义的引用类型。C++/CLI 没有像其他语言那样定义对象和字符串类型。当然，可以使用 .NET Framework 定义的类。

表 53-1

.NET 类型	C#	C++/CLI	Visual Basic	F#
System.Object	object	未定义	Object	obj
System.String	string	未定义	String	string

53.2.2 值类型

要声明值类型，C#使用 `struct` 关键字；C++/CLI 使用 `value` 关键字，Visual Basic 使用 `Structure` 关键字，F#使用 `type` 关键字和 `[<Struct>]` 特性。另一种选择是使用 `struct` 关键字和 `end` 关键字：

```
// C#
public struct MyStruct
{
}

// C++/CLI
public value class MyStruct
{
};

' Visual Basic
Public Structure MyStruct
End Structure

// F#
[<Struct>]
type MyStruct =
    val x : int
type MyStruct2 =
    struct
        val x : int
    end
```

在 C++/CLI 中，可以把值类型分配到栈上、内置堆上(使用 `new` 操作符)或托管堆上(使用 `gcnew` 操作符)。C#和 Visual Basic 没有这些选项，但用 C++/CLI 混合本地代码和托管代码时，这些选项就变得非常重要。

```
// C#
MyStruct ms;

// C++/CLI
MyStruct ms1;
MyStruct* pms2 = new MyStruct();
MyStruct^ hms3 = gcnew MyStruct();

' Visual Basic
Dim ms as MyStruct
```

表 53-2 列出了这 4 种语言中预定义的值类型。在 C++/CLI 中，`char` 类型的大小为 1 字节，用于存储 ASCII 字符。在 C#中，`char` 类型的大小为 2 字节，用于存储 Unicode 字符，而 C++/CLI 为此使用 `wchar_t` 类型。C++的 ANSI 标准只定义了 `short <= int <= long`。在 32 位计算机上，`int` 和 `long` 的大小都是 32 位。要在 C++中定义 64 位的变量，需要使用 `long long`。

表 53-2

.NET 类型	C#	C++/CLI	Visual Basic	F#	大小
Char	char	wchar_t	Char	char	2 字节
Boolean	bool	bool	Boolean	bool	1 字节, 包含 true 或 false
Int16	short	short	Short	int16	2 字节
UInt16	ushort	unsigned short	Ushort	uint16	2 字节, 无符号
Int32	int	int	Integer	int	4 字节
UInt32	uint	unsigned Int	UInteger	uint	4 字节, 无符号
Int64	long	long long	Long	int64	8 字节
UInt64	ulong	unsigned long long	ULong	uint64	8 字节, 无符号

53.2.3 类型推断

C#允许在定义本地变量时，不显式声明数据类型，而使用 `var` 关键字。类型从指定的初始值中推断出来。在 Visual Basic 中，只要打开 `Option infer`，就可以使用 `Dim` 关键字推断变量的类型。这个功能需要使用编译器选项 `infer+`，或者在 Visual Basic 中使用项目配置页面。在 F#中，推断值、变量、参数和返回值的类型：

```
// C#
var x = 3;

' Visual Basic
Dim x = 3

// F#
let x = 3
```

53.2.4 接口

这 3 种语言在定义接口方面非常类似，它们都使用 `interface` 关键字。但 F# 不同，因为其接口用仅包含抽象成员的类型定义：



可从
wrox.com
下载源代码

```
// C#
public interface IDisplay
{
    void Display();
}
```

代码段 CSharp/Person.cs



可从
wrox.com
下载源代码

```
// C++/CLI
public interface class IDisplay
{
    void Display();
};
```

代码段 CPPCLI/Person.h



可从
wrox.com
下载源代码

```
' Visual Basic
Public Interface IDisplay
    Sub Display
End Interface
```

代码段 VisualBasic/Person.vb



可从
wrox.com
下载源代码

```
// F#
[<Interface>]
type public IDisplay
    abstract member Display : unit -> unit
```

代码段 FSharp/Person.fs

实现接口的方式则不同。C#和 C++/CLI 在类名后面加上冒号，之后是接口名，并实现用该接口定义的方法。在 C++/CLI 中，方法必须声明为 `virtual`。Visual Basic 使用 `Implements` 关键字实现接口，接口定义的方法也需要附加 `Implements` 关键字。F#在类型定义中列出已实现的接口。接口用 `interface` 和 `with` 关键字实现，并实现下述接口的所有成员：



可从
wrox.com
下载源代码

```
// C#
public class Person: IDisplay
{
    public void Display()
    {
    }
}
```

代码段 CSharp/Person.cs



可从
wrox.com
下载源代码

```
// C# explicit interface implementation
public class Person: IDisplay
{
    void IDisplay.Display()
    {
    }
}
```

```
// C++/CLI
public ref class Person: IDisplay
{
public:
    virtual void Display() { }
};
```

代码段 CPPCLI/Person.h



```
' Visual Basic
Public Class Person
    Implements IDisplay
    Public Sub Display Implements IDisplay.Display
    End Sub
End Class
```

代码段 VisualBasic/Person.vb



```
// F#
type Person as this =
    interface IDisplay with
        member this.Display() = printfn "%s %s" this.FirstName this.LastName
```

代码段 FSharp/Person.fs

53.2.5 枚举

这 3 种语言在定义枚举方面非常类似，它们都使用 `enum` 关键字(只有 Visual Basic 使用新的一行代码，而不是用逗号分隔元素)。F#使用 `type` 关键字和匹配表达式“|”：



```
// C#
public enum Color
{
    Red, Green, Blue
}
```

代码段 CSharp/Color.cs



```
// C++/CLI
public enum class Color
{
    Red, Green, Blue
};
```

代码段 CPPCLI/Color.h



```
' Visual Basic
Public Enum Color
    Red
    Green
    Blue
End Enum
```

代码段 VisualBasic/Color.vb



```
// F#
type Color =
| Red = 0
| Green = 1
```

| Blue = 2

代码段 FSharp/Color.fs

53.3 方法

除了 F# 之外，总是在类中声明方法。C++/CLI 的语法非常类似于 C#，除了访问修饰符不包含在方法声明中，而是写在方法声明之前。访问修饰符必须用冒号结束。在 Visual Basic 中，使用 Sub 关键字定义方法。在 F# 中，可以用 let 关键字定义方法。Foo 有一个参数 x，返回 x 加 3 的结果。参数类型和返回类型可以用函数声明，如下所示。对于 add 函数，x 和 y 的类型是 int，返回一个 int。如果没有声明类型，类型就由编译器推断。对于 Foo 函数，x 也是 int，因为 3 要与 x 相加，而 3 是 int。在类中定义方法使用 member 关键字。实例方法有一个用类定义中的 as 标识符定义的前缀。虽然 F# 不需要使用 this、Me 或 self 访问当前实例，但可以定义任意标识符。

```
// C#
public class MyClass
{
    public void Foo()
    {
    }
}

// C++/CLI
public ref class MyClass
{
public:
    void Foo()
    {
    }
};

' Visual Basic
Public Class MyClass1
    Public Sub Foo
    End Sub
End Class

// F# function
let foo x = x + 3
let add (x : int, y : int) : int = x + y

// method within a class
type MyClass as this =
    member this.Foo() =
        printfn "MyClass.Foo"
```

53.3.1 方法的参数和返回类型

在 C# 和 C++/CLI 中，传递给方法的参数在圆括号中定义。参数的类型在变量名的前面声明。如果从方法中返回一个值，该方法就用返回值的类型定义，而不是 void。

Visual Basic 使用 Sub 语句声明没有返回值的方法，用 Function 语句声明有返回类型的方法。返

回类型放在方法名和圆括号的后面。参数中的变量声明和类型在 Visual Basic 中的顺序也不同：类型在变量的后面，而在 C#和 C++/CLI 中，变量在类型的后面。

在 F#中，如果方法没有返回值，它就声明为 `unit`，`unit` 类似于 C#中的 `void`。`Foo()`方法声明为接收一个 `int` 参数，并返回一个 `int`：

```
// C#
public class MyClass
{
    public int Foo(int i)
    {
        return 2 * i;
    }
}

// C++/CLI
public ref class MyClass
{
public:
    int Foo(int i)
    {
        return 2 * i;
    }
};

' Visual Basic
Public Class MyClass2
    Public Sub Foo1(ByVal i as Integer)
    End Sub
    Public Function Foo(ByVal i As Integer) As Integer
        Return 2 * i
    End Sub
End Class

// F#
type MyClass as this =
    member this.Foo(i : int) : int = i * 2
```

53.3.2 参数修饰符

在默认情况下，值类型按值传递，引用类型按引用传递。如果作为参数传递的值类型要在主调方法中更改，那么 C#允许使用参数修饰符 `ref`。

C++/CLI 定义一个托管引用操作符“%”。这个操作符类似于 C++引用操作符“&”，但“%”可以用于托管类型，垃圾收集器可以跟踪这些对象，以防它们在托管堆中移动。

Visual Basic 使用 `ByRef` 关键字按引用传递参数。

```
// C#
public class ParameterPassing
{
    public void ChangeVal(ref int i)
    {
        i = 3;
    }
}

// C++/CLI
```

```

public ref class ParameterPassing
{
public:
    int ChangeVal(int% i)
    {
        i = 3;
    }
};

' Visual Basic
Public Class ParameterPassing
    Public Sub ChangeVal(ByRef i as Integer)
        i = 3
    End Sub
End Class

```

调用带引用参数的方法时，只有 C# 语言需要使用参数修饰符。C++/CLI 和 Visual Basic 在调用带或不带参数修饰符的方法方面没有区别。这方面 C# 有优势，因为可以立即看出，参数值可以在主调方法中更改。

由于调用语法没有区别，因此 Visual Basic 不允许在重载方法时仅更改修饰符。C++/CLI 编译器允许在重载方法时仅更改修饰符，但不能编译调用者，因为这个解析方法有多义性。虽然在 C# 中允许重载并使用只有参数修饰符的方法，但这不是一个好的编程习惯。

```

// C#
ParameterPassing obj = new ParameterPassing();
int a = 1;
obj.ChangeVal(ref a);
Console.WriteLine(a); // writes 3

// C++/CLI
ParameterPassing obj;
int a = 1;
obj.ChangeVal(a);
Console.WriteLine(a); // writes 3

' Visual Basic
Dim obj as new ParameterPassing()
Dim i as Integer = 1
obj.ChangeVal(i)
Console.WriteLine(i) // writes 3

```



当从方法中返回一个参数时，C# 还定义了 `out` 关键字。这个选项在 C++/CLI 和 Visual Basic 都不可用。只要调用者和被调用者在相同的应用程序域中，`out` 和 `ref` 在后台就没有区别。用 C# 的 `out` 参数修饰符声明的方法也可以在 C++/CLI 和 Visual Basic 中以与 `ref` 参数修饰符相同的方式调用。如果方法要跨多个应用程序域或进程使用，C++/CLI 和 Visual Basic 就可以使用 `[out]` 特性。

53.3.3 构造函数

在 C# 和 C++/CLI 中，构造函数与类同名。Visual Basic 使用 `New` 过程。`this` 和 `Me` 关键字用于

访问这个实例的成员。在一个构造函数中调用另一个构造函数时，C#需要初始化一个成员。C++/CLI 和 Visual Basic 可以将构造函数作为方法调用。

F#有点区别，它使用类型声明定义构造函数。Person 类的构造函数接受两个 string 参数，其他构造函数使用 new 关键字定义。new()方法定义没有参数的构造函数。对于 Person 类，用两个参数调用构造函数。



```
// C#
public class Person
{
    public Person()
        : this("unknown", "unknown") { }
    public Person(string firstName, string lastName)
    {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    private string firstName;
    private string lastName;
}
```

代码段 CSharp/Person.cs



```
// C++/CLI
public ref class Person
{
public:
    Person()
    {
        Person("unknown", "unknown");
    }
    Person(String^ firstName, String^ lastName)
    {
        this->firstName = firstName;
        this->lastName = lastName;
    }
private:
    String^ firstName;
    String^ lastName;
};
```

代码段 CPPCLI/Person.h



```
' Visual Basic
Public Class Person
    Public Sub New()
        Me.New("unknown", "unknown")
    End Sub
    Public Sub New(ByVal firstName As String, ByVal lastName As String)
        Me.MyFirstName = firstName
        Me.MyLastName = lastName
    End Sub
    Private MyFirstName As String
    Private MyLastName As String
End Class
```

代码段 VisualBasic/Person.vb



可从
wrox.com
下载源代码

```
// F#
type Person(firstName0 : string, lastName0 : string) as this =
    let mutable firstName, lastName = firstName0, lastName0
    new () = Person("unknown", "unknown")
```

代码段 FSharp/Person.fs

53.3.4 属性

要定义属性，在属性块中 C# 需要 `get` 和 `set` 存取器。C#、C++/CLI 和 Visual Basic 中还有一种速记表示法：如果 `get` 和 `set` 存取器只返回或设置一个简单的变量，就不需要自定义实现代码。这是一个自动属性。其语法与 C++/CLI 和 Visual Basic 不同，这两种语言都有 `property` 关键字，且需要用 `set` 存取器定义变量的值。C++/CLI 还需要用 `get` 存取器指定返回类型，用 `set` 存取器指定参数类型。

C++/CLI 在用简短版本编写属性时，只需定义属性的类型和名称，`get` 和 `set` 存取器由编译器自动创建。如果除了设置和返回变量之外不需要做其他工作，就可以使用这个简短版本。如果存取器的实现代码还需要做其他工作(例如，检查变量或刷新)就必须使用属性的完整语法。

F# 把属性定义为类的成员，使用 `get()` 作为 `get` 存取器，把 `set (value)` 作为 `set` 存取器。



可从
wrox.com
下载源代码

```
// C#
public class Person
{
    private string firstName;
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    public string LastName { get; set; }
}
```

代码段 CSharp/Person.cs



可从
wrox.com
下载源代码

```
// C++/CLI
public ref class Person
{
private:
    String^ firstName;
public:
    property String^ FirstName
    {
        String^ get()
        {
            return firstName;
        }
        void set(String^ value)
        {
            firstName = value;
        }
    }
    property String^ LastName;
};
```

代码段 CPPCLI/Person.h



可从
wrox.com
下载源代码

```
' Visual Basic
Public Class Person
    Private myFirstName As String
    Public Property FirstName()
        Get
            Return myFirstName
        End Get
        Set(ByVal value)
            myFirstName = value
        End Set
    End Property
    Public Property LastName As String
End Class
```

代码段 VisualBasic/Person.vb



可从
wrox.com
下载源代码

```
// F#
type Person(firstName0 : string, lastName0 : string) as this =
    let mutable firstName, lastName = firstName0, lastName0
    member this.FirstName
        with get() = firstName
        and set(value) = firstName <- value
    member this.LastName
        with get() = lastName
        and set(value) = lastName <- value
```

代码段 FSharp/Person.fs

在 C# 和 C++/CLI 中，只读属性只有 `get` 存取器。在 Visual Basic 中，还必须指定 `ReadOnly` 修饰符，只写属性必须用 `WriteOnly` 修饰符和 `set` 存取器定义。

```
' Visual Basic
Public ReadOnly Property Name()
    Get
        Return myFirstName & " " & myLastName
    End Get
End Property
```

53.3.5 对象初始值设定项

在 C# 和 Visual Basic 中，属性可以使用对象初始值设定项进行初始化。属性可以使用类似于数组初始值设定项的花括号初始化。C# 和 Visual Basic 中的语法非常类似，Visual Basic 仅使用 `With` 关键字：

```
// C#
Person p = new Person { FirstName = "Tom", LastName = "Turbo" };

' Visual Basic
Dim p As New Person With { .FirstName = "Tom", .LastName = "Turbo" }
```

53.3.6 扩展方法

扩展方法是 LINQ 的基础。在 C# 和 Visual Basic 中，可以创建扩展方法。但是其语法不同。C# 在第一个参数中用 `this` 关键字标记扩展方法，Visual Basic 用 `<Extension>` 特性标记扩展方法。在 F# 中，扩展语法的术语是类型扩展。类型扩展通过 `type` 声明指定类型的完全限定名称，并用 `with` 关键字扩展它：

```
// C#
public static class StringExtension
{
    public static void Foo(this string s)
    {
        Console.WriteLine("Foo {0}", s);
    }
}

' Visual Basic
Public Module StringExtension
    < Extension() > _
    Public Sub Foo(ByVal s As String)
        Console.WriteLine("Foo {0}", s)
    End Sub
End Module

// F#
module Wrox.ProCSharp.Languages.StringExtension
type System.String with
    member this.Foo = printfn "String.Foo"
```

53.4 静态成员

静态字段只为某种类型的所有对象实例化一次。C#和 C++/CLI 都使用 `static` 关键字; Visual Basic 则使用 `Shared` 关键字。

使用静态成员的方法是: 指定类名, 其后是 “.” 操作符和静态成员名。C++/CLI 使用 “:” 操作符访问静态成员。



```
// C#
public class Singleton
{
    private static SomeData data = null;
    public static SomeData GetData()
    {
        if (data == null)
        {
            data = new SomeData();
        }
        return data;
    }
}

// use:
SomeData d = Singleton.GetData();
```

代码段 CSharp/Singleton.cs



```
// C++/CLI
public ref class Singleton
{
private:
    static SomeData^ hData;
```

```

public:
    static SomeData^ GetData()
    {
        if (hData == nullptr)
        {
            hData = gcnew SomeData();
        }
        return hData;
    }
};
// use:
SomeData^ d = Singleton::GetData();

```

代码段 CPPCLI/Singleton.h



```

' Visual Basic
Public Class Singleton
    Private Shared data As SomeData
    Public Shared Function GetData() As SomeData
        If data Is Nothing Then
            data = new SomeData()
        End If
        Return data
    End Function
End Class
' Use:
Dim d As SomeData = Singleton.GetData()

```

代码段 VisualBasic/Singleton.fs

53.5 数组

数组在第 6 章讨论过。Array 类总是后台的 .NET 数组。声明数组时，编译器会创建一个派生自 Array 基类的类。在设计 C# 时，采用了 C++ 数组的方括号语法，并用数组初始化值设定项扩展它。

```

// C#
int[] arr1 = new int[3] {1, 2, 3};
int[] arr2 = {1, 2, 3};

```

如果在 C++/CLI 中使用方括号，就会创建一个本地 C++ 数组，而不是基于 Array 类的数组。为了创建 .NET 数组，C++/CLI 引入了 array 关键字。这个关键字使用类似于泛型的语法，即使用尖括号。在尖括号中指定元素的类型。C++/CLI 支持数组初始化值设定项的语法与 C# 相同。

```

// C++/CLI
array<int>^ arr1 = gcnew array<int>(3) {1, 2, 3};
array<int>^ arr2 = {1, 2, 3};

```

Visual Basic 给数组使用括号。它要求在数组声明中指定最后一个元素编号，而不是数组中的元素个数。在每种 .NET 语言中，数组都以元素编号 0 开始，Visual Basic 也是如此。为了使之更清楚，Visual Basic 9 在数组声明中引入了 0 To number 表达式。它总是从 0 开始，To 使该语法可读性更强。

如果数组用 new 操作符初始化，那么 Visual Basic 也支持数组初始化值设定项。

```
' Visual Basic
Dim arr1(0 To 2) As Integer()
Dim arr2 As Integer() = New Integer(0 To 2) {1, 2, 3};
```

F#提供了初始化数组的不同方式。创建小型数组时，可以使用“[”和“]”作为左花括号和右花括号，并指定所有元素，这些元素用分号隔开。序列表达式(如 `for in` 语句)可以用于初始化数组中的元素，如下面的 `arr2`。`arr3` 在初始化时用 `zeroCreate` 类型扩展为 `Array` 类，以创建并初始化 20 个元素。可以使用索引器访问数组。对于数组片段，可以访问数组中的一个范围，例如，`arr2.[4..]` 从第 5 个元素开始访问，直到最后一个元素。

```
// F#
let arr1 = [| 1; 2; 3 |]
let arr2 = [| for i in 1..10 -> i |]
let arr3 : int array = Array.zeroCreate 20
```

53.6 控制语句

控制语句指定应运行什么代码。C#定义了 `if` 和 `switch` 语句，以及条件操作符。

53.6.1 if 语句

C#的 `if` 语句与 C++/CLI 版本相同，Visual Basic 使用 `If-Then/Else/End If` 来替代花括号。

```
// C# and C++/CLI
if (a == 3)
{
    // do this
}
else
{
    // do that
}

' Visual Basic
If a = 3 Then
    ' do this
Else
    ' do that
End If
```

53.6.2 条件操作符

C#和 C++/CLI 支持条件操作符，它是 `if` 语句的一个轻型版本。在 C++/CLI 中，这个操作符称为三元操作符。第一个参数必须有一个布尔值。如果结果为 `true`，就计算第一个表达式；否则，就计算第二个表达式。Visual Basic 在 Visual Basic Runtime Library 中提供了有类似功能的 `IIf` 函数。F# 把 `if/then/else` 用作条件操作符

```
// C#
string s = a > 3 ? "one" : "two";

// C++/CLI
```

```
String^ s = a > 3 ? "one": "two";

' Visual Basic
Dim s As String = IIf(a > 3, "one", "two")

// F#
if a = 3 then printfn "do this" this else printfn "do that"
```

53.6.3 switch 语句

C#和 C++/CLI 中的 switch 语句看起来很类似，但它们有重要的区别。C#支持在 case 选项中使用字符串，但 C++不支持。在 C++中，必须使用 if-else。C++/CLI 支持从一个 case 选项向下一个 case 选项隐式跳转。但在 C#中，如果没有 break 或 goto 语句，编译器就会发出警告。只有 case 中没有语句时，C#才支持从一个 case 选项向下一个 case 选项隐式跳转。

Visual Basic 用 Select/Case 语句替代 switch/case 语句。它不需要也不允许使用 break 语句。即使 Case 后面没有一条语句，也不能从一个 case 选项向下一个 case 选项隐式跳转。Case 可以用 And、Or 和 To 合并，如 3 To 5。

F#用 match 和 with 关键字提供匹配表达式。匹配表达式的每一行都以“|”开头，后跟一种模式。下面的示例对 Suit.Hear | Suit.Diamond 使用 OR 模式。最后一个模式是通配符模式：

```
// C#
string GetColor(Suit s)
{
    string color;
    switch (s)
    {
        case Suit.Heart:
        case Suit.Diamond:
            color = "Red";
            break;
        case Suit.Spade:
        case Suit.Club:
            color = "Black";
            break;
        default:
            color = "Unknown";
            break;
    }
    return color;
}

// C++/CLI
String^ GetColor(Suit s)
{
    String^ color;
    switch (s)
    {
        case Suit::Heart:
        case Suit::Diamond:
            color = "Red";
            break;
        case Suit::Spade:
        case Suit::Club:
```

```

        color = "Black";
        break;
    default:
        color = "Unknown";
        break;
    }
    return color;
}

' Visual Basic
Function GetColor(ByVal s As Suit) As String
    Dim color As String = Nothing
    Select Case s
        Case Suit.Heart And Suit.Diamond
            color = "Red"
        Case Suit.Spade And Suit.Club
            color = "Black"
        Case Else
            color = "Unknown"
    End Select

    Return color
End Function

// F#
type SuitTest =
    static member GetColor(s : Suit) : string =
        match s with
        | Suit.Heart | Suit.Diamond -> "Red"
        | Suit.Spade | Suit.Club -> "Black"
        | _ -> "Unknown"

```

53.7 循环

使用循环，代码会重复执行，直到满足一个条件为止。C#中的循环详见第 2 章，包括 `for`、`while`、`do...while` 和 `foreach`。C#和 C++/CLI 的循环语句非常类似，而 Visual Basic 和 F#定义了不同的语句。

53.7.1 for 语句

C#和 C++/CLI 的 `for` 语句很类似。在 Visual Basic 中，不能在 `For/To` 语句中初始化变量，而必须提前初始化变量。`for/To` 不需要使用 `Step` 语句，因为默认使用 `Step 1`。只有不打算将递增量设置为 1，`for/To` 才需要使用 `Step` 关键字。F#使用 `for/to/do` 和 `for/downto/do` 关键字。

```

// C#
for (int i = 0; i < 100; i++)
{
    Console.WriteLine(i);
}

// C++/CLI
for (int i = 0; i < 100; i++)
{
    Console::WriteLine(i);
}

```

```

' Visual Basic
Dim count as Integer
For count = 0 To 99 Step 1
    Console.WriteLine(count)
Next

// F#
for i = 1 to 10 do
    printfn i
for i = 10 downto 1 do
    printfn i

```

53.7.2 while 和 do...while 语句

while 和 do...while 语句在 C#和 C++/CLI 中相同。Visual Basic 中的 Do While/Loop 和 Do/Loop While 的结构与它们非常类似。F#没有 do/while，但有 while/do。

```

// C#
int i = 0;
while (i < 3)
{
    Console.WriteLine(i++);
}
i = 0;
do
{
    Console.WriteLine(i++);
} while (i < 3);

// C++/CLI
int i = 0;
while (i < 3)
{
    Console::WriteLine(i++);
}
i = 0;
do
{
    Console::WriteLine(i++);
} while (i < 3);

' Visual Basic
Dim num as Integer = 0
Do While (num < 3)
    Console.WriteLine(num)
    num += 1
Loop
num = 0
Do
    Console.WriteLine(num)
    num += 1
Loop While (num < 3)

// F#
let i = 0
let mutable looping = true

```

```

while looping do
    printfn "%d" i
    i++
    if i >= 3
        looping < - false

```

53.7.3 foreach 语句

foreach 语句使用 IEnumerable 接口。foreach 语句在 ANSI C++ 中不存在, 但它是 ANSI C++/CLI 中的一个扩展。与 C# 的 foreach 语句不同, 在 C++/CLI 中, for 和 each 之间有一个空格。F# 使用 for/in/do 关键字提供这个功能。

```

// C#
int[] arr = {1, 2, 3};
foreach (int i in arr)
{
    Console.WriteLine(i);
}

// C++/CLI
array<int>^ arr = {1, 2, 3};
for each (int i in arr)
{
    Console::WriteLine(i);
}

' Visual Basic
Dim arr() As Integer = New Integer() {1, 2, 3}
Dim num As Integer
For Each num as Integer In arr
    Console.WriteLine(num)
Next

// F#
for i in arr do
    printfn i

```



foreach 很容易遍历集合, 而 C# 允许使用 yield 语句创建枚举。在 Visual Basic 和 C++/CLI 中 yield 语句不可用, 而必须手工实现 IEnumerable 和 IEnumerator 接口。yield 语句参见第 6 章。

53.8 异常处理

异常处理详见第 15 章。这 3 种语言的异常处理非常类似。它们都使用 try/catch/finally 处理异常, 用 throw 关键字创建异常。F# 使用 raise 抛出异常, 使用 try/with/finally 处理异常:



可从
wrox.com
下载源代码

```

// C#
public void Method(Object o)
{
    if (o == null)

```

```

        throw new ArgumentException("Error");
    }
    public void Foo()
    {
        try
        {
            Method(null);
        }
        catch (ArgumentException ex)
        { }
        catch (Exception ex)
        { }
        finally
        { }
    }
}

```

代码段 CSharp/ExceptionDemo.cs



可从
wrox.com
下载源代码

```

// C++/CLI
public:
    void Method(Object^ o)
    {
        if (o == nullptr)
            throw gcnew ArgumentException("Error");
    }
    void Foo()
    {
        try
        {
            Method(nullptr);
        }
        catch (ArgumentException^ ex)
        { }
        catch (Exception^ ex)
        { }
        finally
        { }
    }
}

```

代码段 CPPCLI/ExceptionDemo.h



可从
wrox.com
下载源代码

```

' Visual Basic
Public Sub Method(ByVal o As Object)
    If o = Nothing Then
        Throw New ArgumentException("Error")
    End Sub
Public Sub Foo()
    Try
        Method(Nothing)
    Catch ex As ArgumentException
    ,
    Catch ex As Exception
    ,
    Finally
    ,
    End Try
End Sub

```



```
// F#
type ExceptionDemo() as this =
    member this.Method(o : obj) =
        if o = null then
            raise(ArgumentException("error"))
        member this.Foo =
            try
                try
                    this.Method(null)
                with
                    | :? ArgumentException as ex -> printfn "%s" ex.Message
                    | :? IOException as ex -> printfn "%s" ex.Message
            finally
                printfn "finally"
```

53.9 继承

.NET 提供了许多关键字，用于定义多态行为，用来重写或隐藏方法；以及访问修饰符，以允许访问或不允许访问类的成员。C#的这个功能参见第 4 章。C#、C++/CLI 和 Visual Basic 的功能非常类似，但关键字不同。

53.9.1 访问修饰符

C++/CLI、Visual Basic 和 F#的访问修饰符非常类似于 C#，但有一些显著的区别。Visual Basic 使用 Friend 访问修饰符替代 internal，访问同一个程序集中的类型。C++/CLI 还有一个访问修饰符 protected private。internal protected 允许访问同一个程序集中的成员，还可以访问其他程序集中派生自基类的类型。C#和 Visual Basic 不允许访问同一个程序集中的派生类型，但在 C++/CLI 中这可以使用 protected private 来实现。这里 private 表示在程序集的外部不能访问，但在程序集的内部可以进行受保护的访问。其顺序(不管是 protected private 或 private protected)并不重要。访问修饰符在程序集中总是允许访问较多的内容，在程序集的外部总是允许访问较少的内容。F#使用 3 个访问修饰符，还允许把 protected 修饰符用于在其他.NET 语言中使用的类型。F#中的访问修饰符可以在签名文件中声明。

每种语言的访问修饰符见表 53-3。

表 53-3

C#	C++/CLI	Visual Basic	F#
public	Public	Public	public
protected	Protected	Protected	
private	Private	Private	private
internal	Internal	Friend	internal
internal protected	internal protected	Protected Friend	
无	protected private	无	

F#签名文件的扩展名是.fsi，它定义成员的签名。这种文件可以从代码文件中通过-- sig 编译器选项自动创建。根据不同的需求，只需要更改访问修饰符：

```
// F#
type public Person =
    class
        interface IDisplay
            public new : unit -> Person
            public new : firstName:string * lastName:string -> Person
            override ToString : unit -> string
            member public FirstName : string
            member public LastName : string
            member public FirstName : string with set
            member public LastName : string with set
        end
```

53.9.2 关键字

对继承很重要的关键字如表 53-4 所示。

表 53-4

C#	C++/CLI	Visual Basic	F#	功 能
:	:	Implements	interface with	实现一个接口
:	:	Inherits	inherit	继承自一个基类
virtual	virtual	Overridable	abstract	声明一个支持多态性的方法
overrides	overrides	Overrides	override	重写一个虚方法
new	new	Shadows		隐藏基类中的方法
abstract	abstract	MustInherit	[< AbstractClass]	抽象类
sealed	sealed	NotInheritable	[< Sealed >]	密封类
abstract	abstract	MustOverride	abstract	抽象方法
sealed	sealed	NotOverridable		密封方法
this	this	Me		引用当前对象
base	Classname::	MyBase	base	引用基类

关键字的放置顺序在各种语言中很重要。在代码示例中，抽象基类 Base 有一个抽象方法和一个已实现的虚方法。Drived 类派生自 Base，它实现抽象方法，并重写虚方法。



可从
wrox.com
下载源代码

```
// C#
public abstract class Base
{
    public virtual void Foo()
    {
    }
    public abstract void Bar();
}
public class Derived: Base
{
}
```

```

        public override void Foo()
        {
            base.Foo();
        }
        public override void Bar()
        {
        }
    }
}

```

 代码段 CSharp/InheritanceDemo.cs


可从
wrox.com
下载源代码

```

// C++/CLI
public ref class Base abstract
{
public:
    virtual void Foo()
    {
    }
    virtual void Bar() abstract;
};
public ref class Derived: public Base
{
public:
    virtual void Foo() override
    {
        Base::Foo();
    }
    virtual void Bar() override
    {
    }
};

```

 代码段 CPPCLI/InheritanceDemo.h


可从
wrox.com
下载源代码

```

' Visual Basic
Public MustInherit Class Base
    Public Overridable Sub Foo()
    End Sub
    Public MustOverride Sub Bar()
End Class
Public Class Derived
    Inherits Base
    Public Overrides Sub Foo()
        MyBase.Foo()
    End Sub
    Public Overrides Sub Bar()
    End Sub
End Class

```

 代码段 VisualBasic/InheritanceDemo.vb


可从
wrox.com
下载源代码

```

// F#
[<AbstractClass>]
type Base() as this =
    abstract Foo : unit -> unit
    default this.Foo() = printfn "Base.Foo"
    abstract Bar : unit -> unit

