

可以扩展这个示例，在用户界面上根据需要提供这个验证错误列表，还可以在工具箱中添加更多活动，因为我们不希望工具箱中只有 3 个活动。

44.5 小结

Windows 工作流为应用程序的构建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看作活动，允许用户仅将活动拖放到工作流中，就可以修改系统的处理方式。

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。以前需要开发人员给系统编写一个扩展模块，而现在可以提供一个简单的、可扩展的、几乎任何人都可以使用的定制机制。应用程序供应商以前需要提供自定义活动，与系统交互操作，并提供在应用程序中调用工作流的代码，而现在可以让客户定义在事件发生时需要应用程序执行什么操作。

Workflow 4 极大地改进了 **Workflow 3.x**，如果打算第一次使用工作流，最好从这个新版本开始，完全跳过 **Workflow 3.x**。

第 45 章

对等网络

本章内容:

- P2P 概述
- Microsoft Windows Peer-to-Peer Networking 平台, 包括 PNRP 和 PNM
- 用 .NET Framework 构建 P2P 应用程序

对等网络(常常称为 P2P)是近年来出现的最有用且容易误解的技术。人们提到 P2P 时, 通常会想到: 共享音乐文件, 这常常是非法的。这是因为文件共享应用程序(如 BitTorrent)以令人吃惊的速度迅异军突起, 而这些应用程序使用 P2P 技术工作。

尽管 P2P 在文件共享应用程序中使用, 但这并不是说其他应用程序就不使用这个技术。事实上, 如本章所述, P2P 可以用于大量应用程序, 目前在我们生活的这个互联世界中变得越来越重要。本章第一部分将概述 P2P 技术。

微软公司并没有忘记 P2P 的出现, 而是开发了它自己的工具和技术, 来使用 P2P。Microsoft Windows Peer-to-Peer Networking 平台可以用作 P2P 应用程序的通信架构。这个平台包含重要的组件 Peer Name Resolution Protocol(PNRP)和 People Near Me(PNM)。另外, .NET Framework 3.5 还引入了一个新的名称空间 System.Net.PeerToPeer 和几个新类型及特性, 它们可用于构建 P2P 应用程序。

45.1 P2P 网络概述

对等网络是网络通信的另一种方式。为了解 P2P 与网络通信的“标准”方法之间的区别, 需要先回过头来看看客户端-服务器通信。目前, 客户端-服务器通信在网络应用程序中非常普遍。

45.1.1 客户端-服务器体系结构

传统上, 我们使用客户端-服务器体系结构, 通过网络(包括 Internet)与应用程序交互操作。Web 站点就是这方面的一个例子。在 Web 站点上, 通过 Internet 给 Web 服务器发送一个请求, Web 服务器会返回我们需要的信息。如果要下载文件, 就直接从 Web 服务器上下载。

同样, 包含局域网或广域网连接的桌面应用程序一般连接到一个服务器上, 如数据库服务器或包含其他服务的服务器。

客户端-服务器体系结构的这个简单形式如图 45-1 所示。

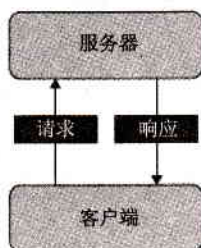


图 45-1

客户端-服务器体系结构本身并没有什么错误，而且在许多情况下，这就是我们需要的，但是它有一个可伸缩性问题。图 45-2 显示了客户端-服务器体系结构如何通过其他客户端来扩展。

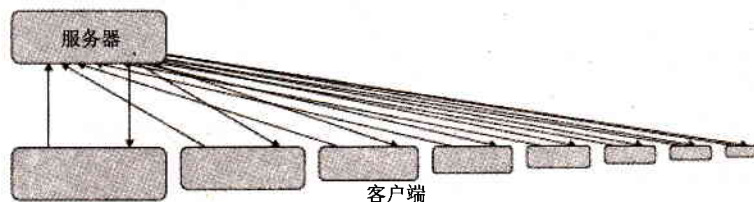


图 45-2

把每个添加了加大的负载的客户端放在服务器上，服务器就必须与每个客户端通信。再看看 Web 站点的例子，不断增加的通信负载就是 Web 站点崩溃的方式。有太多的流量，服务器简直无法响应。

当然，要缓解这种状况，可以实现一些扩展选项。可以增加服务器的功能和资源，也可以添加更多的服务器，来扩展服务器。这种扩展当然受到可用技术和硬件成本的限制。扩展可能比较灵活，但需要一个额外的基础体系层，来确保客户端与各个服务器的通信，并独立于与它们正在通信的服务器来维护会话状态。这有许多解决方案，如 Web 场或服务器场产品。

45.1.2 P2P 体系结构

对等方法完全不同于扩展方法。在 P2P 中，不是集中精力并试图使服务器及其客户端之间的通信更流畅，而是考虑客户端之间的通信方式。

例如，客户端与之通信的 Web 站点是 www.wrox.com。在一个假想的情形中，Wrox 声称，本书的新版本要在 Web 站点 wrox.com 上发布，且可以免费下载，但在某一天后删除它。在本书可以从该 Web 站点上得到之前，假定有许多人都想访问这个 Web 站点，刷新其浏览器，等待下载文件的出现。一旦文件出现在该 Web 站点上，每个人都试图同时下载它，这很可能使 wrox.com 的 Web 服务器在强大的压力下崩溃。