```

```

type Derived() as this =
    inherit Base()
    override this.Foo() =
        base.Foo()
        printfn "Derived.Foo"
    override this.Bar() = printfn "Derived.Bar"

```

代码段 FSharp/InheritanceDemo.fs

53.10 资源管理

第 13 章介绍了资源管理,同时实现 `IDisposable` 接口和一个终结器。本节介绍这些功能在 C++/CLI、Visual Basic 和 F#中如何实现。

53.10.1 IDisposable 接口的实现

为了释放资源, `IDisposable` 接口定义 `Dispose()` 方法。使用 C#、Visual Basic 和 F#时, 必须实现 `IDisposable` 接口。在 C++/CLI 中, 也实现 `IDisposable` 接口, 但如果只编写一个析构函数, 这将由编译器完成。



```

// C#
public class Resource: IDisposable
{
    public void Dispose()
    {
        // release resource
    }
}

```

代码段 CSharp/Resource.cs



```

// C++/CLI
public ref class Resource
{
public:
    ~Resource()
    {
        // release resource
    }
};

```

代码段 CPPCLI/Resource.h



```

' Visual Basic
Public Class Resource
    Implements IDisposable
    Public Sub Dispose() Implements IDisposable.Dispose
        ' release resource
    End Sub
End Class

```

代码段 VisualBasic/Resource.vb

```

// F#
type Resource() as this =

```

```
interface IDisposable with
member this.Dispose() = printfn "release resource"
```

代码段 FSharp/Resource.fs



在 C++/CLI 中，使用 `delete` 语句调用 `Dispose()` 方法。

53.10.2 using 语句

C# 的 `using` 语句使用“获取/使用/释放”模式释放不再使用的资源，甚至在出现异常的情况下也是如此。编译器创建一个 `try/finally` 语句，在 `finally` 语句中调用 `Dispose()` 方法。Visual Basic 支持类似于 C# 的 `using` 语句。C++/CLI 针对这个问题提供了更好的方法。如果引用类型在本地声明，编译器就创建一条 `try/finally` 语句，在该块的最后调用 `Dispose()` 方法。F# 提供了两种不同的结构，轻松地支持资源管理。在值超出作用域时，`use` 绑定会自动调用 `Dispose()` 方法。`using` 表达式创建的对象必须释放，并且在所调用函数的末尾释放。下面的 `bar()` 函数在创建 `Resource` 对象后调用：

```
// C#
using (Resource r = new Resource())
{
    r.Foo();
}

// C++/CLI
{
    Resource r;
    r.Foo();
}

' Visual Basic
Using r As New Resource
    r.Foo()
End Using

// F# use binding
let rs =
    use r = new Resource()
    r.Foo()
    // r.Dispose called implicitly
// F# using expression
let bar (r : Resource) =
    r.Foo()

using (new Resource()) bar
```

53.10.3 重写 `Finalize()` 方法

如果类包含必须释放的本地资源，该类就必须重写 `Object` 类中的 `Finalize()` 方法。在 C# 中，这只需编写一个析构函数。C++/CLI 采用一种特殊的语法：用“!”前缀定义终结器。在终结器中，也不允许释放有终结器的对象，以确保终结的顺序。这就是 `Dispose` 模式定义一个带布尔参数的 `Dispose()` 方法的原因。在 C++/CLI 中，不需要在代码中实现这种模式，因为这由编译器完成。C++/CLI 析构函

数实现两个 Dispose()方法。在 Visual Basic 中, Dispose()方法和终结器都必须手工实现。但是大多数 Visual Basic 类都不直接使用本地资源,而是借助于其他类。在 Visual Basic 中,通常不需要重写 Finalize()方法,但一般需要实现 Dispose()方法。



在 C#中,编写析构函数重写基类的 Finalize()方法。C++/CLI 的析构函数实现 IDisposable 接口。



可从
wrox.com
下载源代码

```
// C#
public class Resource: IDisposable
{
    ~Resource // override Finalize
    {
        Dispose(false);
    }
    protected virtual void Dispose(bool disposing)
    {
        if (disposing) // dispose embedded members
        {
        }
        // release resources of this class
        GC.SuppressFinalize(this);
    }
    public void Dispose()
    {
        Dispose(true);
    }
}
```

代码段 CSharp/Resource.cs



可从
wrox.com
下载源代码

```
// C++/CLI
public ref class Resource
{
public:
    ~Resource() // implement IDisposable
    {
        this->!Resource();
    }
    !Resource() // override Finalize
    {
        // release resource
    }
};
```

代码段 CPPCLI/Resource.h



可从
wrox.com
下载源代码

```
' Visual Basic
Public Class Resource
    Implements IDisposable
    Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
```

```

Protected Overridable Sub Dispose(ByVal disposing)
    If disposing Then
        ' Release embedded resources
    End If
    ' Release resources of this class
End Sub
Protected Overrides Sub Finalize()
    Try
        Dispose(False)
    Finally
        MyBase.Finalize()
    End Try
End Sub
End Class

```

代码段 VisualBasic/Resource.vb

53.11 委托

第 8 章讨论的委托是方法类型安全的指针。在这 4 种语言中，都可以使用 `delegate` 关键字定义委托，但使用委托的方式有区别。

示例代码使用的 `Demo` 类有一个静态方法 `Foo()` 和一个实例方法 `Bar()`。这两个方法都由 `DemoDelegate` 类型的委托实例调用。把 `DemoDelegate` 声明为调用一个返回类型为 `void` 并且带 `int` 参数的方法。

C#使用委托支持委托推断，其中编译器会创建一个委托实例，传递方法的地址。

在 C#和 C++/CLI 中，可以使用 “+” 运算符将两个委托合二为一：



```

// C#
public delegate void DemoDelegate(int x);
public class Demo
{
    public static void Foo(int x) { }
    public void Bar(int x) { }
}

Demo d = new Demo();
DemoDelegate d1 = Demo.Foo;
DemoDelegate d2 = d.Bar;
DemoDelegate d3 = d1 + d2;
d3(11);

```

代码段 CSharp/DelegateSample.cs

C++/CLI 不支持委托推断。C++/CLI 需要创建一个委托类型的新实例，并将方法的地址传递给构造函数。



```

// C++/CLI
public delegate void DemoDelegate(int x);
public ref class Demo
{
public:
    static void Foo(int x) { }
    void Bar(int x) { }
}

```

```
};
Demo^ d = gcnew Demo();
DemoDelegate^ d1 = gcnew DemoDelegate( &Demo::Foo);
DemoDelegate^ d2 = gcnew DemoDelegate(d, &Demo::Bar);
DemoDelegate^ d3 = d1 + d2;
d3(11);
```

代码段 CPPCLI/DelegateSample.h

与 C++/CLI 类似，Visual Basic 也不支持委托推断。必须创建一个委托类型的新实例，传递方法的地址。Visual Basic 使用 AddressOf 运算符传递方法的地址。

因为 Visual Basic 没有为委托重载 “+” 操作符，所以需要调用 Delegate 类的 Combine() 方法。把 Delegate 类写在方括号中，因为 Delegate 是一个 Visual Basic 关键字，不能使用同名的类。给 Delegate 加上方括号，可确保使用 Delegate 类，而不使用 Delegate 关键字。



```
' Visual Basic
Public Delegate Sub DemoDelegate(ByVal x As Integer)
Public Class Demo
    Public Shared Sub Foo(ByVal x As Integer)
    '
    End Sub
    Public Sub Bar(ByVal x As Integer)
    '
    End Sub
End Class
Dim d As New Demo()
Dim d1 As New DemoDelegate(AddressOf Demo.Foo)
Dim d2 As New DemoDelegate(AddressOf d.Bar)
Dim d3 As DemoDelegate = [Delegate].Combine(d1, d2)
d3(11)
```

代码段 VisualBasic/DelegateDemo.vb

在 F# 中，作为对象的函数是头等成员。与其他 .NET 语言的交互操作性一样，由于 F# 也需要委托，因此语法看起来比较复杂。在 F# 中声明委托时，需要把委托关键字赋予类型名，并定义参数和返回类型。对于 DemoDelegate() 方法，参数的类型是 int，返回类型是与 C# 中的 void 类似的 unit。

用一个静态成员 Foo() 和一个实例成员 Bar() 定义 Demo 类型。这两个成员都满足委托类型的要求。

变量 d1 是引用 Demo.Foo() 方法的委托变量，d2 引用实例方法 Bar()。InvokeDelegate() 函数的参数是 DemoDelegate 和 int，它声明为通过委托调用引用的函数。它调用 Delegate 类的 Invoke() 方法，并传递 int 参数。在声明 invokeDelegate() 函数后，就传递委托实例 d1 和值 33，调用该函数。为了合并委托，需要调用 Delegate.Combine() 方法。因为 Combine() 方法需要两个 Delegate 参数，并返回一个 Delegate 类型，所以需要强制向上转换和强制向下转换。这里 F# 的语法是使用 “:>” 把 d1 强制转换为基类型 Delegate，使用 “:?>” 把 Delegate 类型 d3 强制转换为派生类 DemoDelegate。除了使用 “:>” 和 “:?>” 之外，还可以使用 upcast 和 downcast 关键字。



```
// F#
type DemoDelegate = delegate of int -> unit
type Demo() as this =
    static member Foo(x : int) =
```

```

        printfn "Foo %d" x
        member this.Bar(x : int) =
            printfn "Bar %d" x

// F# using delegate
let d = Demo()
let d1 : DemoDelegate = new DemoDelegate(Demo.Foo)
let invokeDelegate (dlg : DemoDelegate) (x : int) =
    dlg.Invoke(x)
(invokeDelegate d1 33)
let d2 : DemoDelegate = new DemoDelegate(d.Bar)
let d11 = d1 :> Delegate
let d22 = d2 :> Delegate
let d33 : Delegate = Delegate.Combine(d11, d22)
let d3 : d33 :?> DemoDelegate
(invokeDelegate d3 11)

```

代码段 FSharp/DelegateDemo.fs

53.12 事件

使用 `event` 关键字可以实现基于委托的订阅机制。这 4 种语言都定义了 `event` 关键字，提供类中的事件。下面的 `EventDemo` 类引发 `DemoDelegate` 类型的 `DemoEvent` 事件。

在 C# 中，引发事件的语法类似于事件的方法调用。只要没有人注册事件，事件变量就是 `null`，所以在引发事件之前要检查事件变量是否为 `null`。在注册处理程序方法时，要使用 “+=” 运算符，在委托推断的帮助下传递处理程序方法的地址。



```

// C#
public class EventDemo
{
    public event DemoDelegate DemoEvent;
    public void FireEvent()
    {
        if (DemoEvent != null)
            DemoEvent(44);
    }
}

public class Subscriber
{
    public void Handler(int x)
    {
        // handler implementation
    }
}

//...
EventDemo evd = new EventDemo();
Subscriber subscr = new Subscriber();
evd.DemoEvent += subscr.Handler;
evd.FireEvent();

```

代码段 CSharp/EventDemo.cs

C++/CLI 与 C# 非常类似，除了触发事件不需要先查看事件变量不为 `null`。这由编译器创建的 IL

代码自动完成。



C#和 C++/CLI 使用 “+=” 运算符来注销事件。



可从
wrox.com
下载源代码

```
// C++/CLI
public ref class EventDemo
{
public:
    event DemoDelegate^ DemoEvent;
    public void FireEvent()
    {
        DemoEvent(44);
    }
}
public class Subscriber
{
public:
    void Handler(int x)
    {
        // handler implementation
    }
}
//...
EventDemo^ evd = gcnew EventDemo();
Subscriber^ subscr = gcnew Subscriber();
evd-> DemoEvent += gcnew DemoDelegate(subscr, &Subscriber::Handler);
evd-> FireEvent();
```

代码段 CPPCLI/EventDemo.h

Visual Basic 使用不同的语法。用 Event 关键字声明事件，这与 C#和 C++/CLI 相同。但是，用 RaiseEvent 语句引发事件。RaiseEvent 语句检查事件变量是否用订阅器初始化。用于注册处理程序的 Addhandler 语句与 C#中的 “+=” 运算符有相同的功能。Addhandler 语句需要两个参数：第一个参数定义事件，第二个参数定义处理程序的地址。RemoveHandler 语句用于从事件中注销处理程序。



可从
wrox.com
下载源代码

```
' Visual Basic
Public Class EventDemo
    Public Event DemoEvent As DemoDelegate
    public Sub FireEvent()
        RaiseEvent DemoEvent(44);
    End Sub
End Class
Public Class Subscriber
    Public Sub Handler(ByVal x As Integer)
        ' handler implementation
    End Sub
End Class
'...
Dim evd As New EventDemo()
Dim subscr As New Subscriber()
AddHandler evd.DemoEvent, AddressOf subscr.Handler
evd.FireEvent()
```

Visual Basic 提供的另一种语法对于其他语言不可用：对订阅事件的方法使用 **Handles** 关键字。它要求用 **WithEvents** 关键字定义一个变量：

```
Public Class Subscriber
    Public WithEvents evd As EventDemo
    Public Sub Handler(ByVal x As Integer) Handles evd.DemoEvent
        ' Handler implementation
    End Sub
    Public Sub Action()
        evd = New EventDemo()
        evd.FireEvent()
    End Sub
End Class
```

53.13 泛型

这 4 种语言都支持泛型的创建和使用。泛型参见第 5 章。

为了使用泛型，C#借用了 C++模板的语法，用尖括号定义泛型类型。C++/CLI 使用相同的语法。在 Visual Basic 中，泛型类型用圆括号中的 **Of** 关键字定义。F#非常类似于 C#。



```
// C#
List<int> intList = new List<int>();
intList.Add(1);
intList.Add(2);
intList.Add(3);
```

代码段 CSharp/GenericsDemo.cs



```
// C++/CLI
List<int>^ intList = gcnew List<int>();
intList->Add(1);
intList->Add(2);
intList->Add(3);
```

代码段 CPPCLI/GenericsDemo.h



```
' Visual Basic
Dim intList As List(Of Integer) = New List(Of Integer)()
intList.Add(1)
intList.Add(2)
intList.Add(3)
```

代码段 VisualBasic/GenericsDemo.vb



```
// F#
let intList =
    new List<int>()
intList.Add(1)
intList.Add(2)
intList.Add(3)
```

代码段 FSharp/GenericsDemo.fs

因为类声明使用尖括号，所以编译器知道要创建一个泛型类型。用 `where` 子句定义约束。



```
public class MyGeneric<T>
    where T: IComparable<T>
{
    private List<T>list = new List<T>();
    public void Add(T item)
    {
        list.Add(item);
    }
    public void Sort()
    {
        list.Sort();
    }
}
```

代码段 CSharp/GenericsDemo.cs

用 C++/CLI 定义泛型类型与用 C++ 定义模板类似。但 C++/CLI 不使用 `template` 关键字，而对于泛型使用 `generic` 关键字。`where` 子句类似于 C# 中的 `where` 子句，但 C++/CLI 没有构造函数的限制。



```
generic <typename T>
where T: IComparable<T>
ref class MyGeneric
{
private:
    List<T>^ list;
public:
    MyGeneric()
    {
        list = gcnew List<T>();
    }
    void Add(T item)
    {
        list-> Add(item);
    }
    void Sort()
    {
        list-> Sort();
    }
};
```

代码段 CPPCLI/GenericsDemo.h

Visual Basic 用 `Of` 关键字定义泛型类，用 `As` 定义约束。



```
Public Class MyGeneric(Of T As IComparable(Of T))
    Private myList = New List(Of T)
    Public Sub Add(ByVal item As T)
        myList.Add(item)
    End Sub
    Public Sub Sort()
        myList.Sort()
    End Sub
End Class
```

代码段 VisualBasic/GenericsDemo.vb

F#用尖括号定义泛型类型。泛型类型用一个撇号标记。约束用 `when` 关键字定义。约束跟在 `when` 的后面。在下面的示例中，使用“`:>`”类型约束。它指定，泛型类型必须派生自接口或基类。也可以定义值类型约束(`: struct`)、引用类型约束(`: not struct`)和默认构造函数约束(`'a : (new : unit -> 'a)`)。不可用于其他语言的约束是空约束(`'a : null`)。这个约束指定 `'a` 泛型类型必须是可空的，这就允许使用所有 .NET 引用类型，但不能使用不能为空的 F#类型：



可从
wrox.com
下载源代码

```
type MyGeneric<'a> when 'a :> IComparable<'a>() as this =
    let list = new List<'a>()
    member this.Add(item : 'a) = list.Add(item)
    member this.Sort() = list.Sort()
```

代码段 FSharp/GenericsDemo.fs

53.14 LINQ 查询

语言集成的查询是 C#和 Visual Basic 的一个功能。这两种语言的语法非常类似：



LINQ 参见第 11 章。



可从
wrox.com
下载源代码

```
// C#
var query = from r in racers
            where r.Country == "Brazil"
            orderby r.Wins descending
            select r;
```

代码段 CSharp/LinqSample.cs



可从
wrox.com
下载源代码

```
' Visual Basic
Dim query = From r in racers _
            Where r.Country = "Brazil" _
            Order By r.Wins Descending _
            Select r
```

代码段 VisualBasic/LinqSample.vb



C++/CLI 不支持 LINQ 查询。

53.15 C++/CLI 混合本地代码和托管代码

C++/CLI 的一大优点是混合本地代码和托管代码。使用 C#中的本地代码需要通过一种机制(称为平台调用)。平台调用参见第 26 章。在 C++/CLI 中使用本地代码切实可行。

在托管类中，可以使用本地代码和托管代码，如下所示。本地类也是如此。可以在一个方法中混合使用本地代码和托管代码。



```
#pragma once
#include <iostream> // include this header file for cout
using namespace std; // the iostream header defines the namespace std
using namespace System;
public ref class Managed
{
public:
    void MixNativeAndManaged()
    {
        cout << "Native Code" << endl;
        Console::WriteLine("Managed Code");
    }
};
```

代码段 CPPCLI/Mixing.h

在托管类中，还可以声明本地类型的字段或本地类型的指针。但不能在本地类中声明托管类型的字段或托管内型的指针。必须注意，可以在垃圾收集器清理内存时删除托管类型的实例。

为了将托管类用作本地类中的成员，C++/CLI 定义了 `gcroot` 关键字，它在头文件 `gcroot.h` 中定义。`gcroot` 关键字包装一个 `CGHandle`，`CGHandle` 用于跟踪本地引用中的 CLR 对象。

```
#pragma once
#include "gcroot.h"
using namespace System;
public ref class Managed
{
public:
    Managed() { }
    void Foo()
    {
        Console::WriteLine("Foo");
    }
};
public class Native
{
private:
    gcroot<Managed^> m_p;
public:
    Native()
    {
        m_p = gcnew Managed();
    }
    void Foo()
    {
        m_p -> Foo();
    }
};
```

53.16 C#的特殊功能

一些 C# 语法功能没有在本章中介绍。C# 定义了 `yield` 语句，它便于创建枚举器。这条语句对于 C++/CLI 和 Visual Basic 不可用。在这些语言中，必须手动实现枚举器。另外，C# 还为可空类型定

义了特殊的语法，而在其他语言中，必须使用泛型结构 `Nullable<T>`。

C#允许使用不安全的代码块，在不安全的代码块中可以使用指针和指针算术。这个功能非常有利于调用本地库中的方法。`Visual Basic` 没有这个功能，这是 C#的一个优势。`C++/CLI` 不需要使用 `unsafe` 关键字定义不安全的代码块，`C++/CLI` 本身就混合了本地代码和托管代码。

53.17 小结

本章学习了如何将 C#的语法映射到 `Visual Basic`、`C++/CLI` 和 `F#`上。`C++/CLI` 定义了对 `C++`的扩展，以编写 .NET 应用程序，并利用 C#进行语法扩展。尽管 C#和 `C++/CLI` 的根源相同，但有许多重要的区别。`Visual Basic` 没有使用花括号，但比较琐碎。`F#`的语法完全不同，因为它主要关注函数编程。

在语法映射上，本章探讨了如何把 C#语法映射到 `Visual Basic`、`C++/CLI` 和 `F#`上，介绍了其他 3 种语言的语法，如何定义类型、方法、属性，哪些关键字用于 OO 功能，资源管理如何进行，委托、事件和泛型在 4 种语言中如何实现。

尽管可以映射大多数语法，但这些语言的功能仍有区别。

第 54 章

.NET Remoting

本章内容:

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

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

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

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

54.1 使用 .NET Remoting 的原因

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

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

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

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

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

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

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



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

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

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



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

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



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

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

54.2 .NET Remoting 术语详解

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

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

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

下面列出了.NET Remoting 体系结构的主要元素:

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

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

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

图 54-1 展示了如何把各个元素联系在一起。

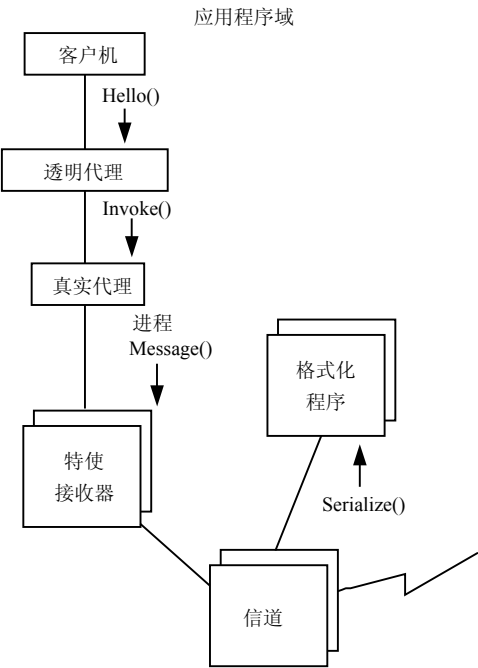


图 54-1

54.2.1 客户端通信

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

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

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

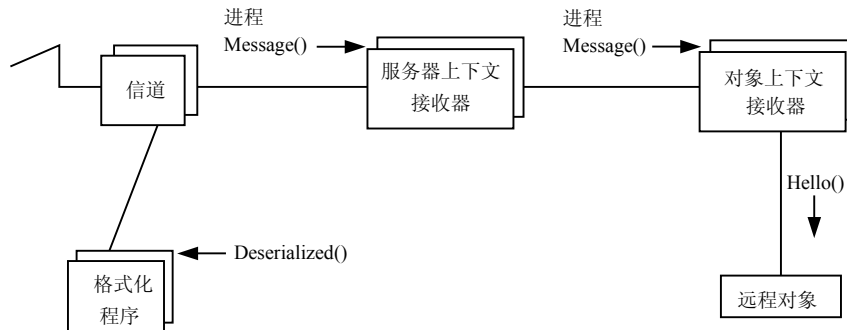


图 54-2

54.2.2 服务器端通信

图 54-2 展示了服务器端的消息传递：

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

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

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



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

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

54.3 上下文

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

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

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

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

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

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

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

为了更好地理解上下文，必须知道下面的术语：

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

54.3.1 激活

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

54.3.2 特性和属性

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

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

使用这个特性的构造函数，可以设置以下 4 个值中的 1 个值：

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

54.3.3 上下文之间的通信

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

54.4 远程对象、客户端和服务端

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

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

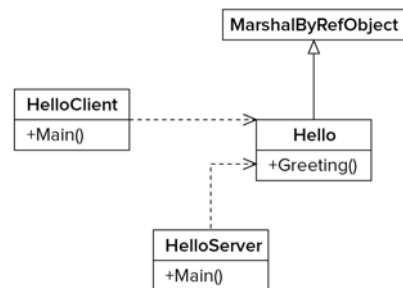


图 54-3

54.4.1 远程对象

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

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

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

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

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



```
using System;

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

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

代码段 RemoteHello/RemoteHello/Hello.cs

54.4.2 简单的服务器应用程序

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

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

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



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

在远程对象注册之后，有必要使服务器一直处于运行状态，直到按任意键为止：



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

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

```

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

```

代码段 RemoteHello/HelloServer/Program.cs

54.4.3 简单的客户端应用程序

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

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



```

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

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

            ChannelServices.RegisterChannel(new TcpClientChannel(), true);
            Hello obj = (Hello)Activator.GetObject(
                typeof(Hello), "tcp://localhost:8086/Hi");

            if (obj == null)
            {
                Console.WriteLine("could not locate server");
                return;
            }
            for (int i=0; i < 5; i++)
            {
                Console.WriteLine(obj.Greeting("Stephanie"));
            }
        }
    }
}

```

代码段 RemoteHello/HelloClient/Program.cs



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

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

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

54.5 .NET Remoting 体系结构

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

本节将论述下面的主题:

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

54.5.1 信道

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

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

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

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

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

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

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

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

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



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

HTTP 信道的使用方式类似于 TCP 信道。可以指定服务器能在哪个端口上创建侦听套接字。服务器可以侦听多个信道。下面的代码创建并注册 HTTP、TCP 和 IPC 信道：

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

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

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

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

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

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

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

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

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

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

1. 设置信道属性

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

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

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

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

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

2. 信道的“可插入性”

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

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

54.5.2 格式化程序

.NET Framework 提供了两个格式化程序类，即：

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

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

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

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

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

所有这些接收器和提供程序类都可以扩展，并可以被自定义实现方式替代。

54.5.3 ChannelServices 和 RemotingConfiguration

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

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

下面是注册 HTTP、TCP 和 IPC 信道的服务器代码：

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

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

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

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

1. 知名对象的服务器

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

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

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

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

2. 客户端激活的对象的服务器

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

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

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

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

54.5.4 对象的激活

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

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

1. 应用程序的 URL

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

protocol://server:port/URI

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

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

2. 激活知名对象

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

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

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

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

指定。

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

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

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

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

3. 激活客户端激活的对象

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

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

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

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

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

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

Hello obj = new Hello();
```

4. 代理对象

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

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

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

if (obj == null)
{
    Console.WriteLine("could not locate server");
    return;
}
if (RemotingServices.IsTransparentProxy(obj))
{
    Console.WriteLine("Using a transparent proxy");
    RealProxy proxy = RemotingServices.GetRealProxy(obj);

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

5. 代理的“可插入性”

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

6. 消息

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

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

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

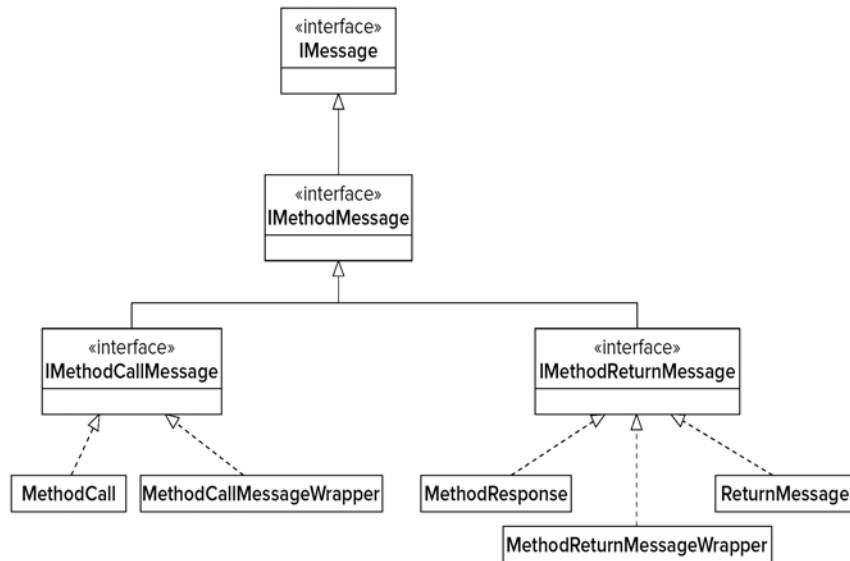


图 54-4

54.5.5 消息接收器

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

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

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

下面几节讨论可以使用的 3 个不同的消息接收器。

1. 特使接收器

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

2. 服务器上下文接收器

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

3. 对象接收器

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

54.5.6 在远程方法中传递对象

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

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

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



```
using System;

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

```

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

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

```

代码段 PassingObjects/RemoteHello/Hello.cs

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



```

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

```

代码段 PassingObjects/HelloClient/Program.cs

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

Press return after the server is started

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

1. 安全性和序列化的对象

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

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

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

下面的代码说明了如何利用完整级别的序列化支持创建一个 TCP 信道：



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

var clientProvider = new BinaryClientFormatterSinkProvider();

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

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

代码段 `PassingObjects/HelloServer/Program.cs`

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

2. 方向特性

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

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



有关 `ref` 和 `out` 关键字的内容详见第 3 章。

54.5.7 生命周期管理

客户端和服务端怎样检测到另一端是否可用？此时，我们遇到的问题是什么呢？

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

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

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

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

1. 租约的续约

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

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

2. 租约的配置值

可以配置下面的一些值：

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

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

表 54-1 中列出了它们的默认值。

表 54-1

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

3. 管理生命周期所使用的类

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

4. 获取租约信息示例

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

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

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

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

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

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

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

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

```

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

```

5. 更改默认的租约配置

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

```

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

```

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

```

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

```

使用后面讨论的配置文件，也可以完成生命周期服务的配置。

54.6 配置文件

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

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

下面就是一个配置文件的示例：



```

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

```

```

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

```

代码段 ConfigurationFiles/HelloServer/app.config

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

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

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

```

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

```

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

```

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

```

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

```

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

```

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

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

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



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

54.6.1 知名对象的服务器配置

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



可从
wrox.com
下载源代码

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

代码段 `ConfigurationFiles/HelloServer/Wellknown.config`

54.6.2 知名对象的客户端配置

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



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

代码段 ConfigurationFiles/HelloClient/Wellknown_Client.config

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



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

代码段 ConfigurationFiles/HelloClient/WellknownHttp_Client.config

54.6.3 客户端激活的对象的服务器配置

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



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

代码段 ConfigurationFiles/HelloServer/ClientActivated.config

54.6.4 客户端激活的对象的客户端配置

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



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

代码段 ConfigurationFiles/HelloClient/ClientActivated_Client.config

54.6.5 使用配置文件的服务器代码

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



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

代码段 ConfigurationFiles/HelloServer/Program.cs

这两个函数显示了知名类型和客户端激活的类型的配置信息:

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

```

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

```

54.6.6 使用配置文件的客户端代码

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

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



```

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

```

代码段 ConfigurationFiles/HelloClient/Program.cs

54.6.7 客户端信道的延迟加载

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

```

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

```

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

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

54.6.8 调试配置

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

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

54.6.9 配置文件中的生命周期服务

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

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

```

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

```

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

54.6.10 格式化程序提供程序

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

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

```

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

```

54.7 在 ASP.NET 中驻留远程服务器

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

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

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

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

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

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



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

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

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



ASP.NET 中驻留的远程对象只支持知名对象,不支持客户端激活的对象。

54.8 类、接口和 Soapsuds

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

54.8.1 接口

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

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

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

54.8.2 Soapsuds

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

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

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

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

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

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

表 54-2

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

(续表)

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

54.9 异步远程调用

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

54.9.1 使用委托和.NET Remoting

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

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

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

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

```

        {
            greeting = d.EndInvoke(ar);
        }

        Console.WriteLine(greeting);
    }
}
}

```

第 8 章详细介绍了委托，第 20 章介绍了如何异步地使用委托。

54.9.2 OneWay 特性

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

```

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

```

54.10 .NET Remoting 的安全性

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

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

首先介绍服务器配置。在下面摘录的配置文件中，显示了如何为服务器定义安全性。

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

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

```

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

```

```

        impersonate="false" />
    </channels>

```

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

`protectionLevel` 可以设置为与服务器配置文件相同的选项。

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

```

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

```

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

```

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

var clientChannel = new TcpClientChannel(dict, null);

```

安全性详见第 21 章。

54.11 远程处理和事件

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

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

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

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

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

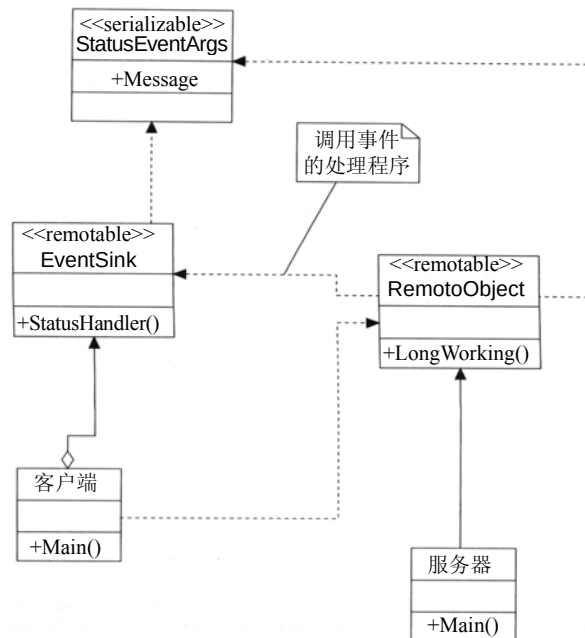


图 54-5

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

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

54.11.1 远程对象

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

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

<StatusEventArgs>委托定义的参数可以满足这些要求:

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



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

代码段 RemotingAndEvents/RemoteObject/RemoteObject.cs

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

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

54.11.2 事件参数

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



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

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

代码段 RemotingAndEvents/RemoteObject/RemoteObject.cs

54.11.3 服务器

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



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

代码段 RemotingAndEvents/RemoteServer/Program.cs

54.11.4 服务器配置文件

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



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

代码段 RemotingAndEvents/RemoteServer/app.config

54.11.5 事件接收器

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



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

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

代码段 RemotingAndEvents/EventSink/EventSink.cs

54.11.6 客户端

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



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

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

代码段 RemotingAndEvents/RemoteClient/Program.cs

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

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

```

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

```

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

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

```

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

```

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

```

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

```

54.11.7 客户端配置文件

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



```

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

```

代码段 `RemotingAndEvents/RemoteClient/app.config`

54.11.8 运行程序

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

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

在客户端，可以看到跨网络触发的事件。

```
wait for server...

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

54.12 调用上下文

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

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

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

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

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

```

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

```

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

```

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

```

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

```

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

```

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

54.13 小结

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

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

而在客户端上，只需创建信道和使用远程对象。

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

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

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

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

第 55 章

Web 服务和 ASP.NET

本章的内容如下：

- SOAP 和 WSDL 的语法
- 如何通过 Web 服务使用 SOAP 和 WSDL
- 提供和使用 Web 服务
- Web 服务的用法。
- 使用 SOAP 标题交换数据

Web 服务是利用 SOAP(Simple Object Access Protocol, 简单对象访问协议)在 HTTP 上执行远程方法调用的一种新方法。过去这个问题一直非常棘手, 因为使用过任何 DCOM(分布式 COM)的人们, 在实例化远程服务器上的对象、调用方法和获取结果时感到非常麻烦, 并且在进行必要的配置时, 需要具有很高的技巧。

SOAP 的出现使事情变得简单多了。SOAP 技术是一个基于 XML 的标准, 它详细描述了怎样在 HTTP 上以可重复的方式进行方法调用。远程 SOAP 服务器能够理解这些调用并执行所有困难的工作, 如实例化所需的对象、进行调用以及给客户端返回 SOAP 格式的响应等。

通过 .NET Framework, 可以非常容易地利用上述技术。与 ASP.NET 一样, 我们可以在服务器上使用完整的 C# 和 .NET 技术, 而且(也许是更重要的)可以从任何平台上通过 HTTP 访问服务器, 从而实现的 Web 服务的简单利用。换句话说, 例如, Linux 代码就可以使用 .NET Web 服务, 或者 Internet 启用的电冰箱。过去作者就曾经成功地把 ASP.NET Web 服务和 Macromedia Flash 组合在一起, 创建启用数据的 Flash 内容。

此外, 也可以使用 WSDL(Web Service Description Language, Web 服务描述语言)完整地描述 Web 服务, 还可以在运行期间动态地查找 Web 服务。WSDL 使用带有 XML 架构的 XML 提供对所有方法的描述(以及对调用方法所需类型的描述)。现在各式各样的类型可用于 Web 服务, 既有简单的基元类型, 又有完整的 DataSet 对象, 这样, 完全存储在内存中的数据库就可以被编组到客户端, 从而大大减少加数据库服务器上加载的数据量。



注意, 本章讨论的是 ASP.NET Web 服务, 而不是 WCF Web 服务, 后者是近期才添加到 .NET 中。ASP.NET Web 服务使用起来比较简单, 足以满足大多数需要, 而 Windows Communication Foundation(WCF) Web 服务包含 ASP.NET Web 服务的全部功能, 还添加了额外的功能。WCF 详解第 43 章。

55.1 SOAP

如前所述, SOAP 是一个与 Web 服务交换数据的方法。有关这项技术的书有很多, 尤其是微软公司决定在 .NET Framework 中采用这项技术之后, SOAP 方面的书就更多了。稍微考虑一下, 可以发现 SOAP 的工作原理和 HTTP 的工作原理比较相似, 这非常有趣, 但并不是必需的。大多数情况下, 我们不必考虑与 Web 服务进行交换时所采用的格式, 只要得到希望的结果就够了。

因此, 本节不深入探讨 SOAP 的技术细节, 而是给出一些简单的 SOAP 请求和响应, 以便您对 SOAP 有一个感性的认识。

假定要用下面的签名调用 Web 服务中的方法:

```
int DoSomething(string stringParam, int intParam)
```

这条语句必需的 SOAP 标题和主体如下所示, 最上面是 Web 服务的地址:

```
POST /SomeLocation/myWebService.asmx HTTP/1.1
Host: hostname
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/DoSomething"
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <DoSomething xmlns="http://tempuri.org/">
      <stringParam>string</stringParam>
      <intParam>int</intParam>
    </DoSomething>
  </soap:Body>
</soap:Envelope>
```

length 参数用于指定内容的总字节数, 它的大小随着 string 和 int 参数中发送的值而变化。Host 也是变化的, 它取决于 Web 服务的位置。

上面代码引用的 soap 名称空间定义用于构建消息的各种元素。通过 HTTP 发送上面的代码时, 实际发送的数据将有所不同(但是相关)。例如, 可以使用简单的 GET 方法调用上面的方法:

```
GET /SomeLocation/myWebService.asmx/DoSomething?stringParam= string & intParam= int
HTTP/1.1
Host: hostname
```

这个方法的 SOAP 响应如下:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
```

```

<soap:Body>
  <DoSomethingResponse xmlns="http://tempuri.org/">
    <DoSomethingResult>int</DoSomethingResult>
  </DoSomethingResponse>
</soap:Body>
</soap:Envelope>

```

其中 `length` 参数的值根据 `int` 参数值的变化而改变。

此外，通过 HTTP 的实际响应也比较简单，例如：

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version="1.0"?>
<int xmlns="http://tempuri.org/">int</int>

```

这是一种比较简单的 XML 格式。

如本节开始时所讨论的，有许多语法问题可以完全忽略。只有在需要考虑语法时，语法才会变得很重要。

55.2 WSDL

WSDL 可以完整地描述 Web 服务、可用的方法，以及调用这些方法的各种方式。此外，虽然过多地讨论 WSDL 的细节对我们并没有太多的好处，但对 WSDL 的总体理解却非常有用。

WSDL 是另一种与 XML 完全兼容的语法。WSDL 通过可用的方法、这些方法所使用的类型、通过各种协议(纯 SOAP、HTTP GET 等)发送给方法的请求消息和从方法中发送出的响应消息的格式，以及上面规范的各种绑定，指定 Web 服务。WSDL 由各种客户端读取，而不只是 .NET，还有本章前言提及的 Macromedia Flash。

WSDL 文件中最重要的部分或许是类型定义部分。这一部分使用 XML 架构描述数据交换的格式，数据交换的格式要通过可使用的 XML 元素和元素之间的关系定义。

例如，上一节中的示例所使用的 Web 服务方法：

```
int DoSomething(string stringParam, int intParam)
```

下面是为请求所做的类型声明：

```

<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl"
  ...other namespaces...>
  <wsdl:types>
    <s:schema elementFormDefault="qualified"
      targetNamespace="http://tempuri.org/">
      <s:element name="DoSomething">
        <s:complexType>

```

```

        <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="stringParam"
                type="s:string" />
            <s:element minOccurs="1" maxOccurs="1" name="intParam"
                type="s:int" />
        </s:sequence>
    </s:complexType>
</s:element>
<s:element name="DoSomethingResponse">
    <s:complexType>
        <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="DoSomethingResult"
                type="s:int" />
        </s:sequence>
    </s:complexType>
</s:element>
</s:schema>
</wsdl:types>
...other definitions...
</definitions>

```

这些类型都是以前我们看到的 SOAP 和 HTTP 请求及响应所必需的，并且这些类型被绑定在文件中的后期操作上。所有这些类型都使用标准的 XML 架构语法指定，例如：

```

<s:element name="DoSomethingResponse">
    <s:complexType>
        <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="DoSomethingResult"
                type="s:int" />
        </s:sequence>
    </s:complexType>
</s:element>

```

这个示例指定一个 DoSomethingResponse 元素，该元素包含一个子元素 DoSomething Result，这个子元素包含了一个整数。这个整数必须出现 1 次，即它必须包含在内。

如果可以访问 Web 服务的 WSDL，就可以使用 WSDL。不久我们可以看到，对 WSDL 的使用并不困难。

上面对 SOAP 和 WSDL 进行简短的讨论，接下来讨论如何创建和使用 Web 服务。

55.3 Web 服务

Web 服务的讨论分为两个方面：

- 创建 Web 服务，这一部分主要讨论如何编写 Web 服务和如何把它们放在 Web 服务器上。
- 使用 Web 服务，这一部分主要讨论如何在客户端应用程序中使用 Web 服务。

55.3.1 提供 Web 服务

把代码直接放到 .asmx 文件中或者从这些文件中引用 Web 服务类，都可以提供 Web 服务。如同 ASP.NET 页面一样，在 Visual Studio .NET 中创建 Web 服务也使用后一种方法，目的是把问题讲述

得更清楚一些。

如图 55-1 所示, 使用 File | New | Web Site 命令在 C:\ProCSharp\Chapter55 目录下创建一个 Web 服务项目 PCSWebService1, 此时系统会生成一组类似的文件, 它们与创建 Web 应用程序项目时所生成的一组文件相似, 其位置选项也相同。实际上, 唯一的区别就是创建 Web 应用程序时生成的文件是 Default.aspx, 而创建 Web 服务项目时生成的文件是 Service.aspx, 其代码隐藏是 App_Code/ Service.cs。

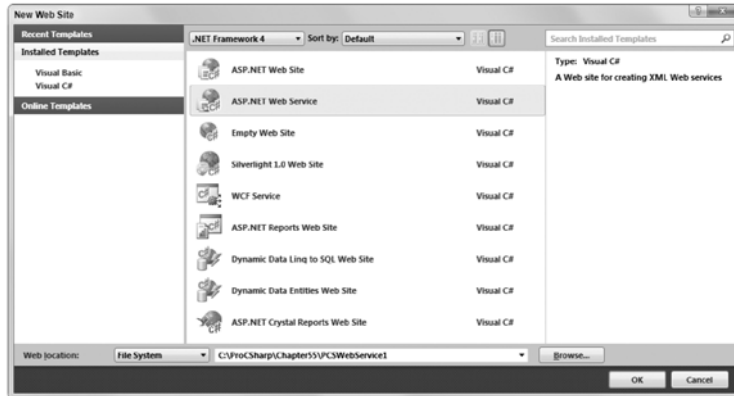


图 55-1

Service.aspx 中的代码如下所示:

```
<%@ WebService Language="C#" CodeBehind="~/App_Code/Service.cs" Class="Service" %>
```

它引用代码文件/App_Code/ Service.cs。下面的程序清单是生成的代码:



可从
wrox.com
下载源代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
// To allow this Web Service to be called from script, using ASP.NET AJAX,
// uncomment the following line.
// [System.Web.Script.Services.ScriptService]
public class Service : System.Web.Services.WebService
{
    public Service()
    {
        //Uncomment the following line if using designed components
        //InitializeComponent();
    }

    [WebMethod]
    public string HelloWorld()
    {
        return "Hello World";
    }
}
```

代码段 PCSWebService1\App_Code\Service.cs

这段代码包含几个标准名称空间引用,并定义 Web 服务类 Service(它在 Service.asmx 中引用),Service 类继承自 System.Web.Services.WebService。WebService 属性指定 Web 服务的名称空间,它允许客户端应用程序区分不同 Web 服务中同名的 Web 服务方法。WebServiceBinding 属性与 Web 服务的交互操作性相关,如 WS-I Basic Profile 1.1 规范中的定义所示。简言之,这个属性可以声明,Web 服务为一个或多个 Web 方法支持标准的 WSDL 描述,或者像这个示例一样,指定一组新的 WSDL 定义。其中还有一个注释掉的 ScriptService 属性,如果取消注释,就可以使用 ASP.NET AJAX 脚本调用 Web 方法。现在可以在这个 Web 服务类上提供其他方法。

在添加可以通过 Web 服务访问的方法时,只需要把方法定义为 public,并给方法提供 WebMethod 属性。这个属性仅把方法标记为可通过 Web 服务访问。稍后将会学习返回类型和参数使用的类型,现在用下面的方法替换自动生成的 HelloWorld()方法。

```
[WebMethod]
public string CanWeFixIt()
{
    return "Yes we can!";
}
```

现在编译该项目。

要检查是否一切正常工作,可用 Ctrl+F5 组合键运行应用程序,就会进入 Web 服务的测试页面,如图 55-2 所示。



图 55-2



注意，这个测试页面默认仅可用于本地计算机的调用者，即使 Web 服务驻留在 IIS(Internet Information Services, Internet 信息服务)中，也是如此。

在浏览器中显示的大多数文本都说明把 Web 服务的名称空间设置为 `http://tempuri.org/`。这在开发过程中不是问题，尽管以后应修改它(如 Web 页面中的文本所示)。为此可以使用 `WebService` 属性，但目前可以不修改它。

单击方法名称，可以得到 SOAP 请求和响应的信息，此外，还可以得到一个示例，说明了如何通过 HTTP GET 和 HTTP POST 方法获得请求和响应。另外，也可以单击 `Invoke` 按钮，对方法进行测试。如果方法需要简单的参数，那么在这个窗体中也可以输入这些参数(如果方法需要较复杂的参数，这个窗体就不允许以这种方式测试方法)。这样，就可以看到方法调用所返回的 XML：

```
<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://tempuri.org/"> Yes we can! </string>
```

这说明方法运行良好。

单击图 55-2 的浏览器屏幕上的 `Service Description` 链接，可以查看 Web 服务的 WSDL 描述。其中最重要的部分是关于请求和响应的元素类型的描述：

```
<wsdl:types>
  <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
    <s:element name="CanWeFixIt">
      <s:complexType />
    </s:element>
    <s:element name="CanWeFixItResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="CanWeFixItResult"
            type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
</wsdl:types>
```

该描述文件比较长，除了包含服务的各种绑定之外，还可以包含请求和响应所需类型的描述。

1. 对于 Web 服务可用的类型

Web 服务可以用于交换表 55-1 中所示的类型。

表 55-1

String	Char	Byte
Boolean	Int16	Int32
Int64	UInt16	UInt32
UInt64	Single	Double

(续表)

Guid	Decimal	DateTime
XmlQualifiedName	class	struct
XmlNode	DataSet	enum

以上所有类型的数组都可以使用，因为它们都是泛型集合类型，如 `List<string>`。还要注意，只能编组 `class` 和 `struct` 类型的公共属性和字段。

55.3.2 使用 Web 服务

既然明白了如何创建 Web 服务，接下来就讨论如何使用它们。为此需要在代码中生成一个知道如何与给定 Web 服务进行通信的代理类。这样，代码中对 Web 服务进行的任何调用都要通过这个代理类，从表面看，这个代理类就等同于 Web 服务，代码也会认为我们有了 Web 服务的本地副本。而实际的情况是有许多 HTTP 通信工作在进行，只是对我们屏蔽了其中的细节。有两种方式可以完成这项任务：第一，可以使用 `WSDL.exe` 命令行工具；第二，可以使用 Visual Studio .NET 中的 `Add Web Reference` 菜单选项。

从命令行中使用 `WSDL.exe` 时，它会根据 Web 服务的 WSDL 描述生成一个包含代理类的 .cs 文件。使用 Web 服务的 URL 来指定该文件，例如：

```
WSDL http://localhost:53679/PCSWebService1/Service.asmx?WSDL
```



注意，这里和随后的例子使用默认的文件系统驻留 Web 应用程序。为了使上面的 URL 起作用，Web 服务的 Visual Web Developer Web Server 必须正在运行。这仍不能保证 Web 服务的端口号(在这里是 61968)仍相同。这适合于演示，因为一般我们希望 Web 服务驻留在固定的 Web 服务器(如 IIS)上，否则就必须继续重新生成代理类。确保 Web 服务可用于测试的一种方式是在一个解决方案中包含多个 Web 站点。

这样就会在 `Service.cs` 文件中为上一节中的示例生成一个代理类。这个代理类将以 Web 服务命名，在这个示例中就是 `Service`，该代理类包含一些方法，那些方法将可以调用与服务同名的方法。在使用这个类时，只需把所生成的 .cs 文件添加到项目中，并在代码中使用下面的代码行：

```
Service myService = new Service();
string result = myService.CanWeFixIt();
```

默认情况下，生成的类将放在根名称空间中，因此不需要使用 `using` 语句，但是，可以使用 `WSDL.exe` 命令行选项 `/n:<namespace>` 指定一个不同的名称空间。

虽然这项技术非常有效，但是，如果服务正处于开发或处于连续更改中，就比较麻烦。当然，为了在每次编译之前自动更新所生成的代理，这项技术可以用项目的构建选项执行，但是我们有更好的方法。

在一个新的空白网站 `PCSWebClient1` 中，为上一节中的示例创建客户端(在 `C:\ProCSharp\Chapter55`

目录下), 阐明这个更好的方法。现在创建这个项目, 添加 Default.aspx 页面, 并在 Default.aspx 页面中添加下面的代码:



```
<form id="form1" runat="server">
  <div>
    <asp:Label Runat="server" ID="resultLabel" />
    <br />
    <asp:Button Runat="server" ID="triggerButton" Text="Invoke CanWeFixIt()" />
  </div>
</form>
```

代码段 PCSWebClient1\Default.aspx

接下来将把单击按钮事件处理程序绑定到 Web 服务。首先需要在项目中添加对 Web 服务的引用。其方法是: 右击 Solution Explorer 窗口中的新客户端项目, 选择 Add Web Reference 选项。然后, 在出现的窗口中输入 Web 服务文件 Service1.asmx 的 URL, 或者使用本地计算机链接中的 Web 服务, 自动查找它, 如图 55-3 所示。

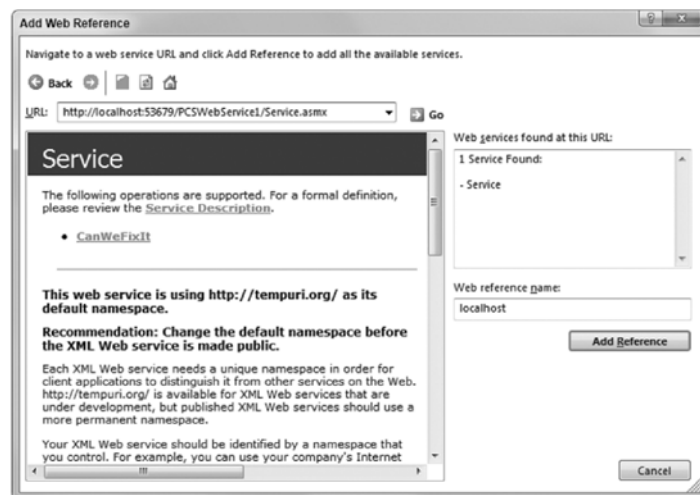


图 55-3

接着, 可以使用 Add Reference 按钮添加引用。但应先把 Web 引用的默认内容从 localhost 改为 myWebService。现在单击 Add Reference 按钮, 在 Solution Explorer 窗口中把 myWebService 添加到该项目的 App-WebReferences 文件夹中。如果在 Solution Explorer 窗口中查看这个文件夹时, 就可以看到 Service.disco、Service.discomap 和 Service1.wsdl 文件已添加到项目中。

Web 引用名(myWebService)也是使用代理类需要引用的名称空间。在 Default.aspx.cs 的代码中添加下面的 using 语句:

```
using myWebService;
```

现在就可以在类中使用服务, 而不必完全限定它的名称。

在设计视图中双击该按钮, 添加下面的代码, 把事件处理程序添加到窗体上的按钮中:

```
protected void triggerButton_Click(object sender, EventArgs e)
{
```

```

        Service myService = new Service();
        resultLabel.Text = myService.CanWeFixIt();
    }

```

运行应用程序并且单击该按钮，浏览器窗口将显示 CanWeFixIt()方法的执行结果。



如果使用 ASP.NET Development Server(即 Web 应用程序驻留在本地文件系统上，而没有驻留在 IIS 上)，就会产生 401:Unauthorized 错误。这是因为，默认情况下，这个服务器配置为需要 NTLM 身份验证。为了修正这个错误，可以在 PCSWebService1 的属性页面的 Start Options 页面上取消选择 NTLM Authentication 复选框，或者在调用 Web 服务方法时传递默认的证书。后一种方式需要以下代码：

```
myService.Credentials = System.Net.CredentialCache.DefaultCredentials;
```

以后这个 Web 服务也许会更改，但是，使用这个方法，可以只右击 Server Explorer 窗口中的 App_WebReference 文件夹，并选择 Update Web/Service Reference 命令，生成一个新的代理类，以供使用。

另外，Web 服务以后可能在部署它会移动。如果查看客户端应用程序的 web.config 中，就会看到如下代码：



可从
wrox.com
下载源代码

```

<appSettings>
  <add key="myWebService.Service"
    value="http://localhost:53679/PCSWebService1/Service.asmx"/>
</appSettings>

```

代码段 PCSWebClient1\web.config

由于这个设置配置把 Web 请求发送到什么地方，因此必须确保它匹配 Web 服务的位置，或者使用其他方式记录这个信息。

55.4 扩充事件登记示例

既然我们已经对 Web 服务的创建和使用有了一定的了解，接下来要应用这些知识扩充第 40 章中的会议室登记应用程序。具体来说，就是从应用程序中提取数据库访问方面的内容，把它们放到一个 Web 服务中。这个 Web 服务有两个方法：

- GetData(), 它返回一个 DataSet 对象，DataSet 对象包含 PCSDemoSite 数据库中的 3 个表。
- AddEvent(), 因为它添加一个事件，返回受影响的行数，所以客户端应用程序可以检查数据是否发生变化。

此外，使用能够减少加载量的技术设计 Web 服务。具体来说，就是把包含会议室登记数据的 DataSet 对象存储在 Web 服务应用程序中的应用层上。这意味着对数据的多个请求将不需要进行额外的数据库请求工作。这样，只有在数据库中添加新的数据时，才会刷新应用层 DataSet 中的数据。通过其他方式对数据库的更改(如手动编辑)就不会反映在 DataSet 对象中。而且，只有知道 Web 服务是唯一可以直接访问数据的应用程序，这样我们就没有什么可担心的。

55.4.1 事件登记 Web 服务

在 Visual Studio 中, 在 C:\ProCSharp\Chapter55 目录下创建一个名为 PCSWebService2 的新 Web 服务项目。首先, 把第 41 章代码的 PCSDemoSite 中的数据库文件(MeetingRoomBooker.mdf)复制到 Web 服务的 App_Data 目录下。接着, 需要给项目添加 Global.asax 文件, 然后在它的 Application_Start() 处理程序中更改代码。把 MeetingRoomBooker 数据库中的所有数据都加载到数据集中并存储它。其中涉及的大部分代码前面已讨论过, 因为前面已经把数据库加载到 DataSet 中。如本章前面所示, 也可以使用存储在 web.config 文件中的连接字符串。web.config 文件的代码如下所示(连接字符串应放在单独一行上):



```
<?xml version="1.0" ?>
<configuration>
  ...
  <connectionStrings>
    <add name="MRBConnectionString"
          connectionString="Data Source=.\SQLEXPRESS;Integrated
          Security=True;AttachDBFilename=|DataDirectory|MeetingRoomBooker.mdf;
          User Instance=True"
          providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

代码段 PCSWebService2\web.config

在 Global.asax 文件的 Application_Start() 处理程序中, 添加如下代码:



```
void Application_Start(Object sender, EventArgs e)
{
    System.Data.DataSet ds;
    System.Data.SqlClient.SqlConnection sqlConnection1;
    System.Data.SqlClient.SqlDataAdapter daAttendees;
    System.Data.SqlClient.SqlDataAdapter daRooms;
    System.Data.SqlClient.SqlDataAdapter daEvents;

    using (sqlConnection1 = new System.Data.SqlClient.SqlConnection())
    {
        sqlConnection1.ConnectionString =
            ConfigurationManager.ConnectionStrings["MRBConnectionString"]
                .ConnectionString;

        sqlConnection1.Open();

        ds = new System.Data.DataSet();
        daAttendees = new System.Data.SqlClient.SqlDataAdapter(
            "SELECT * FROM Attendees", sqlConnection1);
        daRooms = new System.Data.SqlClient.SqlDataAdapter(
            "SELECT * FROM Rooms", sqlConnection1);
        daEvents = new System.Data.SqlClient.SqlDataAdapter(
            "SELECT * FROM Events", sqlConnection1);

        daAttendees.Fill(ds, "Attendees");
        daRooms.Fill(ds, "Rooms");
        daEvents.Fill(ds, "Events");
    }
}
```

```

        Application["ds"] = ds;
    }

```

代码段 PCSWebService2\Global.asax

这里需要注意的最重要的代码在最后一行中。通常, `Application`(类似 `Session`)对象都有一个名/值对的集合, 可以在该集合中存储数据。这里在 `Application` 存储器中创建一个名称(ds), 它从数据库中提取 ds 的序列化值, 其中 ds 包含 `Attendees`、`Rooms` 和 `Events` 表。这样, Web 服务对象的所有实例在任何时间都可以访问这些值。

这项技术非常适合用于只读数据, 因为多个线程可以访问它, 减少了在数据库上的负载。但要注意, 由于 `Events` 表有可能发生变化, 因此在 `Events` 表发生变化时, 必须更新应用层的 `DataSet` 类。稍后会介绍这一内容。

接下来用一个新服务(`MRBService`)替换默认的 `Service` 服务。为此, 删除已有的 `Service.asmx` 和 `Service.cs` 文件, 在项目中添加一个新的 Web 服务, 命名为 `MRBService`(确保选择把代码放在一个单独的文件中)。接着把 `GetData()`方法添加到 `MRBService.cs` 的服务中:



```

[WebMethod]
public DataSet GetData()
{
    return (DataSet)Application["ds"];
}

```

代码段 PCSWebService2\App_Code\MRBService.cs

上面代码使用与 `Application_Load()`方法相同的语法访问存储的 `dataset`, 这样, 就可以仅将数据强制转换为正确的类型, 并返回。

注意, 为了使上述代码正常工作, 并便于以后添加的其他 Web 方法正常工作, 可以添加下面的 `using` 语句:

```

using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Services;

```

`AddEvent()`方法稍微有点复杂。从概念上讲, 需要做下面的事情:

- 接受来自客户端的事件数据
- 使用这些数据创建 SQL INSERT 命令
- 连接到数据库并且执行 SQL 语句
- 如果添加成功, 就需要刷新 `Application["ds"]`中的数据
- 把成功或失败的通知返回给客户端(如果有必要, 客户端就会刷新它的 `DataSet`)

从现在开始, 就要接受正确数据类型的所有字段:

```

[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                    string eventAttendees, DateTime eventDate)
{

```

```
    ...
}
```

下面声明数据库访问需要的对象，连接到数据库并执行查询，完成这些工作使用的所有代码与 PCSDemoSite 中的代码相似(记住，这里也需要连接字符串，从 web.config 中提取它)：

```
[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                   string eventAttendees, DateTime eventDate)
{
    System.Data.SqlClient.SqlConnection sqlConnection1;
    System.Data.SqlClient.SqlDataAdapter daEvents;
    DataSet ds;

    using (sqlConnection1 = new System.Data.SqlClient.SqlConnection())
    {
        sqlConnection1.ConnectionString =
            ConfigurationManager.ConnectionStrings["MRBConnectionString"]
                .ConnectionString;

        System.Data.SqlClient.SqlCommand insertCommand =
            new System.Data.SqlClient.SqlCommand(
                "INSERT INTO [Events] (Name, Room, "
                + "AttendeeList, EventDate) VALUES (@Name, @Room, @AttendeeList, "
                + "@EventDate)", sqlConnection1);
        insertCommand.Parameters.Add("Name", SqlDbType.VarChar, 255).Value
            = eventName;
        insertCommand.Parameters.Add("Room", SqlDbType.Int, 4).Value
            = eventRoom;
        insertCommand.Parameters.Add("AttendeeList", SqlDbType.Text, 16).Value
            = eventAttendees;
        insertCommand.Parameters.Add("EventDate", SqlDbType.DateTime, 8).Value
            = eventDate;

        sqlConnection1.Open();
        int queryResult = insertCommand.ExecuteNonQuery();
    }
}
```

同前，使用 queryResult 存储受查询影响的行数。如果 queryResult 的结果是 1，则说明查询成功，然后就可以对数据库进行新的查询，以便刷新 DataSet 中的 Events 表。在执行更新时，必须锁定应用程序数据，确保在更新过程中其他线程不可以访问 Application["ds"]。使用 Application 对象的 Lock() 和 Unlock() 方法，可以实现对数据的锁定和解锁：

```
[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                   string eventAttendees, DateTime eventDate)
{
    ...
    int queryResult = insertCommand.ExecuteNonQuery();
    if (queryResult == 1)
    {
        daEvents = new System.Data.SqlClient.SqlDataAdapter(
            "SELECT * FROM Events", sqlConnection1);
```

```

        ds = (DataSet)Application["ds"];
        ds.Tables["Events"].Clear();
        daEvents.Fill(ds, "Events");
        Application.Lock();
        Application["ds"] = ds;
        Application.Unlock();
    }
}
}

```

最后，返回 `queryResult`，以便让客户端知道查询是否成功：

```

[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                  string eventAttendees, DateTime eventDate)
{
    ...
    return queryResult;
}

```

至此，就完成了 Web 服务。同前，在 Web 浏览器中查看 `.asmx` 文件，可以测试 Web 服务，这样，不用编写任何客户端代码，就可以添加记录，并查看 `GetData()` 方法返回的 `DataSet` 的 XML 表示。

在继续之前，需要讨论 `DataSet` 对象和 Web 服务的组合使用。初看起来这似乎是交换数据的一种荒谬方式，而实际上这是一种极其有用的技术。然而，`DataSet` 类的用途非常广泛，该事实有隐含意义。如果查看为 `GetData()` 方法生成的 WSDL，就会看到如下代码：

```

<s:element name="GetDataResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="GetDataResult">
        <s:complexType>
          <s:sequence>
            <s:element ref="s:schema" />
            <s:any />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:sequence>
  </s:complexType>
</s:element>

```

可以看出，这是非常通用的代码，允许传递的 `DataSet` 对象包含用内联架构指定的任何数据。但是，这表示 WSDL 没有完整地描述 Web 服务。对于 .NET 客户端这不是个问题，在前面的示例中传递简单的字符串时，一切都很正常，唯一的区别是我们交换了一个 `DataSet` 对象。但是，非 .NET 客户端必须提前了解要传递的数据，或者某个等价的 `DataSet` 类，才能访问数据。这包括脚本客户端，例如，使用客户端 ASP.NET AJAX 代码处理已检索数据的客户端。

这个问题的解决方法是把数据重新打包为另一种格式，如结构数组。如果这么做，就可以以任意方式定制生成的 XML，XML 可以由 Web 服务的架构完整地描述。这还可以影响性能，因为传递 `DataSet` 对象会生成大量 XML，在大多数情况下远远超过了需要的 XML。重新打包数据导致的系统开销比在 Web 上发送数据的系统开销少得多，而且数据较少，序列化和反序列化也较快。所以如果

性能比较重要，就应避免以这种方式使用 DataSet 对象，除非要使用 DataSet 对象提供的其他功能。

但是，这里使用 DataSet 对象不成问题，而且还可以大大简化其他代码。

55.4.2 事件登记客户端

本节使用的客户端是第 41 章开发的 PCSDemoSite 网站。下面把这个应用程序命名为 PCSDemoSite2，放在 C:\ProCSharp\Chapter55 目录下，并且使用 PCSDemoSite 中的代码作为起点。

对项目主要进行两个主要的修改：第一，从应用程序中删除对数据库直接访问的所有内容，并使用 Web 服务代替；第二，引入一个从 Web 服务返回的 DataSet 的应用层存储器，并且只有在必要时才对 DataSet 进行更新，这将大大降低数据库上的负载。