使用 P2P 技术可以避免这个 Web 服务器崩溃。P2P 技术不把文件直接从服务器发送给所有客户端，而是把文件仅发送给几个客户端。另外几个客户端可以从已经包含该文件的客户端上下载它，更多的客户端从这些二级客户端上下载该文件，以此类推。实际上，这个过程把文件分解成几个块，再把这些块分解到客户端上，一些客户端直接从服务器上下载文件，而一些客户端从其他客户端上下载文件，所以这个过程会更快完成。这就是文件共享技术(如 BitTorrent)的工作方式，如图 45-3 所示。

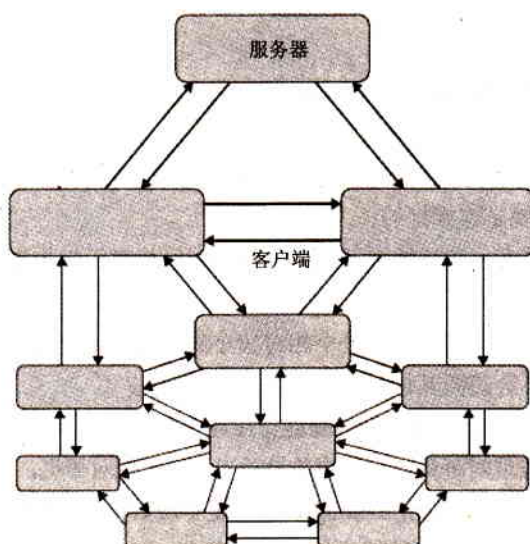


图 45-3

45.1.3 P2P 体系结构的挑战

在文件共享体系结构中，仍有一些问题需要解决。首先，客户端如何检测其他客户端的存在？如何定位其他客户端包含的文件块？另外，如何确保客户端之间的通信是按整个大陆来分割、优化的？

每个参与 P2P 网络应用程序的客户端都必须能执行如下操作，才能克服这些问题：

- 它必须能发现其他客户端
- 它必须能连接其他客户端
- 它必须能与其他客户端通信

发现问题有两个明显的解决方案。可以在服务器上保存客户端的列表，这样客户端就可以获得这个列表，并联系其他客户端(称为对等机)；也可以使用一个基础结构(如 PNRP，详见下一节)，它可以使客户端直接找到其他客户端。大多数文件共享系统都使用“服务器上的列表”解决方案，方法是把服务器称为跟踪器。另外，在文件共享系统中，任何客户端都可以用作服务器，如图 45-3 所示，只有声明客户端有一个文件，并用跟踪器注册它即可。实际上，纯粹的 P2P 网络根本不需要服务器，只需要对等机。

连接问题比较棘手，需要考虑 P2P 应用程序使用的网络的整体结构。如果有一组客户端，它们都可以彼此通信，这些客户端之间的连接拓扑结构就会非常复杂。为了提高性能，常常需要有多组客户端，每组客户端都包含该组中客户端之间的连接，但不包含其他组中的客户端连接。如果在本地建立了这些组，性能就会得到极大的提升，因为客户端可以彼此通信，而大大需要联网的计算机之间的跳跃通信。

通信问题不太重要，因为也建立了通信协议(如 TCP/IP)，并可以在这种情况下重用这些协议。但是，高端技术(例如，可以使用 WCF 服务，因此可以使用 WCF 提供的所有功能)和低端协议(例如，把数据同时发送给多个端点的多播协议)都有性能提升的空间。

发现、连接和通信是所有 P2P 实现方案的核心。本章介绍的实现方案使用 System.Net.PeerToPeer 类型和 PNM 进行发现，使用 PNRP 进行连接。如下面几节所述，这些技术涵盖了上述 3 个操作。

45.1.4 P2P 术语

前面几节介绍了对等机的概念,这是在 P2P 网络中引用客户端的方式。“客户端”在 P2P 网络中没有意义,因为客户端不需要服务器。

彼此连接的对等机的组合称为网(mesh)、云(cloud)或图(graph),这些术语可以互换。如果满足如下条件之一,给定的组就是连接好的:

- 每对对等机之间都有一条连接路径,这样每个对等机都可以根据需要连接到任何其他对等机上。
- 在任意一对对等机之间遍历都有一个相对较小的连接数。
- 删除一个对等机不会妨碍其他对等机的彼此连接。

注意,这并不意味着,每个对等机都必须能直接连接到其他所有对等机上。实际上,如果用数学方法分析网络,那么会发现对等机只需连接数量相当少的其他对等机,来满足这些条件。

另一个要注意的 P2P 概念是溢出(flooding)。溢出是单块数据通过网络传播到所有对等机上的方式,或者在网络中查询其他节点以定位特定块数据的方式。在未结构化的 P2P 网络中,先连接与自己最近的对等机,这个对等机再连接与它最近的对等机,以此类推,直到连接了网络中的所有对等机为止,这是一个相当随机的过程。也可以创建结构化的 P2P 网络,这样,查询和对等机之间的数据流就有定义好的路径了。

45.1.5 P2P 解决方案

为 P2P 建立了基础结构后,不仅可以开发客户端-服务器应用程序的改进版本,还可以开发全新的应用程序。P2P 特别适合于下述类型的应用程序:

- 内容发布应用程序,包括前面讨论的文件共享应用程序。
- 合作应用程序,例如,桌面共享和共享白板应用程序。
- 多用户通信应用程序,允许用户直接通信和交换数据,而不是通过服务器通信。
- 分布式处理应用程序,作为超级计算应用程序的另一种方式,处理海量数据。
- Web 2.0 应用程序,在下一代动态 Web 应用程序中合并上述的一部分或全部功能。

45.2 Microsoft Windows Peer-to-Peer Networking

Microsoft Windows Peer-to-Peer Networking 平台是 Microsoft 的 P2P 技术实现方式。它是 Windows XP SP2、Windows Vista 和 Windows 7 的一部分,也可用作 Windows XP SP1 的一个插件。它包括两种技术,在创建.NET P2P 应用程序时可以使用这两种技术:

- Peer Name Resolution Protocol(PNRP),它用于发布和解析对等机的地址。
- People Near Me(PNM)服务器,它用于定位本地的对等机(目前只能在 Vista 和 Windows 7 中使用)。

本节学习这两种技术。

45.2.1 PNRP

当然,可以使用任意协议实现 P2P 应用程序,但如果在 Microsoft Windows 环境下工作(如果读

者在阅读本书,就可能在 Microsoft Windows 环境下工作),考虑 PNRP 至少是有意义的。目前 PNRP 发布了两个版本。PNRP 1 包含在 Windows XP SP2、Windows XP Professional x64 Edition 和 Windows XP SP1 with the Advanced Networking Pack for Windows XP。PNRP 2 随 Windows Vista 一起发布,Windows XP SP2 用户可以通过一个独立的下载包使用它(参见 support.microsoft.com/kb/920342 的 KB920342)。Windows 7 也使用版本 2。PNRP 1 和 PNRP 2 版本不兼容,本章仅介绍版本 2。

PNRP 本身并没有提供创建 P2P 应用程序所需的功能,而只是其中一种可用于解析对等机地址的底层技术。PNRP 允许客户端注册一个端点(称为对等机名称),该端点自动在云中的对等机之间循环。这个对等机名称封装在 PNRP ID 中。发现这个 PNRP ID 的对等机可以使用 PNRP 把它解析为实际的对等机名称,然后它就可以直接与相关联的客户端通信。

例如,定义一个表示 WCF 服务端点的对等机名称。可以使用 PNRP 把云中的这个对等机名称注册为 PNRP ID。运行合适的客户端应用程序的对等机使用一个发现机制,该机制可以标识为服务提供的对等机名称,接着这个对等机可能发现这个 PNRP ID。之后对等机就使用 PNRP 定位 WCF 服务的端点,并使用该服务。



重要的是,PNRP 没有假定对等机名称实际上表示什么。而是在发现对等机名称时,由对等机决定如何使用它们。在解析 PNRP ID 时,对等机从 PNRP 中获得的信息包括 ID 发布者的 IPv6(实际上还有 IPv4)地址和一个端口号,有时还有少量其他数据。除非对等机知道对等机名称的含义,否则不可能利用该信息执行有意义的操作。

1. PNRP ID

PNRP ID 是一个 256 位的标识符。低位上的 128 位用于唯一标识对等机,高位上的 128 位标识一个对等机名称。高位上的 128 位是来自发布对等机的一个散列公钥和一个至多 149 个字符的字符串的散列组合,该字符串用于标识对等机名称。散列公钥(称为授权)和这个字符串(称为分类器)统称为 P2P ID。也可以使用 0 代替散列公钥,此时对等机名称是不安全的(与使用公钥的安全对等机名称相反)。

PNRP ID 的结构如图 45-4 所示。

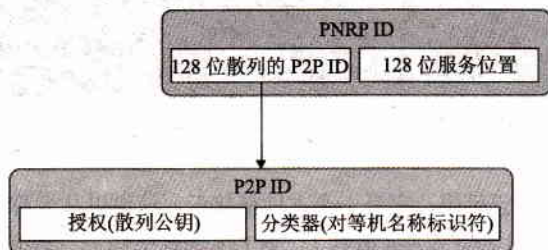


图 45-4

对等机上的 PNRP 服务负责维护一个 PNRP ID 列表,包括它发布的 PNRP ID 和通过云中其他地方的 PNRP 服务实例获得的 PNRP ID 缓存列表。对等机尝试解析 PNRP ID 时,PNRP 服务会使用端点的一个缓存副本来解析发布 PNRP 的对等机,如果对等机的邻居能解析它,就请求它们解析。最终建立与发布对等机之间的连接,PNRP 服务能解析 PNRP ID。

注意,在进行上述所有操作时,不需要外界干预。我们只需确保对等机在用本地的 PNRP 服务

解析了 PNRP ID 后, 知道如何处理对等机名称。

对等机可以使用 PNRP 定位匹配某个特定 P2P ID 的 PNRP ID。使用这个功能可以为不安全的对等机名称实现很基本的发现机制。这是因为如果几个对等机提供了一个使用相同分类器且不安全的对等机名称, P2P ID 就会相同。当然, 因为任意对等机都可以使用不安全的对等机名称, 用户必须确保所连接的端点就是希望的端点类型, 所以这只是在本地网络中发现对等机的一种可行解决方案。

2. PNRP 云

在前面的讨论中, 学习了 PNRP 如何注册和解析云中的对等机名称。云通过种子服务器维护, 该服务器可以是运行 PNRP 服务的任意服务器, 但 PNRP 服务至少要维护一个对等机记录。PNRP 服务可以使用两种类型的云:

- **本地链接**——这类云包含连接到本地网络上的计算机。如果 PC 有多个网络适配器, 它就可以连接到多个本地链接的云上。
- **全局**——这类云包含默认连接到 Internet 上的计算机, 尽管也可以定义一个私有的全局云。其区别是 Microsoft 为全局的 Internet 云维护种子服务器, 而如果定义了私有的全局云, 就必须使用自己的种子服务器。如果使用自己的种子服务器, 就必须配置策略设置, 从而确保所有对等机都连接到种子服务器上。



在 PNRP 的以前版本中, 有第 3 种云: 本地站点。现在不再使用它, 所以本章不探讨它。

使用下面的命令可以确定自己连接到什么类型的云上:

```
netsh p2p pnrp cloud show list
```

通常结果如图 45-5 所示。

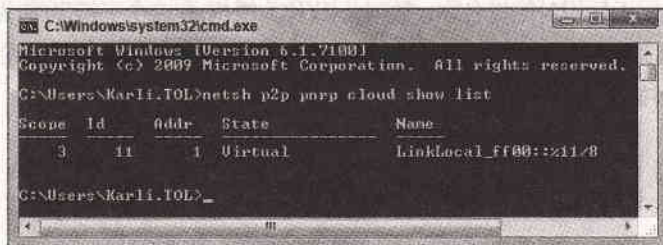


图 45-5

图 45-5 显示, 只有一个云可用, 它是一个本地链接云。这可以从名称和 Scope 值看出, 本地链接云的 Scope 值是 3, 而全局云的 Scope 值是 1。为了连接到全局云, 必须有一个全局的 IPv6 地址。因为用于获得图 45-5 的计算机没有全局的 IPv6 地址, 所以只有一个本地云。

云的状态如下:

- **Active**——如果云的状态是 active, 就可以使用它发布和解析对等机名称。
- **Alone**——如果从云中查询的对等机没有连接到其他对等机上, 其状态就是 alone。
- **No Net**——如果对等机没有连接到网络上, 云的状态就从 active 改为 no net。

- **Synchronizing**——对等机连接到云上时，云的状态就是 **synchronizing**。这个状态可以很快变成其他状态，因为连接不需要很长时间，所以可能从来看不到云在这个状态下。
- **Virtual**——PNRP 服务仅根据需要，通过对等机名称注册和解析功能连接到云上。如果云的连接停用的时间超过 15 分钟，它就会进入这个 **virtual** 状态。



如果读者遇到网络连接问题，就应检查防火墙，以防它禁止通过 UDP 端口 3540 或 1900 的本地网络流量。UDP 端口 3540 由 PNRP 使用，UDP 端口 1900 由 SSDP(Simple Service Discovery Protocol, 简单服务发现协议)使用，SSDP 由 PNRP 服务(以及 UPnP 设备使用)。

3. Windows 7 中的 PNRP

在 Windows 7 中，PNRP 使用一个新的组件 DRT (Distributed Routing Table, 分布式路由表)，这个组件负责确定 PNRP 使用的密钥的结构，其默认的实现方式是前面描述的 PNRP ID。使用 DRT API 可以定义另一种密钥模式，但密钥必须是 256 位的整数值(与 PNRP ID 一样)。这表示，虽然可以使用任意模式，但必须自己负责密钥的生成和安全性。使用这个组件可以在 PNRP 的范围之外创建新的云拓扑结构。这是一种高级技术，超出了本章的范围。

Windows 7 还引入了一种新方式 Easy Connect，用于连接到 Remote Assistance 应用程序的其他用户上。这个连接选项使用 PNRP 定位要连接的用户。一旦创建了一个会话，用户就可以通过 Easy Connect 或其他方式(如电子邮件邀请)共享其桌面，并通过 Remote Assistance 接口互相帮助。



图 45-6

45.2.2 People Near Me

如上一节所述，PNRP 用于定位对等机。在考虑 P2P 应用程序的发现/连接/通信过程时，这显然是一种重要的可用技术，但 PNRP 本身并不是这些阶段的完整实现方式。People Near Me 服务是发现阶段的一种实现方式，可以定位在本地区域(即，我们连接到的本地链接云)的 Window People Near Me 服务中签名的本地对等机。

我们可能要使用这个服务，因为它内置在 Vista 和 Windows 7 中，在 Windows Meeting Space 应用程序中使用它，该应用程序可用于在对等机之间共享应用程序。通过控制面板中的 Change People Near Me 选项(在“开始”菜单的搜索框中输入 **people**，就可以快速导航到这个选项上)，可以配置这个服务。这个控制面板项会显示如图 45-6 所示的对话框。

一旦在 PNM 服务中签名，该服务就可以用于配置为使用该服务的应用程序。

编写本书时，PNM 只能在 Windows Vista 系列的操作系统上使用(它从 Windows 7 中删除了)。但未来的服务包或其他下载软件包可能使之可用于 Windows XP 上。

45.3 构建 P2P 应用程序

既然学习了 P2P 网络的概念和 .NET 开发人员可用于实现 P2P 应用程序的技术, 就该讨论如何构建它们了。从前面的讨论中知道, 因为使用 PNRP 可以发布、分配和解析对等机名称, 所以这里首先要了解如何使用 .NET 完成这个任务。接着陈述如何把 PNM 作为 P2P 应用程序的架构。这很有优势, 因为如果使用 PNM, 就不必实现自己的发现机制。