对于新的 Web 应用程序，首先需要添加对 PCSWebService2/MRBSservice.asmx 服务的 Web 引用。添加的方法在本章前面部分中已经讨论过，即在 Server Explorer 窗口中右击项目，定位.asmx 文件，调用 Web 引用 MRBService，单击 Add Reference 按钮。在此之前，可能还需要在浏览器中查看 Web 服务的.asmx 文件，以启动 ASP.NET Development Server。因为我们不再使用本地数据库，所以还可以从 App_Data 目录中删除它，从 web.config 中删除 MRBConnectionString 项。其余修改操作都在 MRB.ascx 和 MRB.ascx.cs 中进行。

开始时，删除 MRB.ascx 上的所有数据源，并删除当前所有数据绑定控件上的 DataSourceID 项。这是因为我们要在代码隐藏文件中处理数据绑定。



注意，在修改或删除 Web 服务器控件的 DataSourceID 属性时，需要确认是否要删除已定义的模板，因为不能保证控件使用的数据对于这些模板仍旧有效。因为这里要使用相同的数据，只是从另一个数据源中提取，所以应保留模板。如果确定删除了模板，最终的 HTML 布局就会变成默认布局，它看起来不漂亮，于是必须从头添加它们或重写它们。

接着，需要给 MRB.ascx.cs 添加一个属性，用于存储 Web 服务返回的 DataSet。这个属性使用 Application 状态存储器，其方式与 Web 服务中的 Global.asax 相同。代码如下：



```
public DataSet MRBData
{
    get
    {
        if (Application["mrbData"] == null)
        {
            Application.Lock();
            MRBService.MRBService service = new MRBService.MRBService();
            service.Credentials = System.Net.CredentialCache.DefaultCredentials;
            Application["mrbData"] = service.GetData();
            Application.Unlock();
        }
        return Application["mrbData"] as DataSet;
    }
    set
    {
        Application.Lock();
```

```

        if (value == null && Application["mrbData"] != null)
        {
            Application.Remove("mrbData");
        }
        else
        {
            Application["mrbData"] = value;
        }
        Application.Unlock();
    }
}

```

代码段 PCSDemoSite2\MRB\MRB.ascx.cs

注意需要锁定和解锁 `Application` 状态，这与 `Web` 服务中相同。另外，注意只有在需要时才填充 `Application["mrbData"]` 存储器，即在它为空时填充它。这个 `DataSet` 对象现在可用于 `PCSDemoSite2` 的所有实例，也就是说多个用户可以读取数据，而无需调用 `Web` 服务或访问数据库。这里还设置了证书，如前所述，在使用 `ASP.NET Development Server` 中的 `Web` 服务时需要证书。如果不需要它，就可以注释掉这行代码。

要绑定 `Web` 页面上的控件，可以提供 `DataView` 属性，它映射到存储在这个属性中的数据，如下所示：

```

private DataView EventData
{
    get
    {
        return MRBData.Tables["Events"].DefaultView;
    }
}

private DataView RoomData
{
    get
    {
        return MRBData.Tables["Rooms"].DefaultView;
    }
}

private DataView AttendeeData
{
    get
    {
        return MRBData.Tables["Attendees"].DefaultView;
    }
}

private DataView EventDetailData
{
    get
    {
        if (EventList != null && EventList.SelectedValue != null)
        {
            return new DataView(MRBData.Tables["Events"], "ID=" +
                EventList.SelectedValue.ToString(), "",

```

```

        DataRowState.CurrentRows);
    }
    else
    {
        return null;
    }
}
}

```

还可以删除已有的 `eventData` 字段和 `EventData` 属性。

大多数属性都很简单，只有最后一个属性有点新意。在本例中，要筛选 `Events` 表中的数据，只获得一个事件，用于在细节视图 `FormView` 控件中显示。

既然没有使用数据源控件，就必须自己绑定数据。为此，可调用页面的 `DataBind()` 方法，但仍需要为页面上的各种数据绑定控件设置数据源 `DataSource` 属性。一种较好的方式是在 `OnDataBinding()` 事件处理程序的重写版本中设置它，如下所示：

```

protected override void OnDataBinding(EventArgs e)
{
    roomList.DataSource = RoomData;
    attendeeList.DataSource = AttendeeData;
    EventList.DataSource = EventData;
    FormView1.DataSource = EventDetailData;
    base.OnDataBinding(e);
}

```

这里把 `roomList`、`attendeeList`、`EventList` 和 `FormView1` 的 `DataSource` 属性设置为前面定义的属性。接着给 `Page_Load()` 方法添加 `DataBind()` 调用：

```

void Page_Load(object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        nameBox.Text = Context.User.Identity.Name;
        DateTime trialDate = DateTime.Now;
        calendar.SelectedDate = GetFreeDate(trialDate);
        DataBind();
    }
}

```

此外，还需要更改 `submitButton_Click()` 方法，以使用 Web 服务的 `AddData()` 方法。同样，其中的大部分代码保持不变，只有数据添加代码需要更改：

```

void submitButton_Click(object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        ...
        try
        {
            MRBService.MRBService service = new MRBService.MRBService();
            if (service.AddEvent(eventBox.Text, int.Parse(roomList.SelectedValue),
                                attendees, calendar.SelectedDate) == 1)

```

```

        {
            MRBData = null;
            DataBind();
            calendar.SelectedDate =
                GetFreeDate(calendar.SelectedDate.AddDays(1));
        }
    }
    catch
    {
    }
}
}

```

事实上，这里所做的全部工作是为了简化事情，因此，当使用设计良好的 Web 服务时，通常可以忽略许多内部工作机制，而应更关注用户体验。

这段代码没有很多注释。继续使用 `queryResult` 很有帮助，锁定应用程序也是必需的，这些内容在前面曾经提到过。

最后要修改的地方有两处，一处是 `EventList_SelectedIndexChanged()`方法：

```

protected void EventList_SelectedIndexChanged(object sender, EventArgs e)
{
    FormView1.DataSource = EventDetailData;
    EventList.DataSource = EventData;
    EventList.DataBind();
    FormView1.DataBind();
}

```

这就确保事件列表的数据源和细节视图正确地刷新。

还需要添加 `EventList_SelectedIndexChanged()`处理程序，它必须关联到 `EventList` 控件的 `SelectedIndexChanging` 事件上：

```

protected void EventList_SelectedIndexChanged(object sender,
    ListViewSelectEventArgs e)
{
    EventList.SelectedIndex = e.NewSelectedIndex;
}

```

不添加这个处理程序，代码就会失败，因为我们进行的修改需要这个事件的处理程序。

虽然 `PCSDemoSite2` 网站从外表和功能上来看与 `PCSDemoSite` 非常相似，但是 `PCSDemoSite2` 执行起来会更好一些。很容易把同一 Web 服务用到其他应用程序中，例如，如果添加更多方法，就可以在页面上显示事件，甚至可以编辑事件、与会者姓名和房间等。这样做并不会破坏 `PCSDemoSite2`，它仅忽略任何新创建的方法。但必须引入新的触发机制，以更新缓存在事件列表中的数据，因为在其他地方修改这些数据会使数据过时。

55.5 使用 SOAP 标题交换数据

本章要讨论的最后一个主题是使用 SOAP 标题交换信息，而不包括方法参数中的信息。讨论这个主题的原因是：这是一个用于维护用户登录的非常好的系统。我们将不详细论述如何设置服务器，

用于 SSL 连接，也不讨论可以使用 IIS 配置的各种身份验证方法，因为这些都不影响交换信息所需的 Web 服务代码。

假定有一个服务包含一个简单的身份验证方法，其签名如下所示：

```
AuthenticationToken AuthenticateUser(string userName, string password);
```

其中 `AuthenticationToken` 是我们定义的一个类型，它可以在后续方法调用中由用户使用，例如：

```
void DoSomething(AuthenticationToken token, OtherParamType param);
```

一旦用户登录，他就可以使用从 `AuthenticateUser()` 方法中接收到的令牌(token)访问其他方法。尽管常常用较复杂的方式实现它，但这种技术是典型的安全 Web 系统。

使用 SOAP 标题交换标识(或任何其他数据)可以进一步简化这个过程。由于可以对方法进行限制，因此只有在方法调用中包含指定的 SOAP 标题，才调用它们。这就简化了它们的结构，如下所示：

```
void DoSomething(OtherParamType param);
```

其优点是，一旦在客户端上设置了标题，它就一直存在。在设置初始位后，就可以在后续的所有 Web 方法调用中忽略身份验证标识。

为了查看这个过程，在 `C:\ProCSharp\Chapter55\` 目录下创建一个新的 Web 服务项目，命名为 `PCSWebService3`，在 `App_Code` 目录下添加一个新类 `AuthenticationToken`：



```
using System;
using System.Web.Services.Protocols;

public class AuthenticationToken : SoapHeader
{
    public Guid InnerToken;
}
```

代码段 PCSWebService3\App_Code\AuthenticationToken.cs

通常使用 GUID 来标识该令牌，因为这样可以确保它是唯一的。

要声明 Web 服务可以有一个自定义 SOAP 标题，只需给服务类添加一个新类型的公共成员：



```
public class Service : System.Web.Services.WebService
{
    public AuthenticationToken AuthenticationTokenHeader;
```

代码段 PCSWebService3\App_Code\Service.cs

还需要使用 `System.Web.Services.Protocols.SoapHeaderAttribute` 属性，标记那些需要额外 SOAP 标题才能正常工作的 Web 方法。在添加这样的方法前，应先添加一个非常简单的 `Login()` 方法，客户端可以使用该方法获得身份验证令牌：

```
[WebMethod(true)]
public Guid Login(string userName, string password)
{
    if ((userName == "Karli") && (password == "Cheese"))
    {
        Guid currentUser = Guid.NewGuid();
        Session["currentUser"] = currentUser;
```

```

        return currentUser;
    }
    else
    {
        Session["currentUser"] = null;
        return Guid.Empty;
    }
}

```

如果使用正确的用户名和密码,就会生成一个新的 Guid 对象,它存储在会话层的变量中,并返回给用户。如果身份验证失败,就返回一个空的 Guid 实例,该 Guid 存储在会话层中。True 参数允许会话状态用于这个 Web 方法,因为在 Web 服务中,默认情况下禁用它,而这个功能需要它。

接着是一个接受标题的方法,该方法用 SoapHeaderAttribute 属性指定:

```

[WebMethod(true)]
[SoapHeaderAttribute("AuthenticationTokenHeader",
    Direction = SoapHeaderDirection.In)]
public string DoSomething()
{
    if (Session["currentUser"] != null &&
        AuthenticationTokenHeader != null &&
        AuthenticationTokenHeader.InnerToken
            == (Guid)Session["currentUser"])
    {
        return "Authentication OK.";
    }
    else
    {
        return "Authentication failed.";
    }
}

```

这将根据 AuthenticationTokenHeader 标题是否存在,是否不是空的 Guid,以及是否匹配存储在 Session["currentUser"]中的一个字符串(此时存在 Session 变量),返回两个字符串中的一个。

接着创建一个简单的客户端,测试该服务。创建一个新的 PCSWebClient2 网站,给用户界面使用包含如下简单代码的 Default.aspx 页面:



```

<form id="form1" runat="server">
    <div>
        User Name:
        <asp:TextBox Runat="server" ID="userNameBox" /><br />
        Password:
        <asp:TextBox Runat="server" ID="passwordBox" /><br />
        <asp:Button Runat="server" ID="loginButton" Text="Log in" /><br />
        <asp:Label Runat="server" ID="tokenLabel" /><br />
        <asp:Button Runat="server" ID="invokeButton"
            Text="Invoke DoSomething()" /><br />
        <asp:Label Runat="server" ID="resultLabel" /><br />
    </div>
</form>

```

代码段 PCSWebClient2\Default.aspx

把 PCSWebService3 服务添加为一个 Web 引用(因为 Web 服务相对于解决方案位于本地, 所以可以单击 This Solution 链接中的 Web Services, 快速获得引用), Web 服务的名称为 authenticateService, 再把下面的 using 语句添加到 Default.aspx.cs 中:



```
using System.Net;
using authenticateService;
```

代码段 PCSWebClient2\Default.aspx.cs

需要使用 System.Net 名称空间, 因为它包含 CookieContainer 类。这个类用于存储 cookie 的引用, 如果正在运行使用会话状态的 Web 服务, 就需要它。这是因为 Web 服务需要跨客户端的多个调用, 以某种方式检索正确的会话状态, 其中在每次回发时重新创建服务的代理。检索 Web 服务使用的 cookie, 以存储会话状态, 在 Web 服务调用之间存储它, 然后在以后的调用中使用它, 就可以在 Web 服务中维护正确的会话状态。如果不这么做, Web 服务就会丢失其会话状态, 此时需要客户提供登录信息。

在代码中, 将使用一个受保护的成员存储 Web 引用代理, 用另一个受保护的成员存储一个布尔值, 表示用户是否通过了身份验证:

```
public partial class _Default : System.Web.UI.Page
{
    protected Service myService;
    protected bool authenticated;
```

Page_Load()方法首先初始化 myService 服务, 并准备一个 CookieContainer 实例, 由该服务使用:

```
protected void Page_Load(object sender, EventArgs e)
{
    myService = new Service();
    myService.Credentials = CredentialCache.DefaultCredentials;
    CookieContainer serviceCookie;
```

接着检查已存储的 CookieContainer 实例或创建一个新实例。在这两种方式中, 都要把 CookieContainer 赋予 Web 服务代理, 以准备在调用后检索 Web 服务的 cookie 信息。这里使用的存储器是窗体的 ViewState 集合(这是在回发之间持久化信息的一种有效方式, 其工作方式类似于在应用程序或会话层中存储信息):

```
if (ViewState["serviceCookie"] == null)
{
    serviceCookie = new CookieContainer();
}
else
{
    serviceCookie = (CookieContainer)ViewState["serviceCookie"];
}
myService.CookieContainer = serviceCookie;
```

然后 Page_Load()方法确定是否有一个已存储的标题, 并相应地把该标题赋予代理(以这种方式赋予标题是把要发送的数据作为 SOAP 标题必要的唯一一步)。这样, 调用的任何事件处理程序(例如, Web 方法调用按钮的事件处理程序)就不需要担心赋予标题的问题了——这一步已经完成了:

```

        AuthenticationToken header = new AuthenticationToken();
        if (ViewState["AuthenticationTokenHeader"] != null)
        {
            header.InnerToken = (Guid)ViewState["AuthenticationTokenHeader"];
        }
        else
        {
            header.InnerToken = Guid.Empty;
        }
        myService.AuthenticationTokenValue = header;
    }

```

接着在设计器中双击 Login 按钮，为它添加一个事件处理程序：

```

protected void loginButton_Click(object sender, EventArgs e)
{
    Guid authenticationTokenHeader = myService.Login(userNameBox.Text,
                                                    passwordBox.Text);
    tokenLabel.Text = authenticationTokenHeader.ToString();
    if (ViewState["AuthenticationTokenHeader"] != null)
    {
        ViewState.Remove("AuthenticationTokenHeader");
    }
    ViewState.Add("AuthenticationTokenHeader", authenticationTokenHeader);
    if (ViewState["serviceCookie"] != null)
    {
        ViewState.Remove("serviceCookie");
    }
    ViewState.Add("serviceCookie", myService.CookieContainer);
}

```

这个处理程序使用在两个文本框中输入的任何数据调用 Login()方法，显示返回的 Guid，并把该 Guid 存储在 ViewState 集合中。它还更新已存储在 ViewState 集合中的 CookieContainer 存储器，以备重用。

最后，必须以相同的方式为 Invoke DoSomething()按钮添加一个处理程序：

```

protected void invokeButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = myService.DoSomething();
    if (ViewState["serviceCookie"] != null)
    {
        ViewState.Remove("serviceCookie");
    }
    ViewState.Add("serviceCookie", myService.CookieContainer);
}

```

这个处理程序仅输出 DoSomething()方法返回的文本，并更新 CookieContainer 存储器，这与 loginButton_Click()方法相同。

在运行这个应用程序时，可以直接单击 Invoke DoSomething()按钮，因为 Page_Load()方法已经指定一个备用的标题(如果没有指定标题，就会抛出一个异常，因为已经指定这个方法必须有一个标题)。这会使 DoSomething()方法返回一条失败消息，如图 55-4 所示。

如果试着用除 Karli 和 Cheese 之外的任何用户名和密码登录, 就会得到相同的结果。另一方面, 如果使用这些证书进行登录, 接着调用 DoSomething() 方法, 就会获得成功的消息, 如图 55-5 所示。

此外, 还可以看到 Guid 用于验证有效性的字符串表示。

当然, 使用这种技术通过 SOAP 标题交换数据的应用程序可能会复杂得多。它们需要以比使用会话存储更灵活的方式存储登录令牌, 或许存储在数据库中。为了完整起见, 也可以在经过某段时间时给这些令牌执行自己的过期机制, 给用户提提供注销选项, 即删除令牌。尽管会话状态在指定的时间(默认为 20 分钟)过后过期, 但还可以使用更复杂、更强大的机制。甚至可以根据用户使用的 IP 地址验证令牌, 获得更高的安全性。但此处的关键点是用户的用户名和密码只发送一次, 然后使用 SOAP 标题简化以后的方法调用。

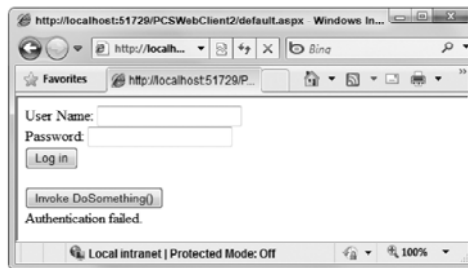


图 55-4

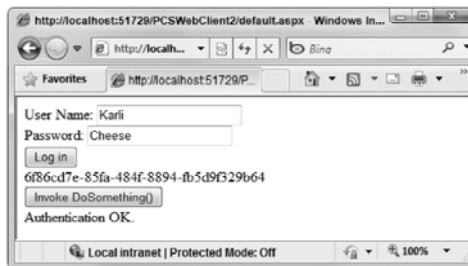


图 55-5

55.6 小结

本章讨论了如何应用 C# 和 Visual Studio .NET 开发平台创建和使用 Web 服务。虽然它们易学易用, 但是用途却非常大。

目前已经有许多 Web 服务可以从任何平台上访问。这是因为使用了 SOAP 协议, 由于 SOAP 协议不限制使用 .NET, 因此从任何平台上都可以访问 Web 服务。

本章开发的主要示例阐明了如何轻松地创建 .NET 分布式应用程序。这里假定用户使用一个服务器测试示例, 但是这并不是说 Web 服务不应与客户端完全分离。如果需要额外的数据层, 则它可以位于数据库中一个单独的服务器上。

对于大型应用程序, 数据缓存技术是另一项非常重要的技术, 因为可能会出现数千个用户同时连接的情况。

从最后一个示例可看出, 通过 SOAP 标题交换数据是另一种可以用在应用程序中的有效技术。这个示例交换的是注册令牌, 更复杂的数据也可以通过这种方式交换。例如, 它可以用于 Web 服务的简单“密码保护”, 而无需借助于施加更复杂的安全性。

Web 服务的使用者并不一定是 Web 应用程序, 因为也可以从 Windows 窗体或 WPF 应用程序(在公司的内联网中使用这些应用程序当然看起来更好一些)中使用 Web 服务。另外, 在 ASP.NET 中, Web 服务可以使用简单的事件驱动系统, 异步调用其他 Web 服务。这非常适合于 Windows 窗体应用程序, 它意味着, 在复杂的 Web 服务执行复杂的工作, 应用程序仍可以保持响应。

最后, 值得一提的是 Web 服务是最新 WCF 技术(参见第 43 章)中的一个被分离出来的子集。但这并不意味着应总是使用 WCF 服务, 而不使用 Web 服务——Web 服务实现起来要简单得多, 而简单是一个巨大的优势。

第 56 章

LINQ to SQL

本章内容:

- 使用 LINQ to SQL 和 Visual Studio 2010
- 把 LINQ to SQL 对象映射到数据库实体上
- 脱离 O/R 设计器构建 LINQ to SQL 操作
- 使用 O/R 设计器和自定义对象
- 用 LINQ 查询 SQL Server 数据库
- 存储过程和 LINQ to SQL

在 C# 2010 中, 令人激动的新增功能之一是.NET LINQ(Language Integrated Query, 语言集成查询架构)架构。LINQ 主要在编程数据集成的基础上提供了一种轻型外观。这是一个很好的功能, 因为数据是最重要的。

每个应用程序都以某种方式处理数据, 无论这些数据来自内存(内存中的数据)、数据库、XML 文件、文本文件还是其他地方。许多开发人员发现很难把 C#中强类型化的、面向对象的数据移动到数据层中(其中对象是辅助类的成员)。在最好的情况下, 从一个环境转换到另一个环境充满了易于出错的操作, 且顶多是一个重组过程。

在 C#中, 用对象编程意味着利用一个强类型化的强大功能处理代码。很容易在名称空间中导航, 使用 Visual Studio IDE 中的调试器等。但是, 在访问数据时, 情况就大不相同了。

这是一个没有强类型化的环境, 调试很痛苦, 甚至是不可能的, 大部分时间都花在把字符串作为命令发送给数据库。开发人员还必须注意基础数据, 了解它的构造方式或者所有数据点的相互关系。

LINQ 为基础数据存储器提供了一个强类型化的界面。LINQ 为开发人员提供了在编码环境下工作的方式, 可以把基础数据作为对象访问, 使用 IDE、IntelliSense 甚至调试功能。

有了 LINQ, 我们创建的查询现在就变成.NET Framework 和其他环境中的重要成员。在对数据存储器执行查询时, 会很快发现它们现在正常工作且行为方式类似于系统中的类型。这说明, 现在可以使用任意兼容.NET 的语言并查询基础数据存储器, 这在以前是不可能的。



第 11 章概述了 LINQ。

图 56-1 显示了 LINQ 在查询数据中的作用。

在图 56-1 中，根据要在应用程序中处理的基础数据的不同，有不同类型的 LINQ 功能。下面列出了各种 LINQ 技术：

- LINQ to Objects
- LINQ to DataSets
- LINQ to SQL
- LINQ to Entities
- LINQ to XML

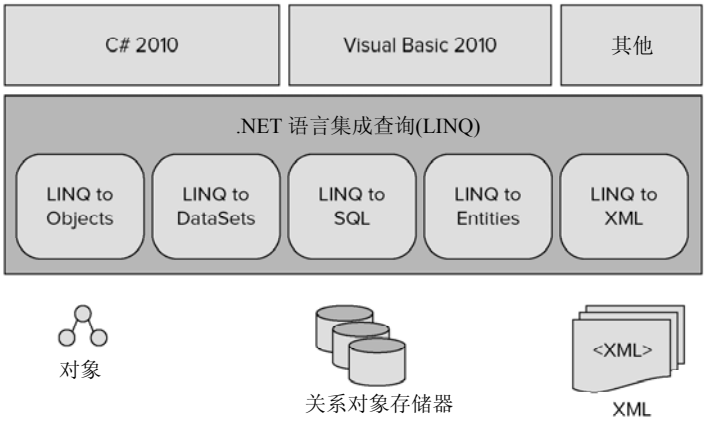


图 56-1

对于开发人员，类库提供了可以使用 LINQ 查询的对象，像查询任何其他数据存储器那样。其实，对象只不过是存储在内存中的数据。实际上，对象本身可能就在查询数据。此时应使用 LINQ to Objects。

LINQ to SQL(本章的重点)、LINQ to Entities 和 LINQ to DataSets 都提供了查询关系数据的方式。使用 LINQ 可以直接查询数据库，甚至查询数据库提供的存储过程。图 56-1 中的最后一项是使用 LINQ to XML 查询 XML(详见第 33 章)。LINQ 非常强大的原因是它对所查询的内容没有任何限制，因为查询非常类似。

56.1 LINQ to SQL 和 Visual Studio 2010

特殊的是 LINQ to SQL 是在 SQL Server 数据库上设置一个强类型化界面的方式。LINQ to SQL 提供的方法是当前可用于查询 SQL Server 最简单的方法。它不仅在数据库中查询单个表。例如，如果调用 Northwind 数据库的 Customers 表，提取该数据库的 Orders 表中某位顾客的特定订单，LINQ 就使用两个表之间的关系并代表用户进行查询。LINQ 会查询数据库，加载要在代码中使用的数据(也是强类型化的)。

注意，LINQ to SQL 不仅可以查询数据，还可以执行需要的 Insert/Update/Delete 语句。

也可以与整个过程交互操作，并定制所执行的操作，给 CRUD(Create/Read/Update/Delete)操作添加自己的业务逻辑。

Visual Studio 2010 非常重视 LINQ to SQL，因为这是一个功能丰富的用户界面，允许设计要使用的 LINQ to SQL 类。

下一节介绍如何创建第一个 LINQ to SQL 实例，从 Northwind 数据库的 Products 表中提取数据项。

56.1.1 调用 Products 表

作为使用 LINQ to SQL 的一个例子，本章首先调用 Northwind 数据库中的单个表，用这个表把一些结果显示在屏幕上。

首先创建一个控制台应用程序(使用 .NET Framework 4)，把 Northwind 数据库文件添加到这个项目中(Northwind.MDF)。



下面的例子使用 SQL Server Express Database 文件 Northwind.mdf。要获得这个数据库，请搜索“Northwind and pubs Sample Database for SQL Server 2000”。这个链接在 <http://www.microsoft.com/downloads/details.aspx?FamilyId=06616212-0356-46A0-8DA2-EEBC53A68034&displaylang=en>。安装后，Northwind.mdf 文件位于 C:\SQL Server 2000 Sample Database 目录下。要把这个数据库添加到应用程序中，可以右击正在使用的解决方案，选择 Add Existing Item 命令。在打开的对话框中，找到刚才安装的 Northwind.mdf 文件。如果在获得使用该数据库的权限方面有问题，那么可以从 Visual Studio Server Explorer 中建立与该文件的数据连接，使自己成为该数据库的相应用户。Visual Studio 就会进行相应的改变，允许使用该数据库。

现在，在 Visual Studio 2010 中创建 .NET Framework 4 提供的许多应用程序类型时，默认已经有了使用 LINQ 的正确引用。在创建控制台应用程序时，代码中会包含如下 using 语句：

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;
```

可以看出，代码已经包含需要的 LINQ 引用。

56.1.2 添加 LINQ to SQL 类

下一步是添加一个 LINQ to SQL 类。使用 LINQ to SQL 时，一个主要优点是 Visual Studio 2010 会使该任务尽可能简单。Visual Studio 提供了一个与对象相关的映射设计器，称为 O/R 设计器，它允许可视化地设计数据库要映射的对象。

首先，右击解决方案，从弹出的菜单中选择 Add New Item 命令。在 Add New Item 对话框中，包含 LINQ to SQL Classes 选项，如图 56-2 所示。

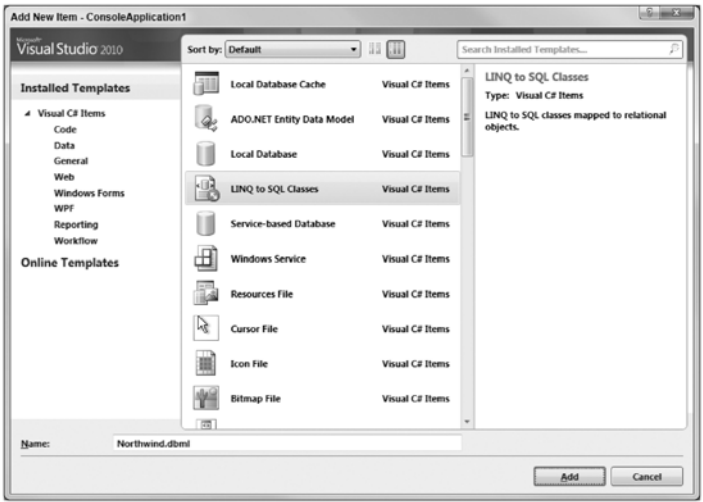


图 56-2

因为这个例子使用 Northwind 数据库，所以把文件命名为 Northwind.dbml。单击 Add 按钮，就会创建几个文件，图 56-3 显示了添加 Northwind.dbml 文件后的 Solution Explorer 窗口。

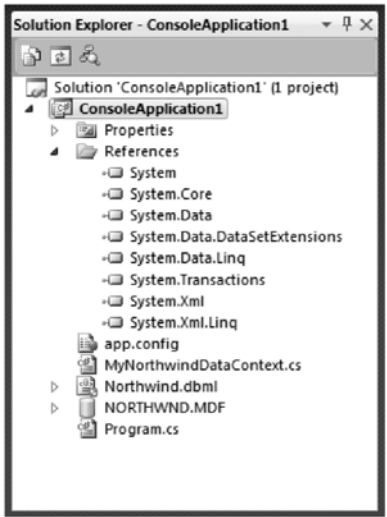


图 56-3

这个操作在项目中添加了许多内容。添加了 Northwind.dbml 文件，它包含两个组件。因为前面添加的 LINQ to SQL 类要使用 LINQ，所以也添加了下面的引用：System.Core、System.Data.DataSetExtensions、System.Data.Linq 和 System.XML.Linq。

56.1.3 O/R 设计器概述

在项目(Northwind.dbml 文件)中添加 LINQ to SQL 类时，还在 IDE 上添加了一个新功能：.dbml 文件的可视化表示。新的 O/R 设计器在 IDE 的文档窗口中显示为一个选项卡。图 56-4 显示了 O/R 设计器第一次启动时的视图。

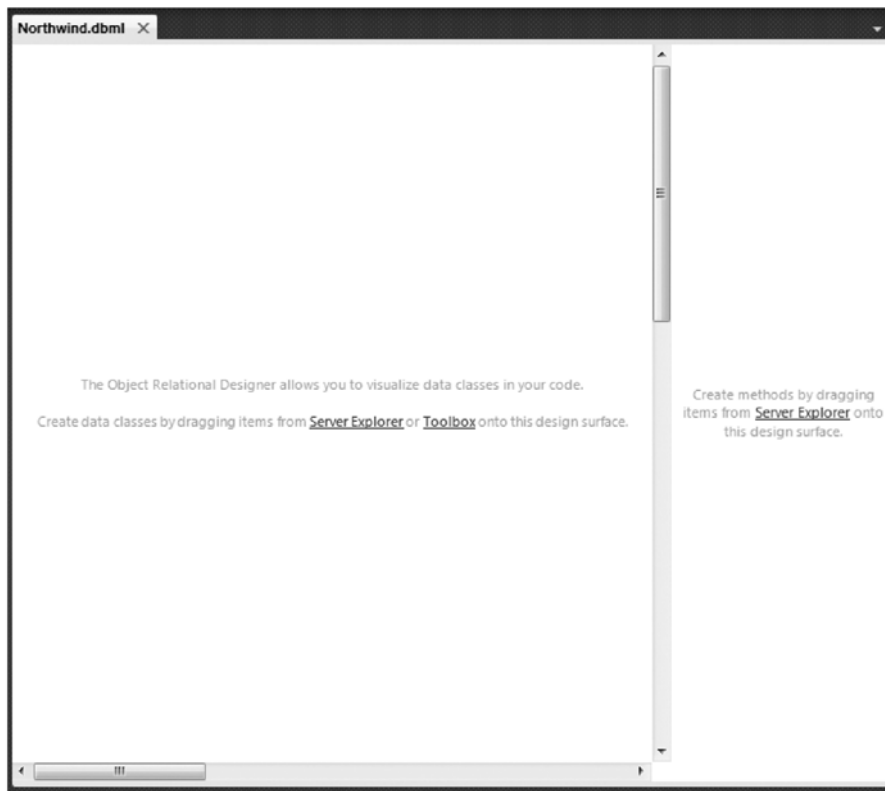


图 56-4

O/R 设计器由两部分组成。第一部分用于显示数据类，它可以是表、类、关联和继承。在这个设计界面上拖动这些项，可以显示所使用的对象的可视化表示。第二部分(右边)用于显示方法，这些方法映射到数据库中的存储过程上。

在 O/R 设计器中查看.dbml 文件时，Visual Studio 工具箱中还有一组 Object Relational Designer 控件。该工具箱如图 56-5 所示。

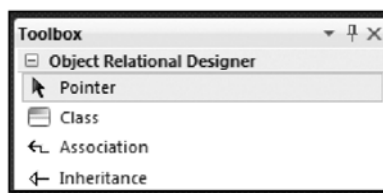


图 56-5

56.1.4 创建 Product 对象

这个例子要使用 Northwind 数据库中的 Products 表，这表示必须创建一个 Products 表，Products 表使用 LINQ to SQL 映射到这个表上。完成该任务只需在 Visual Studio 的 Server Explorer 对话框中打开包含在该数据库中的表对应的视图，把 Products 表拖放到 O/R 设计器的设计界面上。这个操作的结果如图 56-6 所示。

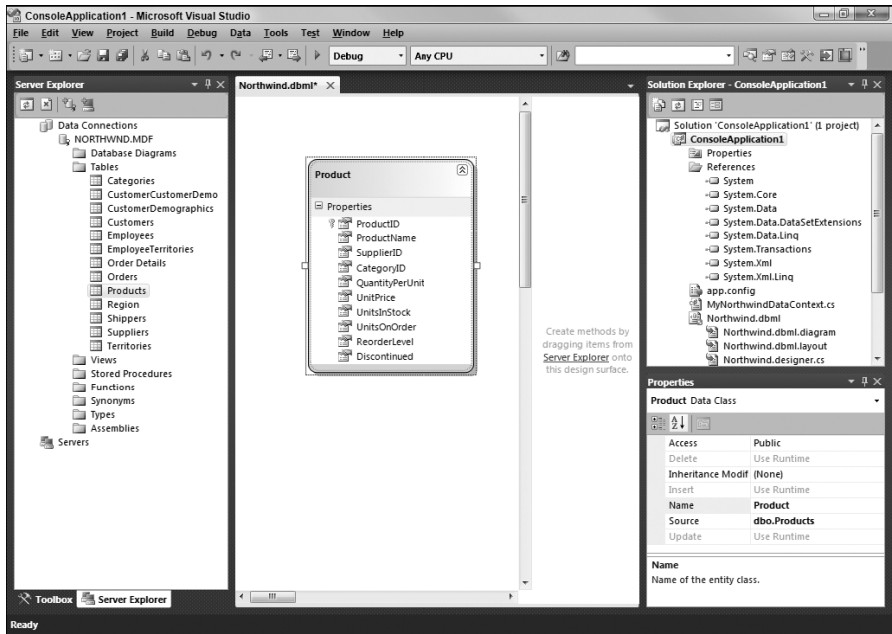


图 56-6

执行这个操作，会把一些代码添加到.dbml 文件的设计器文件中。这些类可以对 Products 表进行强类型化访问。为了演示这个访问，把注意力转向控制台应用程序的 Program.cs 文件上。下面显示了这个例子需要的代码：



```
using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            var query = dc.Products;

            foreach (Product item in query)
            {
                Console.WriteLine("{0} | {1} | {2}",
                    item.ProductID, item.ProductName, item.UnitsInStock);
            }

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

代码下载 ConsoleApplication1.sln

虽然这些代码并不多，但它查询 Northwind 数据库中的 Products 表，提取出要显示的数据。下面单步执行这段代码，从 Mian()方法的第一行开始：

```
NorthwindDataContext dc = new NorthwindDataContext();
```

NorthwindDataContext 对象是 DataContext 类型的一个对象。基本上可以把这个对象看作映射到 Connection 类型对象上的一个对象。这个对象可以使用连接字符串，并为任何需要的操作连接到数据库上。

下一行比较有趣：

```
var query = dc.Products;
```

这里使用了新的 var 关键字，它是一个隐式类型化的变量。如果不能确定输出类型，就可以使用 var 替代类型定义，再在编译期间设置类型。事实上，代码(dc.Products;) 返回一个 System.Data.Linq.Table<ConsoleApplication1.Product> 对象，这就是编译应用程序时设置的 var。因此，这表示也可以编写如下语句：

```
Table<Product>query = dc.Products;
```

这种方法实际上比较好，因为程序员以后查看应用程序的代码时，很容易理解其含义。使用 var 关键字还有一个隐含的优点：程序员可能会发现它有问题。要使用 Table<Product> (这基本上是 Product 对象的一个泛型列表)，就应引用 System.Data.Linq 名称空间。

赋予 query 对象的值是 Products 属性的值，它是 Table<Product> 类型。之后，下面的代码迭代 Table<Product> 中的 Product 对象集合：

```
foreach (Product item in query)
{
    Console.WriteLine("{0} | {1} | {2}",
        item.ProductID, item.ProductName, item.UnitsInStock);
}
```

在本例中，这个迭代从 Product 对象中提取出 ProductID、ProductName 和 UnitsInStock 属性，并把它们写到程序中。因为我们仅使用该表的几项，所以还可以使用 O/R 设计器中的选项删除从数据库中提取的、不感兴趣的列。程序的结果如下：

```
1 | Chai | 39
2 | Chang | 17
3 | Aniseed Syrup | 13
4 | Chef Anton's Cajun Seasoning | 53
5 | Chef Anton's Gumbo Mix | 0

** Results removed for space reasons **

73 | R d Kaviar | 101
74 | Longlife Tofu | 4
75 | Rh nbr u Klosterbier | 125
76 | Lakkalik ri | 57
77 | Original Frankfurter grüne Soße | 32
```

从这个例子可以看出，很容易使用 LINQ to SQL 查询 SQL Server 数据库。

56.2 对象如何映射到 LINQ 对象上

LINQ 的优点是提供了在代码(和 IntelliSense)中使用的强类型化对象，这些对象映射到已有的数

数据库对象上。另外，LINQ 不过是这些已有数据库对象上的一个瘦外观。表 56-1 列出了数据库对象和 LINQ 对象之间的映射。

表 56-1

数据库对象	LINQ 对象
数据库	DataContext
表	类和集合
视图	类和集合
列	属性
关系	嵌套的集合
存储过程	方法

左侧是正在处理的数据库。数据库是一个完整的实体——表、视图、触发器、存储过程构成数据库。在右侧 LINQ 对象中，有一个 DataContext 对象，它绑定到数据库上。为了与数据库进行必要的交互操作，该对象包含一个连接字符串，并管理所发生的所有事务；它还负责记录操作，并管理数据的输出。DataContext 对象通过数据库全面管理事务。

如前面的控制台应用程序例子所示，表转换为类。这表示如果有一个 Products 表，就有一个 Product 类。注意 LINQ 的名称是友好的，因为它把复数形式的表名改为单数形式，给要在代码中使用的类指定合适的名称。除了用作类的数据库表之外，把数据库视图也看作类。另一方面，把列看作属性，因此可以直接管理列的属性(名称和类型定义)。

关系是在各个对象之间映射的嵌套集合。因此可以定义映射到多个项上的关系。

还必须理解存储过程的映射。在代码中，存储过程实际上映射到 DataContext 实例的方法上。下一节将详细介绍 DataContext 实例和 LINQ 中的表对象。

在处理 LINQ to SQL 的体系结构时，注意其中有 3 层：应用层、LINQ to SQL 层和 SQL Server 数据库层。从前面的例子中可以看出，可以在应用程序的代码中创建强类型化的查询：

```
dc.Products;
```

这个查询会被 LINQ to SQL 层转换为 SQL 查询，之后提供给数据库：

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID],
[t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice],
[t0].[UnitsInStock], [t0].[UnitsOnOrder], [t0].[ReorderLevel],
[t0].[Discontinued]
FROM [dbo].[Products] AS [t0]
```

结果，LINQ to SQL 层通过这个查询从数据库中提取行，把返回的数据变成一个强类型化的对象集合，以便于使用。

56.2.1 DataContext 对象

同样，在使用 LINQ to SQL 时，DataContext 对象管理在所使用的数据库中发生的事务。使用 DataContext 对象可以执行许多操作。

在实例化这些对象时，需要几个可选参数，如下所示：

- 一个字符串，表示 SQL Server Express 数据库文件的位置或所使用的 SQL Server 的名称
- 连接字符串
- 另一个 DataContext 对象

前两个可选字符串参数也可以包含自己的数据库映射文件。实例化这个对象后，就可以在许多不同的操作中以编程方式使用它。

1. 使用 ExecuteQuery() 方法

使用 DataContext 对象完成的一个简单任务是用 ExecuteQuery<T>() 方法快速运行自己编写的命令。例如，如果要使用 ExecuteQuery<T>() 方法提取 Products 表中的所有产品，那么代码应如下所示：



```
using System;
using System.Collections.Generic;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            DataContext dc = new DataContext(@"Data Source=.\\SQLEXPRESS;
            AttachDbFilename=|DataDirectory|\\NORTHWND.MDF;
            Integrated Security=True;User Instance=True");

            IEnumerable<Product>myProducts =
                dc.ExecuteQuery<Product>("SELECT * FROM PRODUCTS", "");

            foreach (Product item in myProducts)
            {
                Console.WriteLine(item.ProductID + " | " + item.ProductName);
            }

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

代码下载 ConsoleApplication1.sln

在这个例子中，调用 ExecuteQuery<T>() 方法时传递了一个查询字符串，并返回一个 Product 对象集合。在该方法调用中使用的查询是一条简单的 Select 语句，它不需要传递任何参数。由于在查询中没有传递任何参数，因此可以使用双引号作为方法调用的第二个必选参数，甚至也可以不提供这个参数。如果要在查询中替换任何值，那么可以按如下方式构建 ExecuteQuery<T>() 方法调用：

```
IEnumerable <Product> myProducts =
    dc.ExecuteQuery<Product>("SELECT * FROM PRODUCTS WHERE UnitsInStock > {0}",
    50);
```

在这个例子中，{0} 是一个占位符，用于替换要传入的参数值，ExecuteQuery<T>() 方法的第二个参数是在替换过程中使用的参数。

2. 使用 Connection 属性

Connection 属性返回 System.Data.SqlClient.SqlConnection(由 DataContext 对象使用)的一个实例。如果需要与应用程序中使用的其他 ADO.NET 代码共享这个连接,或者需要获得该连接提供的 SqlConnection 属性或方法,就可以使用这个属性。例如,获得连接字符串就很简单:

```
NorthwindDataContext dc = new NorthwindDataContext();

Console.WriteLine(dc.Connection.ConnectionString);
```

3. 使用 ADO.NET 事务

如果有一个可以使用的 ADO.NET 事务,就可以使用 Transaction 属性把这个事务赋予 DataContext 对象实例。还可以通过 .NET 2.0 Framework 中的 TransactionScope 对象使用事务(需要引用 System.Transactions 程序集):



```
using System;
using System.Collections.Generic;
using System.Data.Linq;
using System.Transactions;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            using (TransactionScope myScope = new TransactionScope())
            {
                Product p1 = new Product() { ProductName = "Bill's Product" };
                dc.Products.InsertOnSubmit(p1);

                Product p2 = new Product() { ProductName = "Another Product" };
                dc.Products.InsertOnSubmit(p2);

                try
                {
                    dc.SubmitChanges();

                    Console.WriteLine(p1.ProductID);
                    Console.WriteLine(p2.ProductID);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.ToString());
                }

                myScope.Complete();
            }

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

```
}

```

代码段 ConsoleApplication1.sln

在这个例子中，使用了 `TransactionScope` 对象，如果数据库中的某个操作失败，就把所有操作滚到原始状态。

4. DataContext 对象的其他方法和属性

除了前面介绍的项之外，`DataContext` 对象还有许多其他可用的方法和属性。表 56-2 列出了 `DataContext` 对象的一些方法。

表 56-2

方 法	说 明
<code>CreateDatabase</code>	可以在服务器上创建数据库
<code>DatabaseExists</code>	可以确定数据库是否存在，是否可以打开
<code>DeleteDatabase</code>	删除相关联的数据库
<code>ExecuteCommand</code>	可以给数据库传入要执行的命令
<code>ExecuteQuery</code>	可以给数据库直接传递查询
<code>GetChangeSet</code>	<code>DataContext</code> 对象跟踪数据库中发生的变化，这个方法可以访问这些变化
<code>GetCommand</code>	可以访问要执行的命令
<code>GetTable</code>	可以访问数据库中的表集合
<code>Refresh</code>	可以利用数据库中存储的数据刷新对象
<code>SubmitChanges</code>	在代码中建立的数据库中执行 CRUD 命令
<code>Translate</code>	把 <code>IDataReader</code> 转换为对象

除了这些方法之外，`DataContext` 对象还包含一些属性，如表 56-3 所示。

表 56-3

属 性	说 明
<code>ChangeConflicts</code>	调用 <code>SubmitChanges()</code> 方法时，提供一个导致并发冲突的对象集合
<code>CommandTimeout</code>	可以设置一个超时期限，允许在这个时间段内在数据库上运行命令。如果查询需要执行较长时间，就应把这个属性设置为较高的值
<code>Connection</code>	可以使用客户端使用的 <code>System.Data.SqlClient.SqlConnection</code> 对象
<code>DeferredLoadingEnabled</code>	可以指定是否延迟加载一对多或一对一关系
<code>LoadOptions</code>	可以指定或检索 <code>DataLoadOptions</code> 对象的值
<code>Log</code>	可以指定在查询中使用的命令的输出位置
<code>Mapping</code>	提供映射所基于的 <code>MetaModel</code>
<code>ObjectTrackingEnabled</code>	指定是否跟踪数据库中对象的变化，以进行事务处理。如果正在处理只读数据库，就应把这个属性设置为 <code>false</code>
<code>Transaction</code>	可以指定数据库中使用的本地事务

56.2.2 Table<TEntity>对象

Table<TEntity>对象表示在数据库中使用的表。例如，前面使用了 Product 类，它就是一个 Table<Product>实例。如本章所述，许多方法都可用于 Table<TEntity>对象。其中一些方法在表 56-4 中定义。

表 56-4

方 法	说 明
Attach	可以把一个实体关联到 DataContext 实例上
AttachAll	可以把一个实体集合关联到 DataContext 实例上
DeleteAllOnSubmit<TSubEntity>	可以把所有挂起的操作置于准备删除的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行这里的所有操作
DeleteOnSubmit	可以把一个挂起的操作置于准备删除的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行这里的所有操作
GetModifiedMembers	提供一个已修改的对象数组，可以访问它们的当前值和更改后的值
GetNewBindingList	提供一个新列表，以绑定到数据存储器上
GetOriginalEntityState	提供一个对象的实例，且显示其初始状态
InsertAllOnSubmit<TSubEntity>	可以把所有挂起的操作置于准备插入的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行这里的所有操作
InsertOnSubmit	可以把一个挂起的操作置于准备插入的状态。从 DataContext 对象上调用 SubmitChanges()方法时，就执行这里的所有操作

56.3 脱离 O/R 设计器工作

Visual Studio 2010 中的新 O/R 设计器很容易创建 LINQ to SQL 需要的所有对象，但基础架构允许从头开始完成所有任务。这将最大限度地控制实际发生的事件。

56.3.1 创建自己的自定义对象

为了完成与前面 Customer 表相同的任务，需要通过一个类提供 Customer 表。首先在项目 (Customer.cs)中创建一个新类，这个类的代码如下所示：



可从
wrox.com
下载源代码

```
using System.Data.Linq.Mapping;

namespace ConsoleApplication1
{
    [Table(Name = "Customers")]
    public class Customer
    {
        [Column(IsPrimaryKey = true)]
        public string CustomerID { get; set; }
        [Column]
        public string CompanyName { get; set; }
        [Column]
```

```

        public string ContactName { get; set; }
        [Column]
        public string ContactTitle { get; set; }
        [Column]
        public string Address { get; set; }
        [Column]
        public string City { get; set; }
        [Column]
        public string Region { get; set; }
        [Column]
        public string PostalCode { get; set; }
        [Column]
        public string Country { get; set; }
        [Column]
        public string Phone { get; set; }
        [Column]
        public string Fax { get; set; }
    }
}

```

代码段 Customer.cs

这里, Customer.cs 文件定义要用于 LINQ to SQL 的 Customer 对象。这个类把 Table 属性赋予它, 用于表示表类。Table 类属性包含一个 Name 属性, 它定义在数据库中使用的表的名称, 这个表名要通过连接字符串中引用。使用 Table 属性还表示, 需要在代码中引用 System.Data.Linq.Mapping 名称空间。

除了 Table 属性之外, 类中定义每个属性都使用 Column 属性。如前所述, SQL Server 数据库中的列映射到代码的属性上。

56.3.2 通过自定义对象和 LINQ 查询

只有具备 Customer 类, 才可以查询 Northwind 数据库中的 Customers 表。完成这一任务的代码通过同一控制台应用程序中的以下示例说明:

```

using System;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            DataContext dc = new DataContext(@"Data Source=.\SQLEXPRESS;
            AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
            Integrated Security=True;User Instance=True");

            dc.Log = Console.Out; // Used for outputting the SQL used

            Table<Customer>myCustomers = dc.GetTable<Customer>();

            foreach (Customer item in myCustomers)
            {
                Console.WriteLine("{0} | {1}",

```

```

        item.CompanyName, item.Country);
    }
    Console.ReadLine();
}
}
}

```

在这个例子中，使用默认的 `DataContext` 对象，把包含 SQL Server Express 数据库 Northwind 的连接字符串作为一个参数传递。再使用 `GetTable<TEntity>()` 方法填充 `Customer` 类型的 `Table` 类。这个例子中的 `GetTable<TEntity>()` 操作使用定制的 `Customer` 类：

```
dc.GetTable<Customer>();
```

LINQ to SQL 使用 `DataContext` 对象在 SQL Server 数据库上执行查询，把返回的行作为强类型化的 `Customer` 对象。这就允许迭代 `Table` 对象的集合中的每个 `Customer` 对象，获得需要的信息，如下面的 `Console.WriteLine()` 语句所示：

```

foreach (Customer item in myCustomers)
{
    Console.WriteLine("{0} | {1}",
        item.CompanyName, item.Country);
}

```

运行这段代码，会在控制台应用程序中生成如下结果：

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [Customers] AS [t0]
--Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 4.0.21006.1

Alfreds Futterkiste | Germany
Ana Trujillo Emparedados y helados | Mexico
Antonio Moreno Taquería | Mexico
Around the Horn | UK
Berglunds snabbköp | Sweden

// Output removed for clarity

Wartian Herkku | Finland
Wellington Importadora | Brazil
White Clover Markets | USA
Wilman Kala | Finland
Wolski Zajazd | Poland

```

56.3.3 通过查询限制所调用的列

注意，该查询检索在 `Customer` 类文件中指定的每一列。如果删除不需要的列，就可以得到一个全新的 `Customer` 类文件，如下所示：



```

using System.Data.Linq.Mapping;

namespace ConsoleApplication1
{
    [Table(Name = "Customers")]

```

```

public class Customer
{
    [Column(IsPrimaryKey = true)]
    public string CustomerID { get; set; }
    [Column]
    public string CompanyName { get; set; }
    [Column]
    public string Country { get; set; }
}

```

代码段 Customer.cs

这里删除了应用程序未使用的所有列。现在，如果运行控制台应用程序，并查看生成的 SQL 查询，结果如下：

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[Country]
FROM [Customers] AS [t0]

```

可以看出，在对 Customers 表的查询中，仅使用在 Customer 类中定义的 3 列。

CustomerID 属性比较有趣，因为可以在 Column 属性中使用 IsPrimaryKey 设置，把该列指定为表的主键。这个设置接受一个布尔值，这里它设置为 true。

56.3.4 使用列名

列的另一个要点是在 Customer 类中定义的属性名必须与数据库中使用的名称相同。例如，如果把 CustomerID 属性名改为 MyCustomerID，尝试运行控制台应用程序时，就会得到如下异常：

```

System.Data.SqlClient.SqlException was unhandled
  Message="Invalid column name 'MyCustomerID'."
  Source=".Net SqlClient Data Provider"
  ErrorCode=-2146232060
  Class=16
  LineNumber=1
  Number=207
  Procedure=""
  Server="\\\\.\\pipe\\F5E22E37-1AF9-44\\tsql\\query"

```

为了改正这个错误，需要在前面创建的 Customer 自定义类中定义列的名称。为此，可以使用 Column 属性，如下所示：

```

[Column(IsPrimaryKey = true, Name = "CustomerID")]
public string MyCustomerID { get; set; }

```

与 Table 属性一样，Column 属性也包含一个 Name 属性，Name 属性指定列在 Customers 表中显示的名称。

这会生成如下查询：

```

SELECT [t0].[CustomerID] AS [MyCustomerID], [t0].[CompanyName], [t0].[Country]
FROM [Customers] AS [t0]

```

这也意味着，需要使用新的名称 MyCustomerID 引用该列，如 item.MyCustomerID。

56.3.5 创建自己的 DataContext 对象

使用普通的 DataContext 对象有时并不是最好的方法，而创建自己的 DataContext 类可以进行更多控制。为此，创建一个新类 MyNorthwindDataContext.cs，使这个类继承自 DataContext。该类最简单的形式如下：



```
using System.Data.Linq;

namespace ConsoleApplication1
{
    public class MyNorthwindDataContext: DataContext
    {
        public Table<Customer>Customers;

        public MyNorthwindDataContext()
            : base(@"Data Source=.\SQLEXPRESS;
                  AttachDbFilename=|DataDirectory|\NORTHWND.MDF;
                  Integrated Security=True;User Instance=True")
        {
        }
    }
}
```

代码段 MyNorthwindDataContext.cs

这里，MyNorthwindDataContext 类继承自 DataContext，从前面创建的 Customer 类中提供 Table<Customer>对象的一个实例。构造函数是这个类的另一个要求。这个构造函数使用一个基本实例初始化对象的新实例，这个对象引用文件(这里是 SQL 数据库文件的一个连接)。

现在使用自己的 DataContext 对象，可以改变应用程序中的代码，如下所示：



```
using System;
using System.Data.Linq;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            MyNorthwindDataContext dc = new MyNorthwindDataContext();
            Table<Customer> myCustomers = dc.Customers;

            foreach (Customer item in myCustomers)
            {
                Console.WriteLine("{0} | {1}",
                                    item.CompanyName, item.Country);
            }

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

代码下载 ConsoleApplication1.sln

创建 `MyNorthwindDataContext` 对象的实例后, 就允许该类管理到数据库的连接。现在还可以通过 `dc.Customers` 语句直接访问 `Customer` 类。



注意, 本章的例子都比较纯粹, 因为它们都没有包含构建应用程序时通常包含的错误处理和日志功能, 而只是阐述本章讨论的要点。

56.4 自定义对象和 O/R 设计器

除了在自己的.cs 文件中构建自定义对象再把该类关联到自己构建的 `DataContext` 对象上之外, 还可以在 Visual Studio 2010 中使用 O/R 设计器构建自己的类文件。以这种方式使用 Visual Studio 时, Visual Studio 会创建相应的.cs 文件, O/R 设计器还提供了类文件和自己建立的所有关系的可视化表示。

在查看.dbml 文件的设计器视图时, 注意工具箱中有 3 项: `Class`、`Association` 和 `Inheritance`。

对于这个例子, 从工具箱中选择 `Class` 对象, 把它拖放到设计界面上。这会显示泛型类的图像, 如图 56-7 所示。

现在可以单击 `Class1` 名称, 把这个类重命名为 `Customer`。右击该名称的旁边, 从弹出的菜单中选择 `Add | Property` 命令, 可以给类文件添加属性。对于这个例子, 给 `Customer` 类添加 3 个属性——`CustomerID`、`CompanyName` 和 `Country`。如果突出显示 `CustomerID` 属性, 那么可以在 Visual Studio 的 `Properties` 对话框中配置该属性, 把 `Primary Key` 设置从 `False` 改为 `True`。也可以突出显示整个类, 进入 `Properties` 对话框, 把 `Source` 属性改为 `Customers`, 因为这是 `Customer` 对象需要使用的表名。之后, 类的可视化表示就如图 56-8 所示。



图 56-7

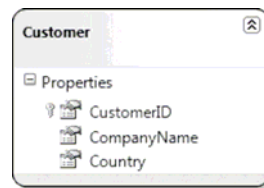


图 56-8

从图 56-8 可以看出, `CustomerID` 属性的名称旁边可能显示主键图标。此时单击 `Northwind.dbml` 文件旁边的加号, 就会看到 3 个文件——`Northwind.dbml.layout`、`Northwind.designer.cs` 和 `Northwind.dbml.diagram`。`Northwind.dbml.layout` 文件是有助于 Visual Studio 在 O/R 设计器中进行可视化表示的 XML 文件。但最重要的文件是 `Northwind.designer.cs`, 这是我们创建的 `Customer` 类文件。打开这个文件, 可以查看 Visual Studio 创建的对象。

首先, 会发现 `Customer` 类文件在页面的代码中:

```
[Table(Name="Customers")]
public partial class Customer: INotifyPropertyChanging,
                               INotifyPropertyChanged
{
    // Code removed for clarity
}
```

`Customer` 类是根据我们在设计器中提供的名称而指定的类名。这个类有一个 `Table` 属性, 其值

是 Customers，因为这是该对象在连接到 Northwind 数据库上时需要使用的数据库的名称。

在 Customer 类中，会发现前面定义的 3 个属性，这里只列出其中一个属性——CustomerID：



```
[Column(Storage="_CustomerID", CanBeNull=false, IsPrimaryKey=true)]
public string CustomerID
{
    get
    {
        return this._CustomerID;
    }
    set
    {
        if ((this._CustomerID != value))
        {
            this.OnCustomerIDChanging(value);
            this.SendPropertyChanging();
            this._CustomerID = value;
            this.SendPropertyChanged("CustomerID");
            this.OnCustomerIDChanged();
        }
    }
}
```

代码段 Customer.cs

与前面示例在构建立类时的方式类似，所定义的属性使用 Column 特性和对这个特性可用的一些属性。例如，用 IsPrimaryKey 项设置主键。

除了 Customer 类之外，在创建的文件中还有一个继承自 DataContext 对象的类：

```
[System.Data.Linq.Mapping.DatabaseAttribute(Name="NORTHWND")]
public partial class NorthwindDataContext: System.Data.Linq.DataContext
{
    // Code removed for clarity
}
```

DataContext 对象(NorthwindDataContext)可以连接到 Northwind 数据库和 Customer 表上，与前面的例子一样。

使用 O/R 设计器可以使数据库对象类文件的创建更简单、直接。然而，同时，如果要完全控制，就可以自己编写所有代码，得到期望的结果。

56.5 查询数据库

如前所述，在应用程序的代码中有许多方式查询数据库。该查询最简单的形式如下：

```
Table<Product> query = dc.Products;
```

这条命令把整个 Products 表放在 query 对象实例中。

56.5.1 使用查询表达式

除了使用 dc.Products 把表直接提取出数据库之外，还可以在代码中直接使用强类型化的查询表

达式。下面的代码就是一个例子：



可从
wrox.com
下载源代码

```
using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            var query = from p in dc.Products
                        select p;
            foreach (Product item in query)
            {
                Console.WriteLine(item.ProductID + " | " + item.ProductName);
            }

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

代码下载 ConsoleApplication1.sln

在这个例子中，用“from p in dc.Products select p;”的查询值填充 query 对象(同样是 Table<Product> 对象)。为了增强可读性，把这条命令放在两行上，根据个人爱好也可以放在一行上。

56.5.2 查询表达式

在代码中可以使用许多查询表达式，上面的例子仅是一条简单的 select 语句，它返回整个表。表 56-5 列出了其他一些查询表达式。

表 56-5	
选 项	语 法
Project	select <expression>
Filter	where <expression>, distinct
Test	any(<expression>), all(<expression>)
Join	<expression> join <expression> on <expression> equals <expression>
Group	group by <expression>, into <expression>, <expression> group join<decision> on <expression> equals <expression> into <expression>
Aggregate	count([<expression>]), sum(<expression>), min(<expression>), max(<expression>), avg(<expression>)
Patition	skip [while] <expression>, take [while] <expression>
Set	union, intersect, except
Order	order by <expression>, <expression> [ascending descending]

56.5.3 使用表达式筛选

除了直接查询整个表之外，还可以使用 `where` 和 `distinct` 选项筛选数据项。下面的例子就查询 `Products` 表中指定类型的记录：

```
var query = from p in dc.Products
             where p.ProductName.StartsWith("L")
             select p;
```

在该示例中，这个查询从 `Products` 表中选择所有以字母 `L` 开头的记录。这通过 `where p.ProductName.StartsWith("L")` 表达式实现。`ProductName` 属性有许多选择方法，可以细调需要的筛选条件。这个操作生成如下结果：

```
65 | Louisiana Fiery Hot Pepper Sauce
66 | Louisiana Hot Spiced Okra
67 | Laughing Lumberjack Lager
74 | Longlife Tofu
76 | Lakkalikööri
```

根据需要还可以给列表添加任意多个表达式。例如，下面的例子给查询添加了两条 `where` 语句：

```
var query = from p in dc.Products
             where p.ProductName.StartsWith("L")
             where p.ProductName.EndsWith("i")
             select p;
```

其中一个筛选表达式查找产品名称以字母 `L` 开头的项，第二个表达式确保还应用了第二个条件，即对应项必须以字母 `i` 结尾。这会生成如下结果：

```
76 | Lakkalik      ri
```

56.5.4 执行连接

除了使用一个表之外，还可以使用多个表，用查询执行连接。如果把 `Customers` 表和 `Orders` 表都拖放到 `Northwind.dbml` 设计界面上，就会得到如图 56-9 所示的结果：

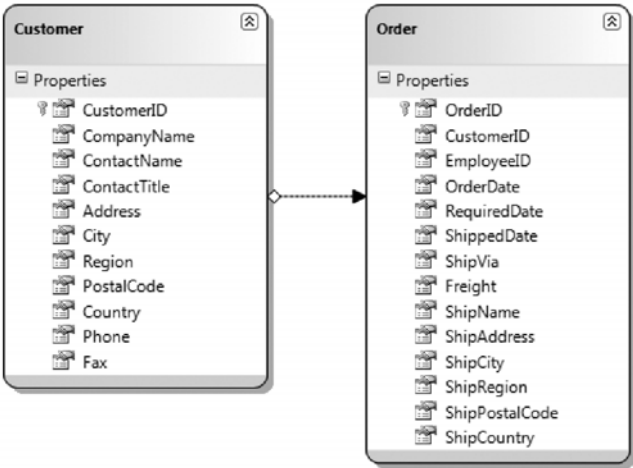


图 56-9

从图 56-9 可以看出, 把这些元素拖放到设计界面上后, Visual Studio 就知道这些项之间存在一个关系, 并在代码中创建这个关系, 用黑色箭头表示它。

现在, 就可以在查询中通过 join 语句使用这两个表, 如下面的例子所示:



```
using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();
            dc.Log = Console.Out;

            var query = from c in dc.Customers
                        join o in dc.Orders on c.CustomerID equals o.CustomerID
                        orderby c.CustomerID
                        select new { c.CustomerID, c.CompanyName,
                                    c.Country, o.OrderID, o.OrderDate };

            foreach (var item in query)
            {
                Console.WriteLine(item.CustomerID + " | " + item.CompanyName
                                   + " | " + item.Country + " | " + item.OrderID
                                   + " | " + item.OrderDate);
            }

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

代码段 ConsoleApplication1.sln

这个例子从 Customers 表中提取数据, 并连接 Orders 表中与 CustomerID 列匹配的记录。这通过 join 语句实现:

```
join o in dc.Orders on c.CustomerID equals o.CustomerID
```

接着用 select new 语句创建一个新对象, 这个新对象包含 Customers 表中的 CustomerID、CompanyName、Country 列, 以及 Orders 表中的 OrderID 和 OrderDate 列。

在迭代这个新对象的集合时, 有趣的是 foreach 语句还使用 var 关键字, 因为该类型在此刻未知:

```
foreach (var item in query)
{
    Console.WriteLine(item.CustomerID + " | " + item.CompanyName
                       + " | " + item.Country + " | " + item.OrderID
                       + " | " + item.OrderDate);
}
```

无论如何, item 对象可以访问我们指定的所有属性。运行这个示例, 会得到类似于如下部分结果的输出:

```

WILMK | Wilman Kala | Finland | 10695 | 10/7/1997 12:00:00 AM
WILMK | Wilman Kala | Finland | 10615 | 7/30/1997 12:00:00 AM
WILMK | Wilman Kala | Finland | 10673 | 9/18/1997 12:00:00 AM
WILMK | Wilman Kala | Finland | 11005 | 4/7/1998 12:00:00 AM
WILMK | Wilman Kala | Finland | 10879 | 2/10/1998 12:00:00 AM
WILMK | Wilman Kala | Finland | 10873 | 2/6/1998 12:00:00 AM
WILMK | Wilman Kala | Finland | 10910 | 2/26/1998 12:00:00 AM

```

56.5.5 分组项

也可以通过查询对项分组。在 Northwind.dbml 示例中, 把 Categories 表拖放到设计界面上, 会发现在这个表和 Products 表之间存在一个关系。下面的例子说明了如何按类别对产品分组:



可从
wrox.com
下载源代码

```

using System;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

            var query = from p in dc.Products
                        orderby p.Category.CategoryName ascending
                        group p by p.Category.CategoryName into g
                        select new { Category = g.Key, Products = g };

            foreach (var item in query)
            {
                Console.WriteLine(item.Category);

                foreach (var innerItem in item.Products)
                {
                    Console.WriteLine(" " + innerItem.ProductName);
                }

                Console.WriteLine();
            }

            Console.ReadLine();
        }
    }
}

```

代码下载 ConsoleApplication1.sln

这个例子创建了一个新对象, 它是一组类别, 再把整个 Product 表打包到这个新表(g)中。在此之前, 类别使用 orderby 语句按名称排序, 因为所提供的订单按升序排列(另一个选项是降序)。输出是 Category(通过 Key 属性传递)和 Product 实例。foreach 语句的迭代对类别迭代一次, 给在类别中找到的每种产品迭代一次。

这个程序的部分输出结果如下:

```

Beverages
    Chai

```

```

    Chang
    Guaran á Fant á stica
    Sasquatch Ale
    Steeleye Stout
    Côte de Blaye
    Chartreuse verte
    Ipoh Coffee
    Laughing Lumberjack Lager
    Outback Lager
    Rhönbräu Klosterbier
    Lakkalikööri

    Condiments
    Aniseed Syrup
    Chef Anton's Cajun Seasoning
    Chef Anton's Gumbo Mix
    Grandma's Boysenberry Spread
    Northwoods Cranberry Sauce
    Genen Shouyu
    Gula Malacca
    Sirop d'érable
    Vegie-spread
    Louisiana Fiery Hot Pepper Sauce
    Louisiana Hot Spiced Okra
    Original Frankfurter grüne Soße

```

除了本章介绍的命令和表达式之外，还有许多其他的命令和表达式。

56.6 存储过程

前面都是直接查询表，使 LINQ 为操作创建相应的 SQL 语句。在使用那些大量使用存储过程的已有数据库和遵循使用存储过程的最佳实践的数据库时，LINQ 也是一个可用的选项。

LINQ to SQL 把存储过程看作方法调用。如图 56-4 所示，O/R 设计器可以把表拖放到其设计界面上，之后就可以通过编程方式使用表。在 O/R 设计器的右侧，有一个区域可以拖放存储过程。

拖放到 O/R 设计器这个部分的任何存储过程都变成可以在 DataContext 对象中可用的方法。例如，把 TenMostExpensiveProducts 存储过程拖放到 O/R 设计器的这个部分上。

下面的例子说明了如何在 Northwind 数据库中调用这个存储过程：



可从
wrox.com
下载源代码

```

using System;
using System.Collections.Generic;
using System.Data.Linq;
using System.Linq;

namespace ConsoleApplication1
{
    class Class1
    {
        static void Main(string[] args)
        {
            NorthwindDataContext dc = new NorthwindDataContext();

```

```
        ISingleResult<Ten_Most_Expensive_ProductsResult> result =  
            dc.Ten_Most_Expensive_Products();  
  
        foreach (Ten_Most_Expensive_ProductsResult item in result)  
        {  
            Console.WriteLine(item.TenMostExpensiveProducts + " | " +  
                item.UnitPrice);  
        }  
  
        Console.ReadLine();  
    }  
}
```

代码下载 ConsoleApplication1.sln

从这个例子中可以看出，存储过程输出的行被收集到 `ISingleResult<Ten_Most_Expensive_ProductsResult>` 对象中。之后，迭代这个对象，这与余下的操作一样简单。

从这个例子可以看出，调用存储过程是很简单的过程。

56.7 小结

.NET Framework 4 版本的一个激动人心的功能是该平台提供的 LINQ 功能。本章介绍了 LINQ to SQL 的使用和查询 SQL Server 数据库时可用的一些选项。

使用 LINQ to SQL 可以通过一组强类型化的操作对数据库执行 CRUD 操作。也可以使用已有的访问功能，不管是与 ADO.NET 交互，还是使用存储过程。

第 57 章

WPF 3.0

本章内容:

- 不同类型的工作流: 顺序工作流和状态机工作流
- 内置活动
- 如何创建自定义活动
- 工作流服务
- 与 WCF 的集成
- 驻留工作流
- 迁移到 Workflow Foundation 4 的提示

本章将概述 Windows Workflow Foundation 3.0(本章称之为 WF), 它提供一个模型, 在该模型中, 可以使用一组构建块(称为活动)定义和执行进程。WF 还提供了一个设计器, 在默认情况下, 该设计器驻留在 Visual Studio 中, 允许将工具箱中的活动拖放到设计界面上, 创建一个工作流模板。

创建一个 WorkflowInstance, 运行该实例, 就可以执行这个模板。执行工作流的代码称为 WorkflowRuntime, 该对象也可以驻留许多服务, 正在运行的工作流可以访问它们。在任意时刻, 均有几个工作流实例在执行, 运行库负责调度这些实例, 保存和还原状态, 它还可以记录每个工作流实例的执行情况。

工作流由许多活动构成, 这些活动由运行库执行。活动可以是发送电子邮件、更新数据库中的一行, 或在后端系统上执行一个事务。有许多内置活动, 它们用于一般性的工作, 也可以创建自己的自定义活动, 根据需要将它们放在工作流中。

注意 Visual Studio 2010 有 Windows Workflow 的一个新版本, 它不后向兼容。本章介绍工作流的老版本。对于新项目, 建议使用 WF 4 版本, 更多信息参见第 44 章。本章末尾还提供了一些升级到 WF 4 的建议。

本章从一个规范的例子 Hello World 开始, 每个人在面对一种新技术时都要使用这个例子, 并描述在开发计算机上使工作流运行所需要做的工作。

57.1 Hello World 示例

Visual Studio 2010 包含对创建工作流的内置支持。打开 New Project 对话框, 会看到一个工作流

项目类型列表，如图 57-1 所示。

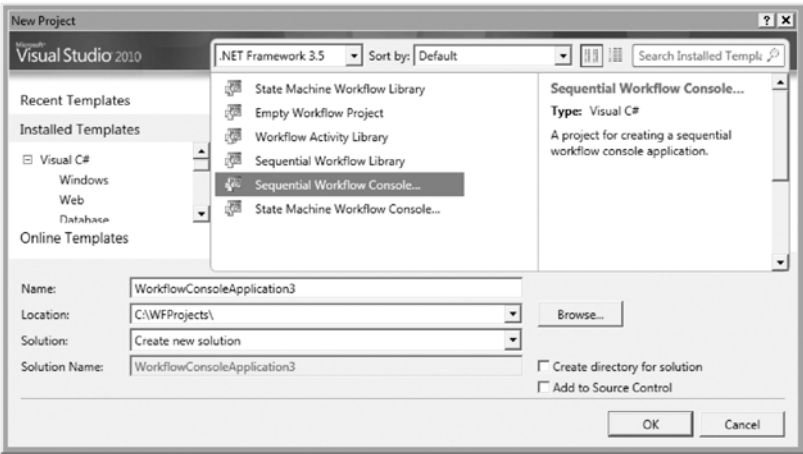


图 57-1

从可用的模板中选择 Sequential Workflow Console Application(它会创建一个驻留工作流运行库的控制台应用程序)和默认的工作流，以后要在该工作流上拖放活动。

接着，把 Code 活动从工具箱拖放到设计界面上，这样就会得到如图 57-2 所示的工作流。

该活动右上角的感叹号标志符号表示，没有定义该活动的强制属性，这里它是 ExecuteCode 属性，它指定活动执行时调用的方法。57.3.1 节将学习如何把自己的属性标记为强制属性。如果双击 Code 活动，就在代码隐藏类中创建一个方法，该方法使用 Console.WriteLine 输出 Hello World 字符串，如下面的代码所示：

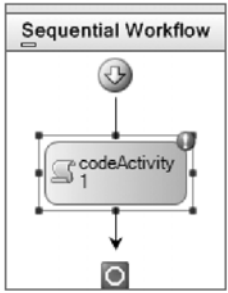


图 57-2

```
private void codeActivity1_ExecuteCode(object sender, EventArgs e)
{
    Console.WriteLine("Hello World");
}
```

如果构建并运行程序，就会在控制台上看到输出的文本。程序执行时，创建了一个 WorkflowRuntime 类型的实例，然后构建一个工作流实例，并执行它。在执行 Code 活动时，它调用已定义的方法，和将字符串输出到控制台上的方法。57.5 节将详细描述如何驻留运行库。上述示例的代码在 01 HelloWorldWorkflowWorld 文件夹中。

57.2 活动

工作流中的内容是一个活动——该活动位于工作流中，其类型比较特殊，它通常允许在其中定义其他活动，这称为复合活动，本章的后面将介绍其他复合活动。活动是一个最终派生自 Activity 类的类。

Activity 类定义许多可重写的方法，其中最重要的是 `Execute()` 方法，如下面的代码段所示：

```
protected override ActivityExecutionStatus Execute
( ActivityExecutionContext executionContext )
{
    return ActivityExecutionStatus.Closed;
}
```

在运行库调度活动以便执行时，最终会调用 `Execute()` 方法，在该方法中，可以编写自定义代码，提供活动的行为。在上一节的简单示例中，当 workflow 运行库在 `CodeActivity` 上调用 `Execute()` 方法时，这个方法的实现代码就会执行在代码隐藏类中定义的方法，在控制台上显示该消息。

`Execute()` 方法需要一个 `ActivityExecutionContext` 类型的上下文参数，本章的后面将探讨这个参数。该方法的返回值是 `ActivityExecutionStatus` 类型，该返回值由运行库用于确定活动是已成功完成、仍在处理，还是处于其他几个状态中的一个状态，这几个状态可以向 workflow 运行库描述活动所处的状态。从这个方法中返回 `ActivityExecutionStatus.Closed`，表示活动已完成其工作，可以释放它。

WF 提供了许多标准活动，下面几节就介绍其中一些活动的例子，并说明使用这些活动的场合。活动的命名约定是在名称的后面追加 `Activity`。例如，图 57-2 中的代码活动就由 `CodeActivity` 类定义。

所有标准活动都在 `System.Workflow.Activities` 名称空间中定义，该名称空间位于 `System.Workflow.Activities.dll` 程序集中。还有另外两个程序集——`System.Workflow.ComponentModel.dll` 和 `System.Workflow.Runtime.dll` 也是 WF 的组成部分。

57.2.1 IfElseActivity

顾名思义，这个活动的操作类似于 C# 中的 If-Else 语句。

当将一个 `IfElseActivity` 拖放到设计界面上时，会看到如图 57-3 所示的活动。`IfElseActivity` 是一个复合活动，它构建两个分支(这两个分支的类型也是活动，这里是 `IfElseBranchActivity`)。每个分支也都是派生自 `SequenceActivity` 的复合活动，这个类自上而下执行每个活动。设计器添加了 `Drop Activities Here` 文本，指出可以把子活动添加到什么地方。

如图 57-3 所示，第一个分支包含一个符号，指定需要定义 `Condition` 属性。条件派生自 `ActivityCondition`，并用于确定是否执行该分支。

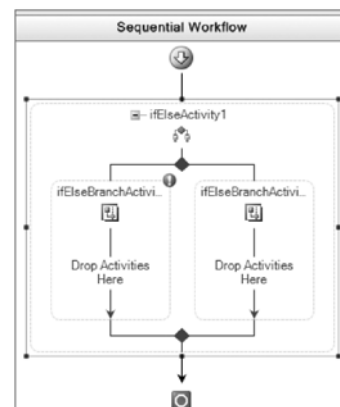


图 57-3

```
protected override ActivityExecutionStatus Execute
( ActivityExecutionContext executionContext )
{
    return ActivityExecutionStatus.Closed;
}
```

在执行 `IfElseActivity` 时，将判断第一个分支的条件，如果该条件等于 `true`，就执行该分支。如果条件等于 `false`，`IfElseActivity` 就尝试执行下一个分支，依此类推，直到活动中的最后一个分支为止。注意，`IfElseActivity` 可以有任意多个分支，每个分支都有自己的条件。最后一个分支不能有条件，因为它相当于 If-Else 语句中的 `else` 部分。要添加一个新分支，可以显示活动的上下文菜单，从

菜单中选择 Add Branch 命令——也可以从 Visual Studio 的 Workflow 菜单中选择它。在添加分支时，每个分支都有一个强制的条件，但最后一个分支例外。

WF 定义两个标准的条件类型——CodeCondition 和 RuleConditionReference。CodeCondition 类在代码隐藏类上执行一个方法，它返回 true 或 false。要创建 CodeCondition，可以显示 IfElseActivity 的属性表，将条件设置为 CodeCondition，再为要执行的代码输入一个名称，如图 57-4 所示。

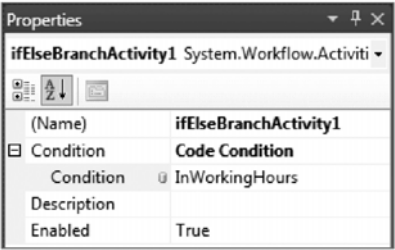


图 57-4

将方法名输入属性网格时，设计器会在代码隐藏类上构建一个方法，如下面的代码段所示：

```
private void InWorkingHours(object sender, ConditionalEventArgs e)
{
    int hour = DateTime.Now.Hour;
    e.Result = ((hour >= 9) && (hour <= 17));
}
```

如果当前时间在 9 am 和 5 pm 之间，上面的代码就将所传递的 ConditionalEventArgs 的 Result 属性设置为 true。可以在代码中定义条件，如上面的代码所示，另一种方法是根据以类似方式判断的 Rule 定义条件。Workflow 设计器包含一个规则编辑器，它可以用于声明条件和语句(类似于上面的 If-Else 语句)。这些规则在运行时根据工作流的当前状态进行判断。

57.2.2 ParallelActivity

这个活动可以定义并行执行的一组活动，或者以伪并行的方式执行的一组活动。当工作流运行库调度一个活动时，它在一个线程中调度活动。这个线程先执行第一个活动，再执行第二个活动，直到完成所有活动为止(或者某个活动在等待某种形式的输入为止)。在执行 ParallelActivity 时，它会迭代每个分支，依次调度执行每个分支。工作流运行库为每个工作流实例维护一个已调度的活动队列，一般以 FIFO(先进先出)的方式执行它们。

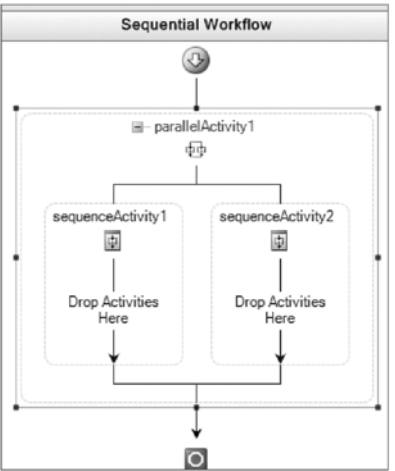


图 57-5

假定有如图 57-5 所示的一个 ParallelActivity，它调度执行 sequenceActivity1 和 sequenceActivity2。SequenceActivity 类型的工作方式是先用运行库调度执行第一个活动，这个活动执行完毕后，就调度第二个活动。这个调度/等待完成方法会遍历系列中的所有子活动，直到所有子活动都执行完毕后，序列活动才完成。

如果 SequenceActivity 一次调度执行一个活动，就表示 WorkflowRuntime 维护的队列会用可调度执行的活动连续不断地更新。假定有一个并行活动 P1，它包含两个系列(S1 和 S2)，每个系列都有两个代码活动(C1 和 C2)，这会在调度程序队列中生成如表 57-1 所示的项。

表 57-1

工作流队列	初始队列在队列中没有活动
P1	在工作流运行时并行执行
S1, S2	执行 P1 时, 添加到队列中
S2, S1.C1	S1 执行, 并将 S1.C1 添加到队列中
S1.C1, S2.C1	S2 执行, 并将 S2.C1 添加到队列中
S2.C1, S1.C2	S1.C1 完成, 所以 S1.C2 进入队列中
S1.C2, S2.C2	S2.C1 完成, 所以 S2.C2 进入队列中
S2.C2	队列中的最后一项

这里, 队列处理第一项(并行活动 P1), 这将序列活动 S1 和 S2 添加到工作流队列中。在执行序列活动 S1 时, 它会将其第一个子活动(S1.C1)放在队列的尾部, 调度并完成这个活动后, 就把第二个子活动添加到队列中。

从上面的例子可以看出, `ParallelActivity` 的执行并非真正是并行的, 它实际上在两个系列分支之间交叉执行。由此可以推断, 最好的情况是活动的执行时间最短, 因为每个工作流都只有一个线程为调度程序队列服务, 所以运行时间较长的活动会妨碍队列中其他活动的执行。但是, 因为活动的执行时间常常是任意的, 所以必须采用某种方式将活动标记为“运行时间较长”, 以便其他活动有机会执行。为此, 可以从 `Execute()`方法中返回 `ActivityExecutionStatus.Executing`, 在活动结束时让运行库知道, 以后还要调用它。这种活动的一个例子是 `DelayActivity`。

57.2.3 `CallExternalMethodActivity`

工作流一般需要调用工作流外部的的方法, 这个活动可以定义一个接口和在该接口上调用的方法。`WorkflowRuntime` 维护一个服务列表(其键为一个 `System.Type` 值), 使用传递给 `Execute()`方法的 `ActivityExecutionContext` 参数可以访问该服务列表。

可以定义自己的服务, 用于添加到这个集合中, 再从自己的活动中访问这些服务。例如, 可以构建一个数据访问层, 作为一个服务接口提供, 再为 `SQL Server` 和 `Oracle` 提供该服务的不同实现代码。因为活动只调用接口方法, 所以从 `SQL Server` 到 `Oracle` 的切换对活动是不透明的。

将一个 `CallExternalMethodActivity` 添加到工作流中后, 就定义两个强制属性(`InterfaceType` 和 `MethodName`)。接口类型定义在执行活动时运行库要使用的服务, 方法名指定要调用该接口的哪个方法。

在执行这个活动时, 它将查询该服务类型的执行上下文, 查找有指定接口的服务, 然后调用该接口上的相应方法。也可以从工作流中给方法传递参数, 对应内容将在 57.4.5 节中讨论。

57.2.4 `DelayActivity`

业务流程常常需要等待一段时间才能完成——考虑使用工作流进行费用申请的过程。工作流给直接经理发送一封电子邮件, 要求他批准某个费用申请。之后工作流进入等待状态, 等待经理批准(或者不批准), 这里最好定义一个超时期限, 这样如果在 1 天的时间内没有返回响应, 费用申请就路由给命令链中的下一个经理。

`DelayActivity` 可以实现这个情形的一部分(另一部分是下面定义的 `ListenActivity`),其任务是等待指定的时间,之后继续执行工作流。定义延迟的持续时间有两种方式:可以将延迟的 `TimeoutDuration` 属性设置为一个字符串,如“1.00:00:00”(1 天,没有指定小时、分钟和秒);也可以提供一个方法,在执行活动时,调用该方法,从代码中将延迟的持续时间设置为一个值。为此,需要为延迟活动的 `InitializeTimeoutDuration` 属性定义一个值。这会在代码隐藏中创建一个方法,如下面的代码段所示:

```
private void DefineTimeout(object sender, EventArgs e)
{
    DelayActivity delay = sender as DelayActivity;

    if (null != delay)
    {
        delay.TimeoutDuration = new TimeSpan(1, 0, 0, 0);
    }
}
```

这里的 `DefineTimeout()` 方法将发送者强制转换为一个 `DelayActivity`,然后在代码中把 `TimeoutDuration` 属性设置为 `TimeSpan`。尽管这里硬编码了这个值,但很可能从其他数据——或许是传递给工作流的一个参数或者从配置文件中读取的一个值——构建该属性值。工作流参数在 57.4 节中讨论。

57.2.5 ListenActivity

一个常见的编程结构是等待一组事件中某个可能的事件,例如 `System.Threading.WaitHandle` 类的 `WaitAny()` 方法。`ListenActivity` 是在工作流中等待事件的一种方式,因为它可以定义任意多个分支,每个分支都把基于事件的活动作为该分支的第一个活动。

事件活动是实现在 `System.Workflow.Activities` 名称空间中定义的 `IEventActivity` 接口的活动。目前 WF 将 3 个这样的活动定义为标准活动: `DelayActivity`、`HandleExternalEventActivity` 和 `WebServiceInputActivity`。图 57-6 中的工作流正在等待外部输入或者某个延迟——这是前面讨论的费用申请工作流的一个示例。

在这个例子中, `CallExternalMethodActivity` 用作工作流中的第一个活动,它会调用在服务接口上定义的方法,服务接口提示经理批准或不批准——因为这是一个外部服务,所以该提示可以是电子邮件、IM 消息或用其他方式通知经理,需要处理一个费用申请。之后,工作流执行 `ListenActivity`,等待来自这个外部服务的输入(批准或不批准),同时也在延迟。

在执行 `ListenActivity` 时,它实际上会将一个等待操作放在每个分支的第一个活动中,在触发一

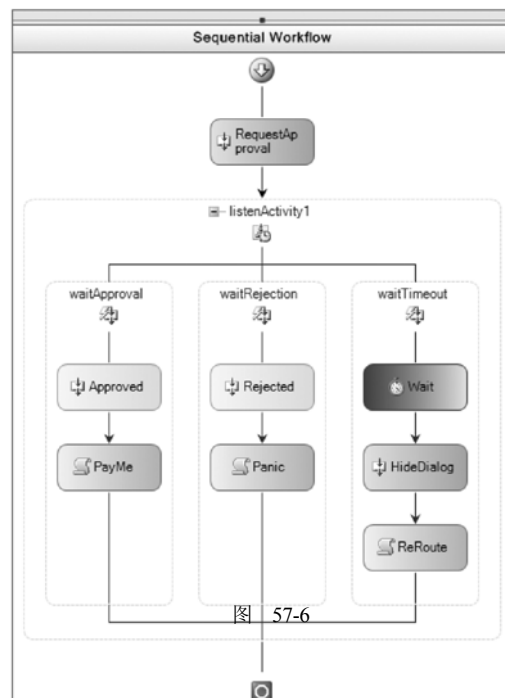


图 57-6

个事件时，会取消所有其他等待事件，然后处理在其中触发事件的分支的其他活动。所以在该情况中，如果批准费用报告，就引发 `Approved` 事件，然后调度 `PayMe` 活动。但如果经理没有批准该费用申请，就引发 `Rejected` 事件，然后调度 `Panic` 活动。

最后，如果 `Approved` 事件和 `Rejected` 事件都没有引发，`DelayActivity` 最终就会在延迟到期后完成，费用报告可以路由给下一个经理——可能在 `Active Directory` 中查找这个人。在本示例中，因为在执行 `RequestApproved` 活动时会给用户显示一个对话框，所以如果执行 `DelayActivity` 时，那么还需要关闭这个对话框，而这就是图 57-6 中 `HideDialog` 活动的作用。

这个例子的代码在 `02 Listen` 目录下。该示例使用了前面没有介绍的一些概念，如 workflow 实例如何标识，如何引发事件返回 workflow 运行库，最终发送给正确的工作流实例。这些概念将在 57.4 节中探讨。

57.2.6 活动执行模型

到目前为止，本章仅讨论了运行库通过调用 `Execute()` 方法执行活动。实际上，活动在执行过程中会经历许多不同的状态，如图 57-7 所示。

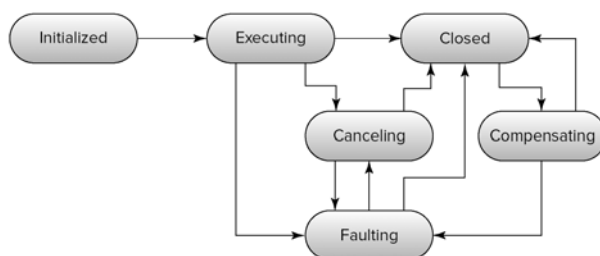


图 57-7

`WorkflowRuntime` 先调用活动的 `Initialize()` 方法，初始化这个活动。给这个方法传递 `IServiceProvider` 实例，该实例映射到运行库中可用的服务。这些服务将在 57.6 节中讨论。大多数活动在这个方法中都是什么都不做，但这个方法会进行一些必要的设置。

接着，运行库调用 `Execute()` 方法，活动就可以从 `ActivityExecutionStatus` 枚举中返回一个值。一般应从 `Execute()` 方法中返回 `Closed`，表示活动处理完毕；但是如果返回其他状态值中的某个值，运行库就使用该值确定活动所处的状态。

从这个方法中可以返回 `Executing`，表示运行库还有额外的工作要做。一个典型的例子是当有一个复合活动并且它需要执行其子活动时。在这种情况下，活动可以调度其每个子活动以便执行，接着等待所有子活动执行完毕，之后通知运行库，活动已完成。

57.3 自定义活动

前面使用的都是 `System.Workflow.Activities` 名称空间中定义的活动。本节将学习如何创建自定义活动，扩展它们，从而在设计时和运行时提供更好的用户体验。

首先，创建 `WriteLineActivity`，它用于一行文本输出到控制台上。这是一个简单的例子，但后面将扩展它，介绍使用该例子的自定义活动的所有可用选项。在创建自定义活动时，可以仅在工作流项目中构建一个类，但最好在一个独立的程序集中构建自定义活动，因为 `Visual Studio` 的设计环

境(尤其是和工作流项目)会从程序集中加载活动,并能在更新程序集时锁定它。所以,应创建一个简单的类库项目,在其中构建自定义活动。

简单的活动(如 `WriteLineActivity`)直接派生自 `Activity` 基类。下面的代码显示一个构建的活动类,并定义 `Message` 属性,在调用 `Execute()`方法时会显示该属性:

```
using System;
using System.ComponentModel;
using System.Workflow.ComponentModel;

namespace SimpleActivity
{
    /// <summary>
    /// A simple activity that displays a message to the console when it executes
    /// </summary>
    public class WriteLineActivity: Activity
    {
        /// <summary>
        /// Execute the activity-display the message on screen
        /// </summary>
        /// <param name="executionContext"></param>
        /// <returns></returns>
        protected override ActivityExecutionStatus Execute
            (ActivityExecutionContext executionContext)
        {
            Console.WriteLine(Message);

            return ActivityExecutionStatus.Closed;
        }

        /// <summary>
        /// Get/Set the message displayed to the user
        /// </summary>
        [Description("The message to display")]
        [Category("Parameters")]
        public string Message
        {
            get { return _message; }
            set { _message = value; }
        }

        /// <summary>
        /// Store the message displayed to the user
        /// </summary>
        private string _message;
    }
}
```

在 `Execute()`方法中,可以将消息写入控制台中,再返回 `Closed` 状态,通知运行库,活动已完成。

也可以在 `Message` 属性上定义特性,以便为这个属性定义其描述内容和类别。这将在 Visual Studio 的属性网格中使用,如图 57-8 所示。

本节用于创建活动的代码在 03 CustomActivities 解决方

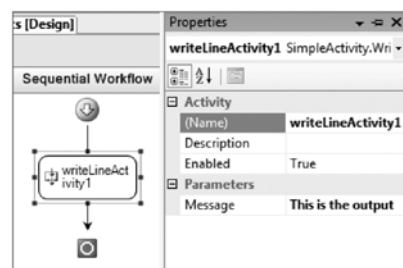


图 57-8

案中。如果编译该解决方案，就可以从工具箱的上下文菜单中选择 **Choose Items** 菜单项，导航到包含活动的程序集所驻留的文件夹，将自定义活动添加到 **Visual Studio** 的工具箱中。程序集中的所有活动都添加到工具箱中。

这个活动肯定是可以使用的，但是，有几个地方可以使这个活动更友好。与本章前面的 **CodeActivity** 一样，它有一些强制属性，如果没有定义，就会在设计界面上生成一个错误标志符号。为了从活动获得相同的行为，需要构建一个派生自 **ActivityValidator** 的类，将这个类与自定义活动关联起来。

57.3.1 活动的验证

当把活动放在设计界面上时，工作流设计器就会在该活动上查找一个属性，它定义对活动进行验证的类。为了验证活动，需要检查是否设置了 **Message** 属性。

给活动实例传递一个自定义验证器，以确定哪些强制属性没有定义，并给设计器使用的 **ValidationErrorsCollection** 添加一个错误。之后工作流设计器读取这个集合，在该集合中发现的错误会将一个标志符号添加到活动中，并将每个错误链接到需要注意的属性上。

```
using System;
using System.Workflow.ComponentModel.Compiler;

namespace SimpleActivity
{
    public class WriteLineValidator: ActivityValidator
    {
        public override ValidationErrorsCollection Validate
            (ValidationManager manager, object obj)
        {
            if (null == manager)
                throw new ArgumentNullException("manager");
            if (null == obj)
                throw new ArgumentNullException("obj");

            ValidationErrorsCollection errors = base.Validate(manager, obj);

            // Coerce to a WriteLineActivity
            WriteLineActivity act = obj as WriteLineActivity;

            if (null != act)
            {
                if (null != act.Parent)
                {
                    // Check the Message property
                    if (string.IsNullOrEmpty(act.Message))
                        errors.Add(ValidationError.GetNotSetValidationError("Message"));
                }
            }

            return errors;
        }
    }
}
```

更新活动的任一部分时，以及将活动拖放在设计界面上时，设计器都会调用 **Validate()** 方法。设计器调用 **Validate()** 方法时，会把活动传递为非类型化的 **obj** 参数。

在这个方法中，先验证传入的参数，再调用基类的 `Validate()` 方法得到一个 `ValidationErrorsCollection`。尽管这不是严格必需的，但如果从有许多需要验证的属性的活动中派生，调用基类方法就能确保也检查这些属性。

代码然后将传递的 `obj` 参数强制转换为 `WriteLineActivity` 实例，检查活动是否有父活动。这个测试是必需的，因为 `Validate()` 函数在编译活动的过程中调用(假定活动在一个工作流项目或活动库中)，此时并没有定义父活动。没有这个检查，实际上就不能构建包含活动的程序集和验证器。如果项目的类型是类库，就不需要这个额外的步骤。

最后一步是检查 `Message` 属性是否设置为一个值，而不是设置为空字符串——这使用 `ValidationError` 类的一个静态方法，它构造一个错误，说明没有定义属性。

为了给 `WriteLineActivity` 实例添加验证支持，最后一步是将 `ActivityValidation` 属性添加到活动中，如下面的代码段所示：

```
[ActivityValidator(typeof(WriteLineValidator))]  
public class WriteLineActivity: Activity  
{  
    .  
}
```

如果编译该应用程序，再将一个 `WriteLineActivity` 实例拖放在工作流上，就会看到如图 57-9 所示的验证错误；单击这个错误，就能在属性网格中查看这个属性。

如果给 `Message` 属性输入一些文本，就会删除验证错误，之后就可以编译并运行应用程序。

既然完成了活动的验证后，接下来就要更改活动的呈现行为，给该活动添加填充色。为此，需要定义 `ActivityDesigner` 类和 `ActivityDesignerTheme` 类，如下一节所述。

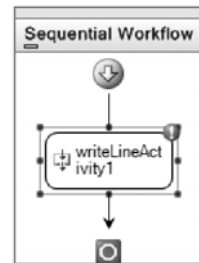


图 57-9

57.3.2 主题和设计器

活动的屏幕呈现使用 `ActivityDesigner` 类执行，这也可以使用 `ActivityDesignerTheme` 类执行。主题类用于在工作流设计器中对活动的呈现行为进行简单的更改。

```
public class WriteLineTheme: ActivityDesignerTheme  
{  
    /// <summary>  
    /// Construct the theme and set some defaults  
    /// </summary>  
    /// <param name="theme"></param>  
    public WriteLineTheme(WorkflowTheme theme)  
    : base(theme)  
    {  
        this.BackColorStart = Color.Yellow;  
        this.BackColorEnd = Color.Orange;  
        this.BackgroundStyle = LinearGradientMode.ForwardDiagonal;  
    }  
}
```

主题派生自 `ActivityDesignerTheme`，它有一个作为 `WorkflowTheme` 参数传递的构造函数。在该构造函数中，为活动设置起始颜色和结束颜色，再定义一个线性渐变画笔，用于绘制背景。

`Designer` 类用于重写活动的呈现行为——因为这里不需要重写，所以下面的代码就足够了：

```
[ActivityDesignerTheme(typeof(WriteLineTheme))]  
public class WriteLineDesigner: ActivityDesigner  
{  
}
```