为了研究这些主题, 需要学习如下名称空间中的类:

- System.Net.PeerToPeer
- System.Net.PeerToPeer.Collaboration

要使用这些类, 必须引用 System.Net.dll 程序集。

45.3.1 System.Net.PeerToPeer

System.Net.PeerToPeer 名称空间中的类封装了 PNRP 的 API, 允许与 PNRP 服务交互操作。使用这些类可以完成两个主要任务:

- 注册对等机名称
- 解析对等机名称

在下面几节中, 除非特别指出, 否则提到的所有类型都来自 System.Net.PeerToPeer 名称空间。

1. 注册对等机名称

要注册对等机名称, 必须执行如下步骤:

(1) 用特定的分类器创建一个安全或不安全的对等机名称。

(2) 为对等机名称配置一个注册项, 指定如下可选信息:

- TCP 端口号
- 用于注册对等机名称的云(如果未指定, PNRP 就在所有可用的云中注册对等机名称)
- 至多 39 个字符的注释
- 至多 4096 个字节的其他数据
- 是否自动为对等机名称生成端点(默认行为是, 端点应从对等机的 IP 地址和端口号(假设指定了端口号)中生成)
- 端点集合

(3) 使用对等机名称注册项, 通过本地 PNRP 服务注册对等机名称

在第 3 步之后, 对等机名称就可以用于所选云中的所有对等机。对等机注册在明确地停止后不再能使用, 或者在注册对等机名称的过程终止后不再能使用。

要创建对等机名称, 可以使用 PeerName 类。以 authority.classifier 的形式从 P2P ID 的字符串表示中创建这个类的一个实例, 或者从一个分类器字符串和 PeerNameType 中创建。可以使用 PeerNameType.Secured 或 PeerNameType.Unsecured。例如:

```
PeerName pn = new PeerName("Peer classifier", PeerNameType.Secured);
```

因为不安全的对等机名称使用的 authority 值是 0, 所以下面的代码行与上述代码等价:

```
PeerName pn = new PeerName("Peer classifier", PeerNameType.Unsecured);

PeerName pn = new PeerName("0.Peer classifier");
```

有了 `PeerName` 实例后, 就可以使用它和一个端口号初始化一个 `PeerNameRegistration` 对象:

```
PeerNameRegistration pnr = new PeerNameRegistration(pn, 8080);
```

另外, 还可以在使用其默认参数创建的 `PeerNameRegistration` 对象上设置 `PeerName` 和 `Port`(可选) 属性。也可以把 `Cloud` 实例指定为 `PeerNameRegistration` 构造函数的第 3 个参数, 或者通过 `Cloud` 属性来指定。从云的名称中可以得到一个 `Cloud` 实例, 或者使用 `Cloud` 如下的一个静态成员:

- `Cloud.Global`——这个静态属性获取全局云的一个引用。根据对等机策略配置, 这可以是私有的全局云。
- `Cloud.AllLinkLocal`——这个静态字段获取一个云, 该云包含可用于对等机的所有本地链接。
- `Cloud.Available`——这个静态字段获取一个云, 该云包含可用于对等机的所有云, 其中包含本地链接云和(如果可用)全局云。

创建了 `Cloud` 实例后, 可以根据需要设置 `Comment` 和 `Data` 属性。但应注意这些属性的限制。如果试图把 `Comment` 设置为超过 39 个 Unicode 字符的字符串, 就会接收到一个 `PeerToPeerException` 异常, 如果试图把 `Data` 设置为超过 4096 个字节的 `byte[]`, 就会得到一个 `ArgumentOutOfRangeException` 异常。还可以使用 `EndPointCollection` 属性添加端点。这个属性是 `System.Net.IPEndPoint` 对象的一个 `System.Net.IPEndPointCollection` 集合。如果使用 `EndPointCollection` 属性, 还需要把 `UseAutoEndPointSelection` 属性设置为 `false`, 以禁止自动生成端点。

准备注册对等机名称时, 可以调用 `PeerNameRegistration.Start()` 方法。要从 PNRP 服务中删除对等机名称注册项, 可以使用 `PeerNameRegistration.Stop()` 方法。

下面的代码通过注释注册一个的安全对等机名称:

```
PeerName pn = new PeerName("Peer classifier", PeerNameType.Unsecured);
PeerNameRegistration pnr = new PeerNameRegistration(pn, 8080);
pnr.Comment = "Get pizza here";
pnr.Start();
```

2. 解析对等机名称

要解析对等机名称, 必须执行如下步骤:

- (1) 从已知的 P2P ID 或通过发现技术得到的 P2P ID 中生成一个对等机名称。
- (2) 使用解析器解析对等机名称, 得到一个对等机名称记录集合。可以把解析器限制为只用于某个特定的云, 和/或要最多返回的结果数。
- (3) 对于获得的每个对等机名称记录, 需要获取对等机名称、端点、注释和额外的数据信息。

这个过程从一个类似于对等机名称注册项的 `PeerName` 对象开始。这里的区别是使用通过一个或多个远程对等机注册的对等机名称。从本地链接云中获取活动对等机列表最简单的方式是使用同一个分类器为每个对等机注册一个不安全的对等机名称, 再在解析阶段使用同一个对等机名称。但是对于全局云, 不推荐使用这个方法, 因为不安全的对等机名称很容易被窃取。