注意使用 `ActivityDesignerTheme` 属性将主题与设计器关联起来。

最后一步是用 `Designer` 属性修饰活动：

```
[ActivityValidator(typeof(WriteLineValidator))]  
[Designer(typeof(WriteLineDesigner))]  
public class WriteLineActivity: Activity  
{  
    .  
}
```

于是，活动的呈现结果如图 57-10 所示。

添加设计器和主题之后，活动现在看上去更专业。主题上还有许多其他属性，如用于呈现边框的钢笔、边框的颜色、边框的样式等。

重写 `ActivityDesigner` 类的 `OnPaint()` 方法，可以完全控制活动的呈现。这里最好适量练习一下，因为可以更进一步，创建一个与工具箱中任何其他活动都不雷同的活动。

在 `ActivityDesigner` 类中，另一个有用的重写属性是 `Verbs`。它允许在活动的上下文菜单中添加菜单项。用 `ParallelActivity` 的设计器将 `Add Branch` 菜单项插入活动的上下文菜单和 `Workflow` 菜单中。还可以重写设计器的 `PreFilterProperties()` 方法，修改为活动提供的属性列表，这是 `CallExternalMethodActivity` 的方法参数显示到属性网格中的方式。如果需要对设计器进行这类扩展，就应运行 Red Gate 的 `Reflector`(<http://www.reflector.red-gate.com>)，并在其中加载工作程序集，查看 Microsoft 如何定义一些扩展属性。

这个活动快完成了，现在需要定义呈现活动时使用的图标，以及与活动关联的工具箱选项。

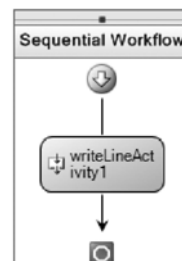


图 57-10

57.3.3 ActivityToolboxItem 和图标

为了完成自定义活动，需要添加一个图标。还可以创建一个派生自 `ActivityToolboxItem` 的类，在 Visual Studio 的工具箱中显示活动时，要用到这个类。

为了给活动定义图标，创建一幅 16×16 像素的图像。将它包含到项目中，之后将图像的构建操作设置为 `Embedded Resource`。这会将该图像包含到程序集的清单资源中。可以给项目添加一个 `Resources` 文件夹，如图 57-11 所示。

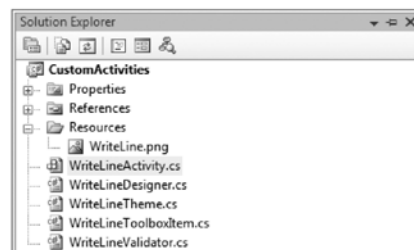


图 57-11

一旦添加了图像文件并将其构建操作设置为 `Embedded Resource` 后，就可以设置活动的属性，如下面的代码段所示：

```

        [ActivityValidator(typeof(WriteLineValidator))]
        [Designer(typeof(WriteLineDesigner))]
        [ToolboxBitmap(typeof(WriteLineActivity), "Resources.WriteLine.png")]
        public class WriteLineActivity: Activity
        {
            .
        }

```

`ToolboxBitmap` 属性定义许多构造函数,这里使用的构造函数的参数是活动程序集中定义的类型和资源名称。当将一个资源添加到文件夹中时,因为从程序集的名称空间和图像驻留的文件夹的名称构造其名称,所以该资源的完全限定名称是 `CustomActivities.Resources.WriteLine.png`。给 `ToolboxBitmap` 属性使用的构造函数将类型参数所驻留的名称空间追加到作为第二个参数传递的字符串后面,这样当 Visual Studio 加载它时,这个名称会解析为相应的资源。

最后一个需要创建的类派生自 `ActivityToolboxItem`。当把该活动加载到 Visual Studio 工具箱中时,会使用这个类。该类的一个典型应用是更改活动在工具箱上显示的名称——所有内置活动都会更改其名称,从类型中删除“Activity”。在这个类中,要将 `DisplayName` 属性设置为 `WriteLine`,以更改活动的名称。

```

        [Serializable]
        public class WriteLineToolboxItem: ActivityToolboxItem
        {
            /// <summary>
            /// Set the display name to WriteLine - i.e. trim off
            /// the 'Activity' string
            /// </summary>
            /// <param name="t"></param>
            public WriteLineToolboxItem(Type t)
                : base(t)
            {
                base.DisplayName = "WriteLine";
            }

            /// <summary>
            /// Necessary for the Visual Studio design time environment
            /// </summary>
            /// <param name="info"></param>
            /// <param name="context"></param>
            private WriteLineToolboxItem(SerializationInfo info,
                                         StreamingContext context)
            {
                this.Deserialize(info, context);
            }
        }

```

这个类派生自 `ActivityToolboxItem`,并重写构造函数,以更改显示名称;它还提供了一个序列化构造函数,当将选项加载到工具箱中时,工具箱会使用这个序列化构造函数。如果没有这个构造函数,在试图将活动添加到工具箱上时,就会接收到一个错误。注意这个类也标记为 `[Serializable]`。

使用 `ToolboxItem` 属性把工具箱中的选项添加到活动中,如下面的代码所示:

```

        [ActivityValidator(typeof(WriteLineValidator))]
        [Designer(typeof(WriteLineDesigner))]

```

```
[ToolboxBitmap(typeof(WriteLineActivity), "Resources.WriteLine.png")]
[ToolboxItem(typeof(WriteLineToolboxItem))]
public class WriteLineActivity: Activity
{
    .
}
}
```

所有这些更改都完成后，就可以编译程序集，然后创建一个新的 workflow 项目。要把活动添加到工具箱中，可以打开一个 workflow，然后显示工具箱的上下文菜单，并单击 **Choose Items** 按钮。

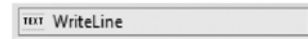


图 57-12

接着可以浏览包含活动的程序集，把活动添加到工具箱中，结果如图 57-12 所示。图标看起来不太漂亮，但它说明我们的工作已经完成了。

下一节将在自定义复合活动中再次访问 `ActivityToolboxItem`，因为这个类有一些额外的功能，将复合活动添加到设计界面上时，需要这些功能。

再次重申，我们创建了一个自定义 `WriteLineActivity`，创建了 `WriteLineValidator` 以添加验证逻辑，使用 `WriteLineDesigner` 类和 `WriteLineTheme` 类创建设计器和主题，为活动创建一个位图，最后创建了 `WriteLineToolboxItem` 类，以修改活动的显示名称。

57.3.4 自定义复合活动

活动有两种主要类型，派生自 `Activity` 的活动可以看作能从 workflow 中调用的函数。派生自 `CompositeActivity` 的活动(如 `ParallelActivity`、`IfElseActivity` 和 `ListenActivity`)是其他活动的容器，它们在设计期间的行为完全不同于简单的活动，因为它们显示在设计器的一个区域中，可以在该区域中拖放子活动。

本节将创建一个活动，称为 `DaysOfWeekActivity`。这个活动可以根据当前的日期执行 workflow 的不同部分。例如，在 workflow 中，为周末到达的订单执行的路径需要与正常工作日到达的订单的执行路径不同。在这个例子中，读者将学习许多高级 workflow 主题，在本节末尾之前将很好地理解如何用自己的复合活动扩展系统。这个例子的代码也放在 03 `CustomActivities` 解决方案中。

首先，创建一个自定义活动，它的一个属性默认为当前日期/时间。可以将该属性设置为另一个值，该值来自于 workflow 中的另一个活动；或把该属性设置为一个执行 workflow 时传递给 workflow 的一个参数。这个复合活动包含用户定义的许多分支，每个分支都包含一个枚举常量，该常量定义将执行该分支的日期。下面的示例定义该活动及其两个分支：

```
DaysOfWeekActivity
    SequenceActivity: Monday, Tuesday, Wednesday, Thursday, Friday
    <other activites as appropriate>
    SequenceActivity: Saturday, Sunday
    <other activites as appropriate>
```

这个例子需要一个定义一周中各天的枚举，其中包含 `[Flags]` 属性(这样就不能使用 `System` 名称空间中定义的内置枚举 `DayOfWeek`，因为它不包含 `[Flags]` 属性)。

```
[Flags]
[Editor(typeof(FlagsEnumEditor), typeof(UIDTypeEditor))]
public enum WeekdayEnum: byte
{
```

```

        None = 0x00,
        Sunday = 0x01,
        Monday = 0x02,
        Tuesday = 0x04,
        Wednesday = 0x08,
        Thursday = 0x10,
        Friday = 0x20,
        Saturday = 0x40
    }

```

这个类型还包含一个自定义编辑器，它允许根据复选框修改枚举值。其代码可以下载。

定义了枚举类型后，就可以考虑活动的主干了。自定义复合活动一般派生自 `CompositeActivity` 类，此外，因为该基类定义 `Activities` 属性，它是所有后续活动的集合。

```

public class DaysOfWeekActivity: CompositeActivity
{
    /// <summary>
    /// Get/Set the day of week property
    /// </summary>
    [Browsable(true)]
    [Category("Behavior")]
    [Description("Bind to a DateTime property, set a specific date time,
                  or leave blank for DateTime.Now")]
    [DefaultValue(typeof(DateTime), "")]
    public DateTime Date
    {
        get { return (DateTime)
                base.GetValue(DaysOfWeekActivity.DateProperty); }
        set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
    }

    /// <summary>
    /// Register the DayOfWeek property
    /// </summary>
    public static DependencyProperty DateProperty =
        DependencyProperty.Register("Date", typeof(DateTime),
            typeof(DaysOfWeekActivity));
}

```

`Date` 属性提供常规的 `get` 和 `set` 存取器，本例还添加了许多标准特性，以便 `Date` 属性能正确地显示在属性浏览器中。其代码看起来与正常的 .NET 属性有所不同，因为 `get` 和 `set` 存取器没有使用标准字段存储其值，而是使用 `DependencyProperty`。

`Activity` 类(之所以是这个类，因为它最终派生自 `Activity`)派生自 `DependencyObject` 类，它定义了一个键为 `DependencyProperty` 的值字典。这种使用 `get` 和 `set` 存取器间接得到的属性值由 WF 用于支持绑定，即把一个活动的属性链接到另一个活动的属性上。例如，我们常常要在代码中传递参数，有时是按值传递，有时是按引用传递。因为 WF 使用绑定将属性值链接在一起，所以本例在工作流上定义了一个 `DateTime` 属性，这个活动需要在运行期间绑定到该属性值上。本章的后面将列举一个绑定的例子。

如果构建这个活动，那么并没有做太多的工作；实际上它甚至不允许在该活动中拖放子活动，因为还没有为该活动定义 `Designer` 类。

1. 添加设计器

与本章前面的 `WriteLineActivity` 一样，每个活动都有一个关联的 `Designer` 类，`Designer` 类用于更改该活动在设计期间的行为。虽然 `WriteLineActivity` 中的 `Designer` 类是空的，但对于复合活动，需要重写两个方法，用于添加某些特殊的分支处理。

```
public class DaysOfWeekDesigner: ParallelActivityDesigner
{
    public override bool CanInsertActivities
        (HitTestInfo insertLocation, ReadOnlyCollection<Activity> activities)
    {
        foreach (Activity act in activities)
        {
            if (!(act is SequenceActivity))
                return false;
        }

        return base.CanInsertActivities(insertLocation, activitiesToInsert);
    }

    protected override CompositeActivity OnCreateNewBranch()
    {
        return new SequenceActivity();
    }
}
```

这个 `Designer` 类派生自 `ParallelActivityDesigner`，它在添加子活动时提供很好的设计时行为。如果所拖放的任何活动不是 `SequenceActivity`，就需要将 `CanInsertActivities` 重写为返回 `false`。如果所有活动的类型都正确，就可以调用基类方法，进一步检查自定义活动所允许的活动类型。

还需要重写 `OnCreateNewBranch()` 方法，它在用户选择 `Add Branch` 菜单项时调用。`Designer` 类使用 `[Designer]` 属性与该活动关联起来，如下面的代码所示：

```
[Designer(typeof(DaysOfWeekDesigner))]
public class DaysOfWeekActivity: CompositeActivity
{
}
```

设计期间的操作就快完成了，然而，还需给活动添加一个派生自 `ActivityToolboxItem` 的类，指定将该活动的一个实例从工具箱中拖出时会发生什么。默认行为是仅构造一个新活动，然而，在本例中还希望创建两个默认分支。下面的代码完整地显示了这个工具箱选项类：

```
[Serializable]
public class DaysOfWeekToolboxItem: ActivityToolboxItem
{
    public DaysOfWeekToolboxItem(Type t)
        : base(t)
    {
        this.DisplayName = "DaysOfWeek";
    }

    private DaysOfWeekToolboxItem(SerializationInfo info,
        StreamingContext context)
    {
    }
}
```

```

        this.Deserialize(info, context);
    }

    protected override IComponent[] CreateComponentsCore(IDesignerHost host)
    {
        CompositeActivity parent = new DaysOfWeekActivity();
        parent.Activities.Add(new SequenceActivity());
        parent.Activities.Add(new SequenceActivity());
        return new IComponent[] { parent };
    }
}

```

代码更改了活动的显示名称，实现了序列化构造函数，并重写了 `CreateComponentsCore()` 方法。

这个方法在拖放操作的最后调用，在该方法中构造 `DayOfWeekActivity` 的一个实例。在代码中还构造两个子序列活动，为活动的用户提供了更好的设计时体验。几个内置活动与 `DayOfWeekActivity` 有相同的作用，当把一个 `IfElseActivity` 实例拖放到设计界面上时，它的工具箱选项类也会添加两个分支。当将 `ParallelActivity` 实例添加到工作流中时，也会添加两个分支。

序列化构造函数和 `[Serializable]` 特性是所有派生自 `ActivityToolboxItem` 的类都需要的。

最后，将这个工具箱选项类与该活动关联起来：

```

[Designer(typeof(DaysOfWeekDesigner))]
[ToolboxItem(typeof(DaysOfWeekToolboxItem))]
public class DaysOfWeekActivity: CompositeActivity
{
}

```

之后，活动的 UI 就接近完成了，如图 57-13 所示。

现在需要在图 57-13 的每个系列活动上定义一个属性，以便用户能指定分支执行的日期。这在 Windows 工作流中有两种方式：可以创建 `SequenceActivity` 的一个子类，在该子类中定义该属性；也可以使用依赖属性的另一个功能——附加属性。

这里采用第二种方式，因为它不需要创建子类，但可以有效地扩展序列活动，而无需活动的源代码。

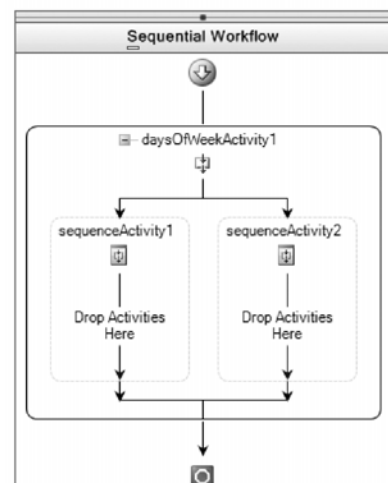


图 57-13

2. 附加属性

在注册依赖属性时，可以调用 `RegisterAttached()` 方法，创建一个附加属性。附加属性是在一个类中定义但在另一个类中显示的属性。所以这里在 `DayOfWeekActivity` 上定义一个属性，但这个属性实际上显示在 UI 上，作为系列活动的一个附加属性。

下面的代码段显示一个 `WeekdayEnum` 类型的 `Weekday` 属性，它会添加到驻留在复合活动中的序列活动中。

```

public static DependencyProperty WeekdayProperty =
    DependencyProperty.RegisterAttached("Weekday",
        typeof(WeekdayEnum), typeof(DaysOfWeekActivity),
        new PropertyMetadata(DependencyPropertyOptions.Metadata));

```

最后一行代码允许指定属性的额外信息，这个例子指定它是一个 `Metadata` 属性。

Metadata 属性与一般的属性不同，它只能在运行期间读取。Metadata 属性类似于 C# 中的常量声明。在程序正在执行时不能更改常量，在工作流正在执行时也不能更改 Metadata 属性。

在这个例子中，因为要指定执行活动的日期，所以在设计器中将这个字段设置为“Saturday, Sunday”。在工作流发出的代码中，应有如下的声明(代码已重新设置了格式，以适应页面的大小)：

```
this.sequenceActivity1.SetValue
    (DaysOfWeekActivity.WeekdayProperty,
     ((WeekdayEnum)((WeekdayEnum.Sunday | WeekdayEnum.Saturday))));
```

除了定义依赖属性之外，还需要利用方法在任意活动上获取和设置这个值。这些方法一般定义为复合活动上的静态方法，如下面的代码所示：

```
public static void SetWeekday(Activity activity, object value)
{
    if (null == activity)
        throw new ArgumentNullException("activity");
    if (null == value)
        throw new ArgumentNullException("value");

    activity.SetValue(DaysOfWeekActivity.WeekdayProperty, value);
}

public static object GetWeekday(Activity activity)
{
    if (null == activity)
        throw new ArgumentNullException("activity");

    return activity.GetValue(DaysOfWeekActivity.WeekdayProperty);
}
```

还需要更改另外两个地方，才能使这个额外的属性显示为 SequenceActivity 的附加属性。第一处更改是创建一个扩展提供程序，它告诉 Visual Studio 在序列活动中包含额外的属性。第二处更改是注册这个提供程序，具体操作是重写活动设计器的 Initialize() 方法，并添加如下代码：

```
protected override void Initialize(Activity activity)
{
    base.Initialize(activity);

    IExtenderListService iels = base.GetService(typeof(IExtenderListService))
        as IExtenderListService;

    if (null != iels)
    {
        bool extenderExists = false;

        foreach (IExtenderProvider provider in iels.GetExtenderProviders())
        {
            if (provider.GetType() == typeof(WeekdayExtenderProvider))
            {
                extenderExists = true;
                break;
            }
        }
        if (!extenderExists)
    }
```

```

        {
            IExtenderProviderService ieps =
                base.GetService(typeof(IExtenderProviderService))
                as IExtenderProviderService;
            if (null != ieps)
                ieps.AddExtenderProvider(new WeekdayExtenderProvider());
        }
    }
}

```

在上述代码中对 `GetService()` 方法的调用允许自定义设计器查询宿主(这里是 Visual Studio)设置的服务。在 Visual Studio 中查询 `IExtenderListService`，它提供用于枚举所有可用的扩展提供程序的一种方式。如果没有找到 `WeekdayExtenderProvider` 服务的实例，就查询 `IExtenderProviderService`，并添加一个新的提供程序。

扩展提供程序的代码如下所示：

```

[ProvideProperty("Weekday", typeof(SequenceActivity))]
public class WeekdayExtenderProvider: IExtenderProvider
{
    bool IExtenderProvider.CanExtend(object extendee)
    {
        bool canExtend = false;

        if ((this != extendee) && (extendee is SequenceActivity))
        {
            Activity parent = ((Activity)extendee).Parent;

            if (null != parent)
                canExtend = parent is DaysOfWeekActivity;
        }

        return canExtend;
    }

    public WeekdayEnum GetWeekday(Activity activity)
    {
        WeekdayEnum weekday = WeekdayEnum.None;

        Activity parent = activity.Parent;

        if ((null != parent) && (parent is DaysOfWeekActivity))
            weekday = (WeekdayEnum)DaysOfWeekActivity.GetWeekday(activity);

        return weekday;
    }

    public void SetWeekday(Activity activity, WeekdayEnum weekday)
    {
        Activity parent = activity.Parent;

        if ((null != parent) && (parent is DaysOfWeekActivity))
            DaysOfWeekActivity.SetWeekday(activity, weekday);
    }
}

```

扩展提供程序用它提供的属性标识，对于这些属性，它必须提供公共的 `Get<Property>()` 和

Set<Property>()方法。这些方法的名称必须匹配属性的名称，并带有相应的 Get 或 Set 前缀。

对设计器进行了上述更改，并添加扩展提供程序之后，在设计器中添加一个序列活动(这里是 SequenceActivity1)时，就会在 Visual Studio 中看到该属性，如图 57-14 所示。

扩展提供程序用于 .NET 中的其他功能，其中一个常见的功能是在 Windows 窗体项目中给控件添加工具提示。在窗体上添加工具提示控件时，会注册一个扩展器，为窗体上的每个控件添加一个 Tooltip 属性。

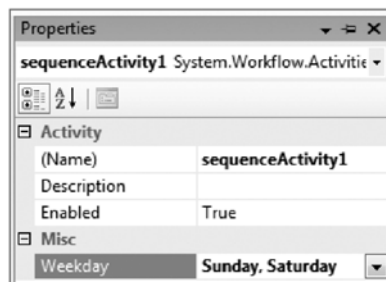


图 57-14

57.4 工作流

到目前为止，本章主要探讨了活动，但没有讨论工作流。工作流只是一个活动列表，实际上工作流本身是另一个类型的活动。使用这个模型简化运行库引擎，因为运行库引擎只需知道如何执行一种对象——派生自 Activity 类的对象。

每个工作流实例都用其 InstanceId 属性唯一地标识，InstanceId 属性是一个可以由运行库指定的 Guid，或者这个 Guid 可以由代码提供给运行库。Guid 的一个常见用途是将正在运行的工作流实例与在工作流外部维护的其他数据关联起来，如数据库中的一行。使用 WorkflowRuntime 类的 GetWorkflow(Guid)方法可以访问特定的工作流实例。

WF 中有两种工作流：顺序工作流和状态机工作流。

57.4.1 顺序工作流

顺序工作流中的根活动是 SequenceWorkflowActivity。这个类派生自前面介绍的 SequenceActivity，它定义两个根据需要可以附加处理程序的事件——Initialized 和 Completed。

顺序工作流首先执行其中的第一个子活动，之后执行第二个子活动，直到所有其他子活动都执行完毕为止。在两种情况下，工作流不会继续执行所有活动：一种是执行工作流的过程中引发了异常，另一种是工作流中存在 TerminateActivity。

工作流不会在所有情况下都执行。例如，当遇到 DelayActivity 时，工作流就进入等待状态，如果定义了工作流持续服务，就会从内存中删除工作流。工作流的持续性在 57.6.1 节中介绍。

57.4.2 状态机工作流

如果进程处于几种状态中的一种，只要给工作流传递数据，就可以使进程从一个状态转换到另一个状态，此时就可以使用状态机工作流。

一个例子是工作流用于对大厦的访问控制。此时，可以为一个 door 类建模，它可以关闭或打开，和一个 lock 类，它可以锁定或解锁。在最初启动系统(或大厦)时，就进入了一个已知状态——为了便于讨论，假定所有门都是关闭，且已上锁，那么某个门的状态是 closed locked。

当雇员从前门输入其访问代码时，会把一个事件发送给工作流，其中包含的细节有输入的代码和用户 ID。接着需要访问数据库，检索信息，例如，是否允许这个人在给定的这个时间打开选定的门，假定授权了该访问权限，工作流就会从其初始状态改为 closed unlocked 状态。

在这个状态下，有两种可能的结果——雇员打开该门(这是因为该门安装了打开/关闭传感器)；或者因为雇员发现把东西落在了汽车上，所以不打算进门了，于是，在过了延迟时间后要重新锁上门。因此该门返回 closed locked 状态，或进入 open unlocked 状态。

现在假定雇员进入大厦，并关上门——接着要从 open unlocked 状态进入 closed unlocked 状态，在过了延迟时间后要转换为 Closed locked 状态。如果该门处于 open unlocked 状态的时间很长，那么还要引发警报。

在 Windows Workflow 中为这种情形建模很简单：需要定义系统的状态，再定义可以使工作流从一个状态转换为另一个状态的事件。表 57-2 描述了系统的状态，并提供了每个已知状态可能的转换的细节，以及改变状态的输入(外部或内部输入)。

表 57-2

状 态	切 换
Closed Locked	这是系统的初始状态 为了响应用户的刷卡动作(且成功通过了访问检查)，状态会改为 Closed Unlocked，门锁会通过电子方式打开
Closed Unlocked	当门处于这种状态时，会发生如下两个事件中的一个： (1) 用户打开门——将状态转换为 Closed Locked 状态 (2) 定时器过期，门返回 Closed Locked 状态
Open Unlocked	在这个状态下，工作流只能转换为 Closed Unlocked 状态
Fire Alarm	这是工作流的最后一个状态，从其他 3 种状态都可以转换为这种状态

可以添加到系统中的另一个功能是响应火警的能力。当发出警报时，应打开所有门，让所有人离开大厦，消防员可以顺畅地进入大厦。可以把这个状态作为门工作流的最后一个状态建模，因为一旦取消火警，就可以从这个状态重置系统。

图 57-15 中的工作流定义这个状态机，并显示工作流可以处于的状态。线条说明系统中可以进行的转换。

工作流的初始状态用 ClosedLocked 活动建模。这包含一些初始化代码(锁上门)，然后是一个基于事件的活动，它等待一个外部事件，在这个例子中，雇员输入了大厦的访问代码。因为状态图中显示的每个活动都由序列工作流组成，所以为系统的初始化定义一个工作流(CLInitialize)，再定义一个 RequestEntry 工作流，该工作流会响应

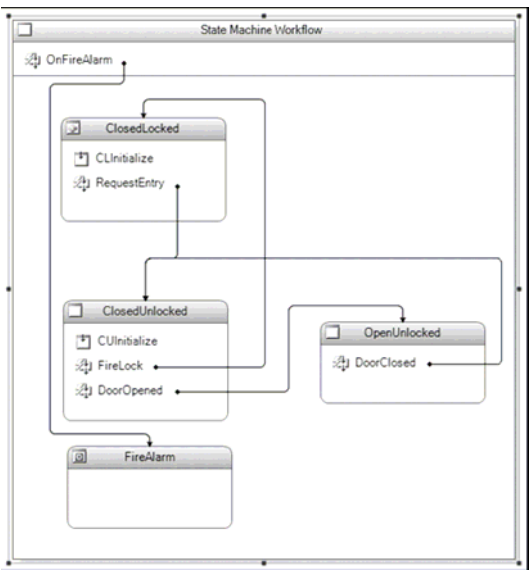


图 57-15

当雇员输入其 PIN 时引发的外部事件。如果查看 RequestEntry 工作流，它的定义就如图 57-16 所示。

每个状态都由许多子工作流组成，每个子工作流的开始都有一个事件驱动的活动，之后是任意多

个其他活动，它们构成了状态中的处理代码。在图 57-16 中，开头有一个 `HandleExternalEventActivity`，它在等待输入 PIN。之后工作流将检查 PIN，如果它是有效的，工作流就转换为 `ClosedUnlocked` 状态。

`ClosedUnlocked` 状态由两个工作流组成。一个工作流响应开门事件，将工作流转换为 `OpenUnlocked` 状态；另一个工作流包含一个延迟活动，该活动用于将状态改为 `ClosedLocked`。状态驱动的活动与本章前面的 `ListenActivity` 采用相同的方式工作——状态由许多事件驱动的工作流组成，当发生一个事件时，就执行其中一个工作流。

为了支持工作流，需要引发系统中的事件，实现状态的改变。为此，需要使用一个接口和该接口的实现代码，这对对象称为外部服务。本章后面将描述用于这个状态机的接口。

上述状态机例子的代码在 04 `StateMachine` 解决方案中，其中还包含一个用户界面，在该用户界面中可以输入 PIN，通过两扇门中的一扇门进入大厦。

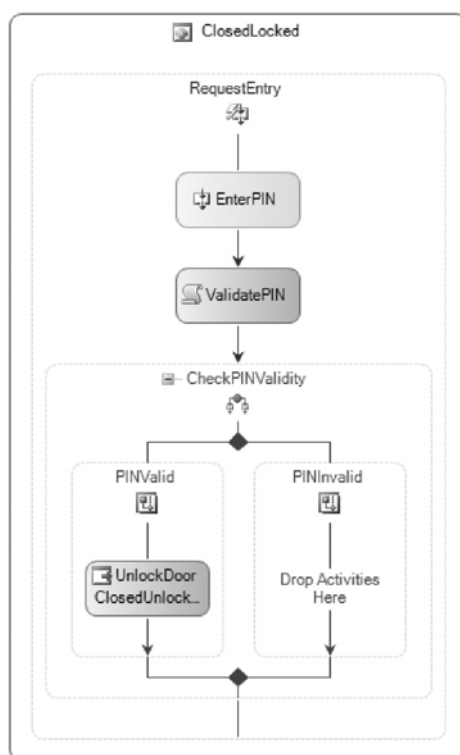


图 57-16

57.4.3 给工作流传递参数

工作流一般需要一些数据才能执行，例如，订单处理工作流需要一个订单 ID，付费处理工作流需要一个顾客账户 ID，或者其他必要的数项。

工作流的参数传递机制与标准的 .NET 类有所不同，在标准的 .NET 类中，一般在方法调用中传递参数。而对于工作流，传递参数时，要把这些参数存储在一个名称/值对字典中，在构造工作流时，要经过这个字典。

当 WF 调度工作流以便执行时，它使用这些名称-值对设置关于工作流实例的公共属性。每个参数名都要用工作流的公共属性检查。如果找到了匹配的属值，就调用属性设置器，把该参数的值

传递给设置器。如果将一个名称-值对添加到字典中，在该字典中，名称并不对应工作流上的一个属性，则在试图构造这个工作流时，会抛出一个异常。

例如，下面的工作流将 OrderID 属性定义为一个整数：

```
public class OrderProcessingWorkflow: SequentialWorkflowActivity
{
    public int OrderID
    {
        get { return _orderID; }
        set { _orderID = value; }
    }

    private int _orderID;
}
```

下面的代码说明了如何将订单 ID 参数传递给工作流的一个实例：

```
WorkflowRuntime runtime = new WorkflowRuntime ();

Dictionary<string,object> parms = new Dictionary<string,object>();
parms.Add("OrderID", 12345);

WorkflowInstance instance = runtime.CreateWorkflow(typeof(OrderProcessingWorkflow),
                                                    parms);

instance.Start();

. Other code
```

在上面的示例代码中，构造一个 Dictionary<string, object>，它包含要传递给工作流的参数，在构造工作流时要使用这个字典。上面的代码包含 WorkflowRuntime 类和 WorkflowInstance 类，这里没有介绍它们，57.8 节将介绍它们。

57.4.4 从工作流中返回结果

工作流的另一个常见要求是返回输出参数，它们可能用于将在数据库或其他永久存储器中记录数据。

因为工作流由工作流运行库执行，所以不能只使用标准的方法调用机制调用工作流，而需要创建一个工作流实例，启动这个实例，然后等待它的完成。工作流完成后，工作流运行库就会引发 WorkflowCompleted 事件。这是传递的关于工作流的上下文信息，该工作流已经完成，并包含从工作流中输出的数据。

所以，要获取从该工作流中返回的输出参数，需要将一个事件处理程序关联到 WorkflowCompleted 事件上，该处理程序可以从工作流中检索输出参数。下面的代码显示了如何实现上述操作的一个例子：

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
    AutoResetEvent waitHandle = new AutoResetEvent(false);
    workflowRuntime.WorkflowCompleted +=
        delegate(object sender, WorkflowCompletedEventArgs e)
        {
            waitHandle.Set();
            foreach (KeyValuePair<string, object> parm in e.OutputParameters)
            {
```

```

        Console.WriteLine("{0} = {1}", parm.Key, parm.Value);
    }
};

WorkflowInstance instance =
    workflowRuntime.CreateWorkflow(typeof(Workflow1));
instance.Start();

waitHandle.WaitOne();
}

```

我们把一个委托关联到 `WorkflowCompleted` 事件上, 在这个委托中, 迭代传递给该委托的 `WorkflowCompletedEventArgs` 类中的 `OutputParameters` 集合, 并在控制台上显示输出参数。这个集合包含工作流的所有公共属性。实际上工作流没有指定输出参数。

57.4.5 将参数绑定到活动上

既然理解了如何将参数传递给工作流, 就还需要明白如何把这些参数链接到活动上。这通过绑定机制实现。在前面定义的 `DaysOfWeekActivity` 中, 有一个 `Date` 属性, 它可以硬编码, 也可以绑定到工作流中的另一个值上。可绑定的属性显示在 Visual Studio 的属性网格中, 如图 57-17 所示。Data 属性名右边的图标表示, 这是一个可绑定的属性。

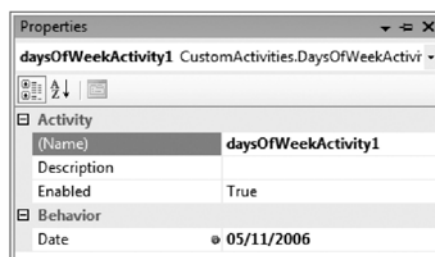


图 57-17

双击绑定图标会显示如图 57-18 所示的对话框。该对话框允许选择一个合适的属性, 链接到 `Date` 属性。

在图 57-18 中, 选择了工作流的 `OrderDate` 属性(把 `OrderData` 属性定义为工作流上一个普通的 .NET 属性)。任何 `Bindable` 属性都可以绑定到定义活动的工作流的属性上, 或者绑定到驻留在工作流中当前活动上方的任何活动的属性上。注意, 被绑定的属性的数据类型必须匹配正在绑定的属性的数据类型, 该对话框不允许绑定不匹配的类型。

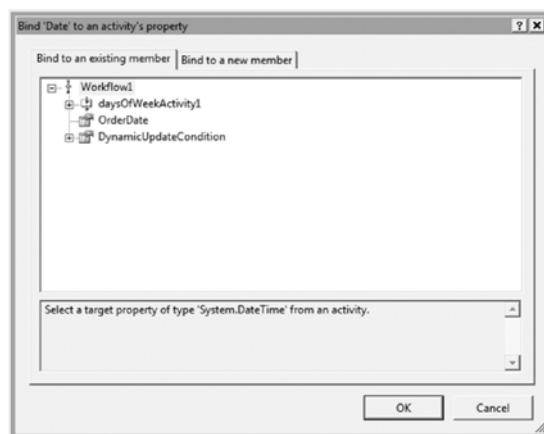


图 57-18

下面再次显示 `Date` 属性的代码, 说明绑定的工作方式, 并在后面几段中详细解释。

```
public DateTime Date
{
    get { return (DateTime)base.GetValue(DaysOfWeekActivity.DateProperty); }
    set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
}
```

当绑定工作流中的一个属性时，会在后台构造一个 `ActivityBind` 类型的对象，它就是存储在依赖属性中的值。所以，需要给属性设置器传递一个 `ActivityBind` 类型的对象，它存储在这个活动的属性字典中。这个 `ActivityBind` 对象包含的数据描述正在绑定的活动和要在运行时使用的活动的属性。

在读取属性值时，调用 `DependencyObject` 的 `GetValue()` 方法，这个方法会检查基础属性值，确定它是否是 `ActivityBind` 对象。如果是，它就解析这个绑定链接的活动，然后从该活动中读取属性值。但是，如果绑定的值是另一种类型，它就仅从 `GetValue()` 方法中返回该对象。

57.5 工作流运行库

为了启动工作流，需要创建 `WorkflowRuntime` 类的一个实例。这一般在应用程序中创建，这个对象通常定义为应用程序的一个静态成员，以便在应用程序的任意地方可以访问它。

在启动运行库时，它会从永久存储器中读取上次执行应用程序时执行的工作流实例，重新加载它们。这需要使用一个持久性服务，详见 57.6 节的内容。

运行库包含 6 个用于构造工作流实例的 `CreateWorkflow()` 方法。运行库还包含几个方法，可以重载工作流实例，并枚举所有正在运行的实例。

运行库还有许多在执行工作流时引发的事件，例如，`WorkflowCreated` (在构造新的工作流实例时引发)、`WorkflowIdled` (在工作流等待输入时引发，如前面的费用处理例子) 和 `WorkflowCompleted` (在工作流完成时引发)。

57.6 工作流服务

工作流不能独自存在，如前一节所述，工作流在 `WorkflowRuntime` 中执行，这个运行库提供了运行工作流的服务。

该服务可以是执行工作流时需要的任何类。运行库为工作流提供了一些标准服务，用户也可以选择构造自己的服务，由正在运行的工作流使用这些服务。

本节将描述运行库提供的两个标准服务，再说明如何构建自己的服务和何时需要一些实例。在活动运行时，要通过 `Execute()` 方法的 `ActivityExecutionStatus` 参数给活动传递一些上下文信息。

```
protected override ActivityExecutionStatus Execute
    (ActivityExecutionContext executionContext)
{
    .
}
```

在这个上下文参数上其中一个可用的方法是 `GetService<T>()`。在下面的代码中，它用于访问与工作流运行库关联的一个服务：

```
protected override ActivityExecutionStatus Execute
(ActivityExecutionContext executionContext)
{
    ICustomService myService = executionContext.GetService<ICustomService>();
    . Do something with the service
}
```

在调用 `StartRuntime()` 方法之前，运行库驻留的服务会添加到运行库中。如果试图将某个服务添加到已启动的运行库中，就会引发一个异常。

两个方法可用于将服务添加到运行库中：可以在代码中构造服务，再调用 `AddService()` 方法将它们添加到运行库中；也可以在应用程序配置文件中定义服务，构造这些服务并把它们添加到运行库中。

下面的代码段说明了如何在代码中将服务添加到运行库中——所添加的服务将在本节的后面介绍。

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
    workflowRuntime.AddService(
        new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                           new TimeSpan(0,10,0)));
    workflowRuntime.AddService(new SqlTrackingService(conn));
}
```

这段代码构造 `SqlWorkflowPersistenceService` 的实例，运行库使用这些实例存储工作流的状态。这段代码还构造 `SqlTrackingService` 的一个实例，它记录工作流运行时执行的事件。

要使用应用程序配置文件创建服务，需要为工作流运行库添加一个节处理程序，然后将服务添加到该节中，如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="WF"
      type="System.Workflow.Runtime.Configuration.WorkflowRuntimeSection,
      System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
  </configSections>

  <WF Name="Hosting">
    <CommonParameters/>
    <Services>
      <add type="System.Workflow.Runtime.Hosting.SqlWorkflowPersistenceService,
        System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"
        connectionString="Initial Catalog=WF;Data Source=.;
          Integrated Security=SSPI;"
        UnloadOnIdle="true"
        LoadIntervalSeconds="2"/>
      <add type="System.Workflow.Runtime.Tracking.SqlTrackingService,
        System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"
        connectionString="Initial Catalog=WF;Data Source=.;
          Integrated Security=SSPI;"
```

```

        UseDefaultProfile="true"/>
    </Services>
</WF>
</configuration>

```

在配置文件中，添加了 WF 节处理程序(这个名称并不重要，但必须匹配其后配置节的名称)，之后为该节创建合适的数据项。<Services>元素可以包含一个任意数据项列表，其中包含一个.NET 类型和运行库构造服务时传递给服务的参数。

要从应用程序配置文件中读取配置设置，可以调用运行库上的另一个构造函数，如下所示：

```

using(WorkflowRuntime workflowRuntime = new WorkflowRuntime("WF"))
{
    .
}

```

这个构造函数会实例化在配置文件中定义的服务，并把它们添加到运行库上的服务集合中。

下面几节将介绍 WF 提供的一些标准服务。

57.6.1 持久性服务

执行工作流时，它可能会进入等待状态——在执行延迟活动时，或者在 ListenActivity 中等待外部输入时，就会进入等待状态。此时，工作流处于空闲状态，这是持久性服务的一个候选对象。

假定开始在服务器上执行 1000 个工作流，之后每个工作流实例都进入空闲状态。此时，因为不需要在内存中维护这些实例的数据，所以最好能卸载工作流，释放它使用的资源。持久性服务就用于完成这个任务。

当某个工作流进入空闲状态时，工作流运行库会检查是否存在一个派生自 Workflow PersistenceService 类的服务。如果该服务存在，就给它传递工作流实例，接着服务就可以捕获工作流的当前状态，并将它存储在永久存储介质上。可以把工作流的状态存储在磁盘的一个文件中，或把这些数据存储在数据库(如 SQL Server)中。

工作流库包含持久性服务的实现代码，持久性服务可以把数据存储在 SQL Server 数据库中，即 SqlWorkflowPersistenceService。为了使用这个服务，需要对 SQL Server 实例运行两个脚本，其中一个脚本构造对应架构，另一个脚本创建持久性服务使用的存储过程。这些脚本默认位于 C:\Windows\Microsoft.NET\Framework\v3.5\Windows Workflow Foundation\SQL\EN 目录下。

在数据库上执行的脚本是 SqlPersistenceServiceProviderSchema.sql 和 SqlPersistenceProvider_Logic.sql。这些脚本需要按顺序执行：先执行架构文件，再执行逻辑文件。SQL 持久性服务的架构包含两个表：InstanceState 和 CompletedScope。这些都是不透明的表，不在 SQL 持久性服务的外部使用。

当工作流空闲时，其状态用二进制序列化机制序列化，接着把这些数据插入 InstanceState 表中。重新激活工作流时，从这一行中读取状态，用于重新构建工作流实例。这一行用工作流实例 ID 作为键，一旦工作流完成就从数据库中删除该行。

SQL 持久性服务可以由多个运行库同时使用——它实现锁定机制，以便一次只能由工作流运行库的一个实例访问一个工作流。如果多个服务器都使用同一个永久存储器运行工作流，这个锁定行为就非常有用。

为了查看添加到永久存储器中的内容，需要构造一个新的工作流项目，并给运行库添加

SqlWorkflowPersistenceService 的一个实例。下面是使用声明性代码的一个例子：

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
    workflowRuntime.AddService(
        new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                           new TimeSpan(0,10,0)));
    // Execute a workflow here.
}
```

接着，如果构造一个包含 DelayActivity 的工作流，并将延迟时间设置为 10 秒，就可以查看在 InstanceState 表中存储的数据。05 WorkflowPersistence 示例包含上述代码，并在 20 秒的时间内执行一个延迟。

持久性服务的构造函数的参数如表 57-3 所示。

表 57-3

参 数	说 明	默 认 值
ConnectionString	持久性服务使用的数据库连接字符串	None
UnloadOnIdle	确定工作流在空闲时是否卸载它。它应总是设置为 true，否则就不会出现任何持久性	False
InstanceOwnershipDuration	定义已加载工作流的运行库拥有工作流实例的时间长度	None
LoadingInterval	在给数据库请求更新的永久性记录时使用的时间间隔	2 分钟

这些值也可以在配置文件中定义。

57.6.2 跟踪服务

当工作流执行时，它可能需要记录运行了哪些活动，对于 IfElseActivity 和 ListenActivity 等复合活动，执行其分支。这些数据可以用作工作流实例的一种审计线索，在以后的某个日期查看，证明执行了哪些活动，在工作流中使用了什么数据。跟踪服务可以用于这类记录操作，并可以配置，根据需要记录关于正在运行的工作流的尽可能少或尽可能多的信息。

在 WF 中，因为跟踪服务作为一个抽象类 TrackingService 实现，所以很容易用自己的跟踪服务替换标准的跟踪实现方式。在工作程序集中，跟踪服务有一个具体的实现方式，即 SqlTrackingService。

要记录工作流的状态数据，就需要定义一个 TrackingProfile。它定义应记录什么事件，例如，可以只记录工作流的开头和结束，忽略正在运行的实例的所有其他数据。更一般的情况是，记录工作流的所有事件和其中的每个活动，提供工作流的执行配置文件的完整描述。

运行库引擎调度工作流时，引擎会检查工作流跟踪服务是否存在。如果找到了一个跟踪服务，它就会要求该服务为正在执行的工作流提供一个跟踪配置文件，接着使用它记录工作流和活动数据。还可以定义用户跟踪数据，把它们存储在跟踪数据存储库中，这些操作不需要改变该架构。

跟踪配置文件类如图 57-19 所示。这个类包含活动、用户和工作流跟踪点的集合属性。跟踪点是一个对象(如 WorkflowTrackPoint)，它一般定义

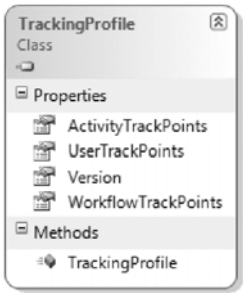


图 57-19

一个匹配位置和一些额外的数据，在单击这个跟踪点时，就会记录这些额外的数据。匹配位置指定这个跟踪点在什么地方有效。例如，可以定义一个 `WorkflowTrackPoint`，它记录了创建工作流时的一些数据，再定义一个 `WorkflowTrackPoint`，记录完成工作流时的一些数据。

一旦记录这些数据，就需要显示工作流的执行路径，如图 57-20 所示。该图显示了执行的工作流，运行的每个活动都包含一个说明它已执行的标志符号。这些数据从工作流实例的跟踪存储器中读取。

为了读取 `SqlTrackingService` 存储的数据，可以直接在 SQL 数据库上执行查询。Microsoft 还为此在 `System.Workflow.Runtime.Tracking` 名称空间中提供了 `SqlTrackingQuery` 类。下面的示例说明了如何检索在两个日期之间跟踪的所有工作流的一个列表：

```
public IList<SqlTrackingWorkflowInstance> GetWorkflows
(DateTime startDate, DateTime endDate, string connectionString)
{
    SqlTrackingQuery query = new SqlTrackingQuery (connectionString);

    SqlTrackingQueryOptions queryOptions = new SqlTrackingQueryOptions();
    query.StatusMinDateTime = startDate;
    query.StatusMaxDateTime = endDate;

    return (query.GetWorkflows (queryOptions));
}
```

这段代码使用 `SqlTrackingQueryOptions` 类，该类定义对应的查询参数。可以定义这个类的其他属性，进一步约束要检索的工作流。

在图 57-20 中，可以看到所有活动都执行了。但如果工作流仍在运行，或者在工作流中做了一些决策，以便在执行过程中采用不同的路径，情况就有所改变。跟踪数据包含执行了哪些活动等信息，这些数据与生成图 57-20 中的映像的活动相关。还可以在工作流执行时检索这些数据，用于为工作流的执行流生成审计线索。

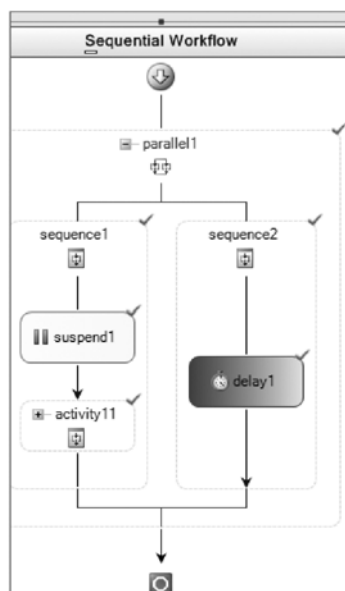


图 57-20

57.6.3 自定义服务

除了持久性服务和跟踪服务等内置服务之外，还可以在 `WorkflowRuntime` 维护的服务集合中添加自己的对象。这些服务一般用一个接口和一种实现方式定义，以便在不记录工作流的情况下替换该服务。

本章前面的状态机利用了下面的接口：

```
[ExternalDataExchange]
public interface IDoorService
{
    void LockDoor();
    void UnlockDoor();

    event EventHandler<ExternalDataEventArgs> RequestEntry;
    event EventHandler<ExternalDataEventArgs> OpenDoor;
    event EventHandler<ExternalDataEventArgs> CloseDoor;
    event EventHandler<ExternalDataEventArgs> FireAlarm;

    void OnRequestEntry(Guid id);
    void OnOpenDoor(Guid id);
    void OnCloseDoor(Guid id);
    void OnFireAlarm();
}
```

工作流使用组成这个接口的方法调用该服务，该服务引发的事件由工作流使用。使用 `ExternalDataExchange` 属性是告诉工作流运行库，这个接口用于正在运行的工作流和服务实现之间的通信。

在状态机中，有许多 `CallExternalMethodActivity` 的实例，它们用于在这个外部接口上调用方法。例如，当门上锁或不上锁时，工作流需要执行对 `UnlockDoor()` 或 `LockDoor()` 方法的方法调用，服务的响应是给门锁发送一条命令，锁上门，或打开门。

当服务需要与工作流通信时，应使用一个事件完成该任务，因为工作流运行库也包含一个 `ExternalDataExchangeService` 服务，它作为这些事件的代理。这个代理在引发事件时使用，因为在传递事件时，工作流可能没有加载到内存中，所以该事件会先路由到外部数据交换服务，它检查工作流是否已加载，如果没有加载，就从永久存储器中解除冻结工作流，再把该事件传递给工作流。

下面的代码构造 `ExternalDataExchangeService`，还为该服务定义的事件构造多个代理：

```
WorkflowRuntime runtime = new WorkflowRuntime();
ExternalDataExchangeService edes = new ExternalDataExchangeService();

runtime.AddService(edes);
DoorService service = new DoorService();
edes.AddService(service);
```

这段代码构造外部数据交换服务的一个实例，把它添加到运行库中。接着它创建 `DoorService` 的一个实例(它本身实现 `IDoorService`)，并把它添加到外部数据交换服务中。

`ExternalDataExchangeService.Add()` 方法为自定义服务定义的每个事件构造一个代理，以便在传递事件之前加载已持久化的工作流。如果没有把服务驻留在外部数据交换服务中，在引发事件时，就不会侦听这些事件，它们也不会传递给正确的工作流。

事件使用 `ExternalDataEventArgs` 类，因为它包含待传递事件的工作流实例 ID。如果需要将其他值从外部事件传递给工作流，就应从 `ExternalDataEventArgs` 中派生一个类，并把这些值作为属性添加到该类中。

57.7 与 WCF 集成

从 .NET 3.5 开始可用两个新活动支持工作流和 WCF 之间的集成，这两个活动是 `SendActivity` 和 `ReceiveActivity`。`SendActivity` 称为 `CallActivity` 更恰当，因为它的工作是向 WCF 服务发出一个请求，并可以选择性地把结果显示为参数，这些参数可以绑定到主调工作流中。

但更有趣的是 `ReceiveActivity`。因为它允许工作流变成 WCF 服务的实现方式，所以现在工作流就是服务。下面例子提供一个使用一个工作流的服务，并且也使用一个新的服务测试宿主工具，测试服务，而无需编写独立的测试工具。

从 Visual Studio 2010 的 New Project 菜单中选择 WCF 节点，再选择 Sequential Workflow Service Library 项，如图 57-21 所示。

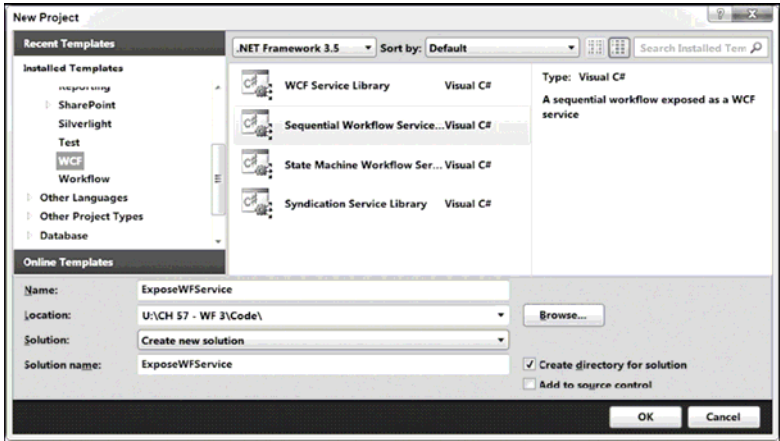


图 57-21

这会创建一个库，其中包含了一个工作流、一个应用程序配置文件和一个服务接口，如图 57-22 所示。这个示例的代码在 06 `ExposeWFSERVICE` 子目录中。

工作流提供协定的 `Hello` 操作，还定义要传递给这个操作的参数属性，以及该操作的返回值。接着只需添加代码，代码提供服务的执行行为，该服务就完成了。

为此，对于本示例，把一个 `CodeActivity` 拖放到 `ReceiveActivity` 上，如图 57-23 所示。再双击该活动，提供服务的实现代码。

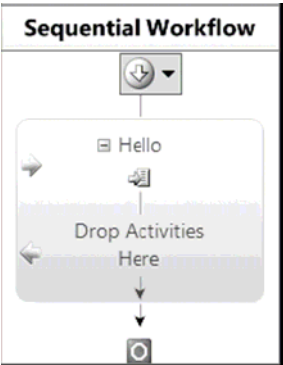


图 57-22

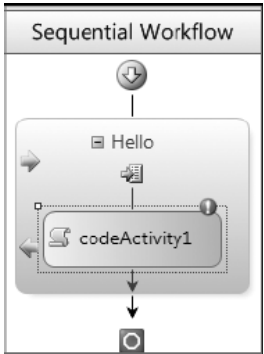


图 57-23

下面就是这个服务的实现代码。

```
public sealed partial class Workflow1: SequentialWorkflowActivity
{
    public Workflow1()
    {
        InitializeComponent();
    }

    public String returnValue = default(System.String);
    public String inputMessage = default(System.String);

    private void codeActivity1_ExecuteCode(object sender, EventArgs e)
    {
        this.returnValue = string.Format("You said {0}", inputMessage);
    }
}
```

因为 Hello 操作的服务协定包含 `inputMessage` 参数和一个返回值，所以它们都作为公共字段提供给工作流。在代码中，把 `returnValue` 设置为一个字符串值，这就是从 WCF 服务的调用中返回的内容。

如果编译这个服务，按 F5 键，就会注意到 Visual Studio 2010 的另一个新功能：WCF 测试客户端应用程序，如图 57-24 所示。

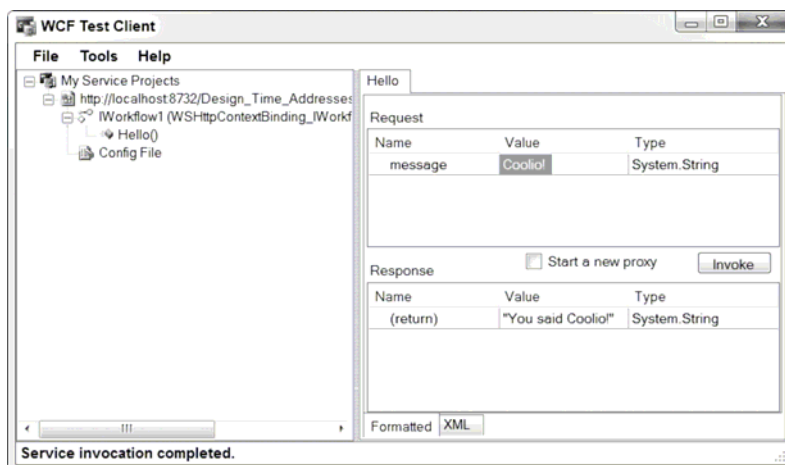


图 57-24

在这里可以浏览服务提供的操作，双击一个操作，会显示窗口的右半部分，其中列出了该服务使用的参数和提供的任意返回值。

要测试该服务，可以给 `message` 属性输入一个值，单击 `Invoke` 按钮。这会通过 WCF 给该服务发送一个请求，构造并执行工作流，调用代码活动，代码活动然后运行代码隐藏，最终给 WCF 测试客户端返回工作流的结果。

如果要手动作为服务驻留工作流，那么可以使用在 `System.WorkflowServices` 名称空间中定义的新 `WorkflowServiceHost` 类。下面的代码段显示了一个非常小的宿主的实现方式：

```
using (WorkflowServiceHost host = new WorkflowServiceHost
```

```

                                (typeof(YourWorkflow)))
{
    host.Open();
    Console.WriteLine ( "Press [Enter] to exit" );
    Console.ReadLine();
}

```

这里构造 `WorkflowServiceHost` 的一个实例，把它传递给要执行的工作流。这类似于驻留 WCF 服务时使用 `ServiceHost` 类的方式。它将读取配置文件，确定服务应侦听哪个端点，然后等待服务请求。

下一节介绍驻留工作流的某些其他选项。

57.8 驻留工作流

在进程中驻留 `WorkflowRuntime` 的代码随应用程序的不同而不同。

对于 Windows 窗体应用程序或 Windows 服务，一般在应用程序的开头构造运行库，把它存储在主应用程序类的一个属性中。

为了响应应用程序中的一些输入(如用户单击了用户界面上的一个按钮)，可能就需要构造工作流的一个实例，在本地执行这个实例。工作流可能还需要与用户通信，例如，可以定义一个外部服务，在将订单发送给后端服务器之前，提示用户确认。

在 ASP.NET 中驻留工作流时，一般不用消息框提示用户，而是导航到站点上请求确认的另一个页面上，再显示一个确认页面。在 ASP.NET 中驻留运行库时，一般要重写 `Application_Start` 事件，在那里构造工作流运行库的一个实例，以便在站点的所有其他部分访问它。运行库实例可以存储在一个静态属性中，但最好把它存储在应用程序状态中，再提供一个访问方法，从应用程序状态中检索工作流运行库，使之可用于应用程序的其他地方。

在 Windows 窗体或 ASP.NET 中，都要构造工作流运行库的一个实例，给它添加服务，如下所示：

```

WorkflowRuntime workflowRuntime = new WorkflowRuntime();

workflowRuntime.AddService(
    new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                     new TimeSpan(0,10,0)));

// Execute a workflow here.

```

要执行工作流，需要使用运行库的 `CreateInstance()` 方法，创建该工作流的一个实例。这个方法有许多重写版本，重写版本可用于构造基于代码的工作流的实例或在 XML 中定义的工作流的实例。

本章前面都把工作流看作 .NET 类，实际上这只是工作流的一种表示形式。还可以使用 XML 定义工作流，此时运行库会构造工作流在内存中的表示形式，在调用 `WorkflowInstance` 的 `Start()` 方法时执行它。

在 Visual Studio 中，可以从 Add New Item 对话框中选择 `Sequential Workflow`(其代码是独立的)或 `State Machine Workflow`(其代码是独立的)。这将创建基于 XML 的工作流。这会创建一个扩展名为 .xoml 的 XML 文件，并将它加载到设计器中。

把活动添加到设计器中时，会将这些活动持久化到 XML 中，元素的结构定义活动之间的父子关系。下面的 XML 是一个简单的序列工作流，其中包含一个 `IfElseActivity` 和两个代码活动，分别

用于 `IfElseActivity` 的每个分支。

```
<SequentialWorkflowActivity x:Class="DoorsWorkflow.Workflow1" x:Name="Workflow1"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
    <IfElseActivity x:Name="ifElseActivity1">
        <IfElseBranchActivity x:Name="ifElseBranchActivity1">
            <IfElseBranchActivity.Condition>
                <CodeCondition Condition="Test" />
            </IfElseBranchActivity.Condition>
            <CodeActivity x:Name="codeActivity1" ExecuteCode="DoSomething" />
        </IfElseBranchActivity>
        <IfElseBranchActivity x:Name="ifElseBranchActivity2">
            <CodeActivity x:Name="codeActivity2" ExecuteCode="DoSomethingElse" />
        </IfElseBranchActivity>
    </IfElseActivity>
</SequentialWorkflowActivity>
```

在活动上定义的特性作为属性持久化到 XML 中，每个活动都作为一个元素持久化。从 XML 可以看出，该结构定义了父活动(如 `SequentialWorkflowActivity` 和 `IfElseActivity`)和子活动之间的关系。

执行基于 XML 的工作流与执行基于代码的工作流没有区别，只需使用 `CreateWorkflow()` 方法的一个重写版本，该版本接受一个 `XMLReader` 实例，接着调用 `Start()` 方法启动该实例即可。

与基于代码的工作流相比，使用基于 XML 的工作流的一个优点是：很容易将工作流的定义存储在数据库中。之后可以在运行时加载这个 XML，执行工作流。修改该工作流定义时，也不需要重新编译代码。

无论工作流是在 XML 中定义，还是在代码中定义，都可以在运行时更改工作流。只需构造一个 `WorkflowChanges` 对象，它包含要添加到工作流中的所有新活动，接着调用在 `WorkflowInstance` 类上定义的 `ApplyWorkflowChanges()` 方法，持久化这些更改。这非常有用，因为业务需求时常更改，例如，需要把这些更改应用于一个保险单工作流，在续保日期的一个月之前给客户发送电子邮件，告诉客户他们的保险单需要续保。因为这种更改要基于每个实例进行，所以如果系统中有 100 个保险单工作流，就需要对每个工作流进行这种更改。

57.9 工作流设计器

本章还有最后一个主题要讨论。用于设计工作流的工作流设计器不与 Visual Studio 关联，可以根据需要自己的应用程序中重新驻留这个设计器。

这意味着，可以发布一个包含工作流的系统，允许最终用户在不需要 Visual Studio 的副本的情况下定制系统。但驻留设计器相当复杂，这个主题可能需要好几章的篇幅，但这超出了本章的范围。在 Web 上有许多重新驻留设计器的例子，建议读者在 <http://msdn2.microsoft.com/enus/library/aa480213.aspx> 上获得驻留设计器的更多信息。

允许用户定制系统的传统方式是定义一个接口，再让客户实现这个接口，根据需要扩展处理过程。

有了 Windows 工作流，该扩展在整体上变成更多图形，因为可以给用户提供一个空白的工作流作为模板，再提供一个工具箱，其中包含适合于应用程序的自定义活动。他们接着可以编写自己的工作流，添加他们自己编写的自定义活动。

57.10 从 WF 3.X 迁移到 WF 4

在 .NET 4 和 Visual Studio .NET 2010 中有一个 WF 新版本，这个新版本与 3.x 不后向兼容。概念大都相同，但实现方式完全不同。其优点是可以继续运行 3.x 工作流，无需任何修改；但如果希望使用新功能，就必须迁移应用程序。本节介绍这个迁移过程的要点，并提供一些简化这个过程的建议。



WF4 的更多信息可参见第 44 章。

57.10.1 把活动代码提取到服务中

因为 WF 4 的类层次结构与 WF 3.x 大不相同，但活动类大都相同；所以为了简化升级过程，建议从活动中删除内联代码，并把它们放在一组服务中(如包含静态方法的简单 .NET 类)。

因为数据绑定的本质在 WF 4 中有了很大的变化，所以在 WF 3.x 中可以使用依赖属性和/或标准的 .NET 属性，而在 WF 4 中需要使用参数对象。迁移代码最简单的方法是移动活动内部的所有处理过程，并把它们移动到一个单独的类中。例如，考虑 WriteLine 活动的代码段：

```
public class WriteLineActivity : Activity
{
    public string Message { get; set; }

    public override ActivityExecutionStatus Execute
    ( ActivityExecutionContext context )
    {
        Console.WriteLine ( Message );
        return ActivityExecutionStatus.Closed;
    }
}
```

如果要把这些代码转换为 WF 4 活动，就需要创建如下类：

```
public static class WriteLineService
{
    public static void WriteLine ( string message )
    {
        Console.WriteLine ( message );
    }
}
```

接着就可以在当前活动中使用这个类(还可以进行独立的单元测试)，当升级到 WF 4 时，要编写的代码会比较少。尽管这是一个简单的例子，但它说明了如何简化升级过程。

57.10.2 删除代码活动

由于 Workflow 4 中活动的代码隐藏样式与 3.x 中已有的样式不同，因此，如果在代码隐藏文件中包含代码，就应使用前面建议的方法把这些代码移植到一个库中，以便在 WF 4 中使用它们。

57.10.3 同时运行 WF 3.x 和 4

只要可能，就应尽量同时运行 WF 3.x 和 WF 4 工作流，除非不再有 3.x 工作流。因为永久存储器和跟踪存储器在 WF 4 中也不同，所以一个合并的系统是最简单的使用方法。如果没有长时间运行的工作流，就没有什么问题，但如果有长时间运行的工作流，最简单的方法是不更改旧工作流，并确保在 WF 4 上创建新工作流。

当应用程序启动工作流时，能用启动 WF 4 工作流的代码替换这些代码吗？如果不能，建议在代码中添加一个能识别版本的执行策略，这样就不必更改调用工作流的代码，并可以调度新工作流实例。

57.10.4 考虑把状态机迁移到流程图上

Workflow 4 目前不支持状态机，也没有对状态机的内置支持。基础活动模型可以处理状态机(实际上可以处理任何工作流处理模型)；然而在最初的 WF 4 版本中，没有状态机活动或设计支持。

与 3.x 状态机最接近的是 WF 4 的流程图，它们不是精确匹配，但流程图允许工作流跳转回处理过程的一个早期阶段，状态机工作流一般会这么做。

57.11 小结

Windows 工作流为应用程序的构造方式带来了根本性的改变。现在可以将应用程序的复杂部分都显示为活动，允许用户将活动拖放到工作流中，更改系统的处理方式。

几乎没有应用程序不能应用工作流，从最简单的命令行工具，到包含上百个模块的最复杂的系统。现在 WCF 的通信功能和 WPF 新的 UI 功能使应用程序前进了一大步，Windows 工作流的出现使开发和配置应用程序的方式有了根本的改变。

在 Visual Studio 2010 中，Workflow 3.x 基本上被 WF 4 取代，本章提供了简化升级的一些建议。如果正在计划第一次使用工作流，建议从 WF 4 开始，完全绕过 Workflow 3.x。