要解析对等机名称, 可以使用 `PeerNameResolver` 类。有了这个类的一个实例后, 就可以使用 `Resolve()` 方法同步解析对等机名称, 或者使用 `ResolveAsync()` 方法异步解析对等机名称。

用一个 `PeerName` 参数可以调用 `Resolve()` 方法, 也可以给该方法传递要解析的可选 `Cloud` 实例, 或者传递要返回的最大对等机数(int), 或者传递这两个参数。这个方法返回一个 `PeerNameRecordCollection` 实例, 这是 `PeerNameRecord` 对象的集合。例如, 下面的代码解析所有本地链接云中一个不安全的对等机名称, 最多返回 5 个结果:

```
PeerName pn = new PeerName("0.Peer classifier");
PeerNameResolver pnres = new PeerNameResolver();
PeerNameRecordCollection pnrc = pnres.Resolve(pn, Cloud.AllLinkLocal, 5);
```

`ResolveAsync()` 方法使用一个标准的异步方法调用模式。给该方法传递一个唯一的 `userState` 对象, 为正在查找的对等机侦听 `ResolveProgressChanged` 事件, 在该方法终止时侦听 `ResolveCompleted` 事件。可以用 `ResolveAsyncCancel()` 方法取消未决的异步请求。

`ResolveProgressChanged` 事件的处理程序使用 `ResolveProgressChangedEventArgs` 事件参数, 该参数派生自标准的 `System.ComponentModel.ProgressChangedEventArgs` 类。可以使用在事件处理程序中接收到的事件参数对象的 `PeerNameRecord` 属性获得对查找到的对等机名称记录的引用。

同样, `ResolveCompleted` 事件也需要一个事件处理程序, 它使用 `ResolveCompletedEventArgs` 类型的参数, 该参数派生自 `AsyncCompletedEventArgs`。这个类型包含一个 `PeerNameRecordCollection` 参数, 可用于获得查找到的对等机名称记录的完整列表。

下面的代码显示了这两个事件的事件处理程序的实现代码:

```
private pnres_ResolveProgressChanged(object sender,
    ResolveProgressChangedEventArgs e)
{
    // Use e.ProgressPercentage (inherited from base event args)
    // Process PeerNameRecord from e.PeerNameRecord
}

private pnres_ResolveCompleted(object sender,
    ResolveCompletedEventArgs e)
{
    // Test for e.IsCancelled and e.Error (inherited from base event args)
    // Process PeerNameRecordCollection from e.PeerNameRecordCollection
}
```

有一个或多个 `PeerNameRecord` 对象后, 就可以处理它们了。这个 `PeerNameRecord` 类提供 `Comment` 和 `Data` 属性, 用于检查在对等机名称注册项(如果存在)中设置的注释和数据, `PeerName` 属性可获取对等机名称记录的 `PeerName` 对象, 最重要的是 `EndPointCollection` 属性。与 `PeerNameRecordRegistration` 一样, 这个属性也是 `System.Net.IPEndPoint` 对象的一个 `System.Net.IPEndPointCollection` 集合。这些对象可用于以任意方式连接对等机提供的端点。

3. System.Net.PeerToPeer 中的代码访问安全性

`System.Net.PeerToPeer` 名称空间还包含如下两个用于 CAS(参见第 20 章)的类, 更多信息可参见第 21 章:

- `PnrpPermission`, 它继承自 `CodeAccessPermission`
- `PnrpPermissionAttribute`, 它继承自 `CodeAccessSecurityAttribute`

这两个类可以用通常的 CAS 方式给 PNRP 访问提供权限功能。

4. 示例应用程序

本章可下载的代码包含一个示例 P2P 应用程序(P2PSample)，它使用本节介绍的概念和名称空间。它是一个 WPF 应用程序，它为对一个等机端点使用一个 WCF 服务。

该应用程序用一个应用程序配置文件来配置，在该文件中，可以指定对等机的名称和侦听的端口，如下所示：



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="username" value="Karli" />
    <add key="port" value="8731" />
  </appSettings>
</configuration>
```

代码段 App.config

构建应用程序后，或者把它复制到本地网络的其他计算机上并运行所有实例来测试它；或者在一台计算机上运行多个实例来测试它。如果选择后一个选项，就必须修改每个配置文件(复制本地计算机上 Debug 目录的内容，依次编辑每个配置文件)，以修改每个实例使用的端口。用这两种方式测试该应用程序时，如果还修改了每个实例的用户名，结果就会更清楚。

运行对等机应用程序后，就可以使用 Refresh 按钮异步地获得对等机列表。定位了一个对等机后，就可以单击对等机的 Message 按钮，发送一条默认消息。

图 45-7 显示了这个应用程序，它在一台计算机上运行 3 个实例。在图 45-7 中，一个对等机给另一个对等机发送了消息，这会生成一个对话框。

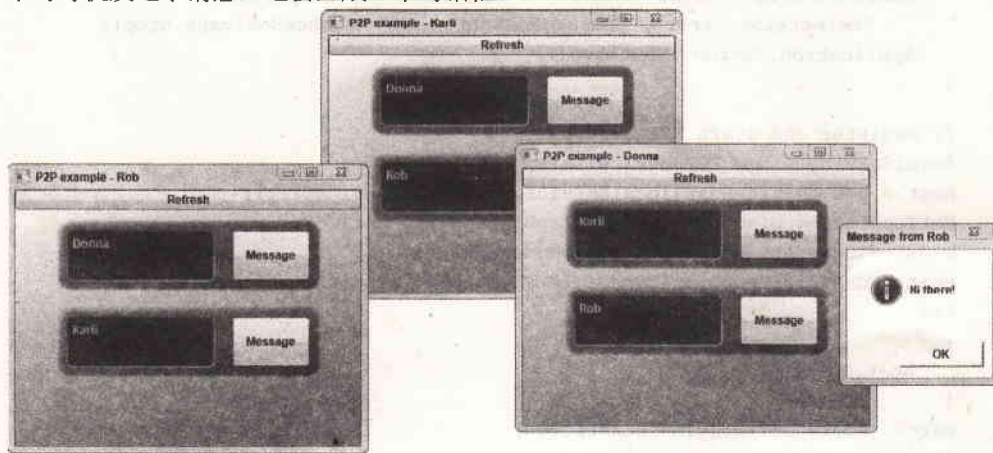


图 45-7

在这个应用程序中，大多数工作都在 Window1 窗口的 Window_Loaded() 事件处理程序中完成。这个方法首先加载配置信息，并用用户名设置窗口的标题：



可从
wrox.com
下载源代码

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Get configuration from app.config
    string port = ConfigurationManager.AppSettings["port"];
    string username = ConfigurationManager.AppSettings["username"];
```

```

string machineName = Environment.MachineName;
string serviceUrl = null;

// Set window title
this.Title = string.Format("P2P example-{0}", username);

```

代码段 Window1.xaml.cs

接着使用对等机主机地址和配置的端口确定要把 WCF 服务存放在哪个端点上。因为该服务使用 `NetTcpBinding` 绑定，所以端点的 URL 使用 `net.tcp` 协议：

```

// Get service url using IPv4 address and port from config file
foreach (IPAddress address in Dns.GetHostAddresses(Dns.GetHostName()))
{
    if (address.AddressFamily ==
        System.Net.Sockets.AddressFamily.InterNetwork)
    {
        serviceUrl = string.Format("net.tcp://{0}:{1}/P2PService",
            address, port);
        break;
    }
}

```

验证端点的 URL 后，就注册并启动 WCF 服务：

```

// Check for null address
if (serviceUrl == null)
{
    // Display error and shutdown
    MessageBox.Show(this, "Unable to determine WCF endpoint.",
        "Networking Error", MessageBoxButton.OK, MessageBoxImage.Stop);
    Application.Current.Shutdown();
}

// Register and start WCF service.
localService = new P2PService(this, username);
host = new ServiceHost(localService, new Uri(serviceUrl));
NetTcpBinding binding = new NetTcpBinding();
binding.Security.Mode = SecurityMode.None;
host.AddServiceEndpoint(typeof(IP2PService), binding, serviceUrl);
try
{
    host.Open();
}
catch (AddressAlreadyInUseException)
{
    // Display error and shutdown
    MessageBox.Show(this, "Cannot start listening, port in use.",
        "WCF Error", MessageBoxButton.OK, MessageBoxImage.Stop);
    Application.Current.Shutdown();
}

```

该服务类的单一实例用于启动主机应用程序和服务之间的简便通信(用于发送和接收消息)。另外要注意，在绑定配置中，为了简单起见，禁用了安全性。

之后，使用 `System.Net.PeerToPeer` 名称空间中的类注册对等机名称：

```

    // Create peer name
    peerName = new PeerName("P2P Sample", PeerNameType.Unsecured);

    // Prepare peer name registration in link local clouds
    peerNameRegistration = new PeerNameRegistration(peerName, int.Parse(port));
    peerNameRegistration.Cloud = Cloud.AllLinkLocal;

    // Start registration
    peerNameRegistration.Start();
}

```

单击 **Refresh** 按钮时, **RefreshButton_Click()** 事件处理程序使用 **PeerNameResolveResolveAsync()** 异步地获得对等机:

```

private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    // Create resolver and add event handlers
    PeerNameResolver resolver = new PeerNameResolver();
    resolver.ResolveProgressChanged +=
        new EventHandler<ResolveProgressChangedEventArgs>(
            resolver_ResolveProgressChanged);
    resolver.ResolveCompleted +=
        new EventHandler<ResolveCompletedEventArgs>(
            resolver_ResolveCompleted);

    // Prepare for new peers
    PeerList.Items.Clear();
    RefreshButton.IsEnabled = false;

    // Resolve unsecured peers asynchronously
    resolver.ResolveAsync(new PeerName("0.P2P Sample"), 1);
}

```

剩余的代码负责显示对等机, 并与对等机通信, 读者可以自己研究它们。

通过 P2P 云提供 WCF 端点是定位企业中的服务的一种好方法, 也是在对等机之间通信的好方法, 如本例所示。

45.3.2 System.Net.PeerToPeer.Collaboration

System.Net.PeerToPeer.Collaboration 名称空间中的类提供了一个架构, 该架构可用于创建使用 PNM 服务和 P2P 协作 API 的应用程序。如前所述, 在编写本书时, 只有使用 Windows Vista 或 Windows 7, 才能创建这类应用程序。

这个名称空间中的类可以用许多方式与对等机和应用程序进行交互操作, 包括:

- 签入和签出
- 发现对等机
- 管理联系人和检测对等机的存在

还可以使用这个名称空间中的类邀请其他用户参与某个应用程序, 在用户和应用程序之间交换数据, 但是, 为此, 需要创建支持 PNM 的应用程序, 这超出了本章的范围。

在下面几节中, 除非特别指出, 否则所有类型都来自 **System.Net.PeerToPeer.Collaboration** 名称空间。

1. 签入和签出

`System.Net.PeerToPeer.Collaboration` 名称空间中最重要的一个类是 `PeerCollaboration` 类。这是一个静态类, 提供了许多用于各种目的的静态方法, 如本节和下面几节所述。使用其中的两个方法 `SignIn()` 和 `SignOut()`, 可以签入和签出 PNM 服务。这两个方法都接受一个 `PeerScope` 类型的参数, 该参数的值如下:

- `PeerScope.None`——如果使用这个值, `SignIn()` 和 `SignOut()` 就没有任何作用。
- `PeerScope.NearMe`——它表示签入和签出本地链接云。
- `PeerScope.Internet`——它表示签入和签出全局云(连接到当前不在本地子网上的联系人时, 需要使用这个值)。
- `PeerScope.All`——它表示签入和签出所有可用的云。

根据需要, 调用 `SignIn()` 会打开 `People Near Me` 配置对话框。

签入对等机后, 就可以把 `PeerCollaboration.LocalPresenceInfo` 属性设置为 `PeerPresenceInfo` 类型的一个值。这会启用标准的 IM 功能, 例如, 把状态设置为“离开”。`PeerPresenceInfo.DescriptiveText` 属性可以设置为至多 255 个字符的一个 Unicode 字符串, `PeerPresenceInfo.PresenceStatus` 属性可以设置为 `PeerPresenceStatus` 枚举的一个值。这个枚举可以使用的值如下:

- `PeerPresenceStatus.Away`——对等机离开。
- `PeerPresenceStatus.BeRightBack`——对等机离开, 但不久会返回。
- `PeerPresenceStatus.Busy`——对等机在忙。
- `PeerPresenceStatus.Idle`——对等机未激活。
- `PeerPresenceStatus.Offline`——对等机离线。
- `PeerPresenceStatus.Online`——对等机在线, 可以使用。
- `PeerPresenceStatus.OnThePhone`——对等机在忙着通话。
- `PeerPresenceStatus.OutToLunch`——对等机离开, 但在午餐后会返回。

2. 发现对等机

如果登录到本地链接云上, 就可以获得附近的对等机列表。为此可以使用 `PeerCollaboration.GetPeersNearMe()` 方法。这个方法返回包含 `PeerNearMe` 对象的一个 `PeerNearMeCollection` 对象。

使用 `PeerNearMe` 对象的 `Nickname` 属性可以获得对等机的名称, 使用 `IsOnline` 属性可以确定对等机是否在线, 使用 `PeerEndpoints` 属性(用于低级操作)可以确定与对等机相关的端点。如果要确定 `PeerNearMe` 对象的在线状态, 那么也需要使用 `PeerEndpoints` 属性。可以给 `GetPresenceInfo()` 方法传递一个端点, 来获得一个 `PeerPresenceInfo` 对象, 如上一节所述。

3. 管理联系人并检测对等机的存在

联系是记住对等机的一种方式。可以添加通过 PNM 服务发现的对等机, 之后, 只要自己在线, 就可以连接到该对等机。也可以通过本地链接云或全局云连接到联系人(假定与 Internet 建立了 IPv6 连接)。

可以调用 `PeerNearMe.AddToContactManager()` 方法, 从对等机中添加已发现的联系人。在调用这个方法时, 可以选择把显示姓名、昵称和电子邮件地址关联到联系人上。但一般使用

ContactManager 类来管理联系人。

无论如何处理联系人,都要处理 PeerContact 对象。这个对象与 PeerNearMe 对象一样,也继承自 Peer 抽象基类。PeerContact 对象的属性和方法比 PeerNearMe 对象多。例如,PeerContact 对象包括 DisplayName 和 EmailAddress 属性,它们进一步描述了 PNM 对等机。这两个类型的另一个区别是 PeerContact 对象与 System.Net.PeerToPeer.PeerName 类有更明确的关系。通过 PeerContact.PeerName 属性可以从 PeerContact 对象中获得 PeerName。之后,就可以使用前面介绍的技术与 PeerName 提供的任意端点通信。

本地对等机的信息也可以通过 ContactManager 类的 ContactManager.LocalContact 静态属性来访问。该属性提供了一个 PeerContact 属性和本地对等机的详细信息。

使用 ContactManager.CreateContact()或 CreateContactAsync()方法,可以把 PeerNearMe 对象添加到联系人的本地列表中。使用 GetContact()方法可以获得 PeerName 对象。使用 DeleteContact()方法可以删除 PeerNearMe 或 PeerName 对象表示的联系人。

最后,可以处理一些事件来响应对联系人的修改。例如,改变了 ContactManager 类已知的任意联系人的存在状态时,可以使用 PresenceChanged 事件来响应。

4. 示例应用程序

这是本章可下载代码中的第二个示例应用程序,它说明了 System.Net.PeerToPeer.Collaboration 名称空间中的类的用法。这个应用程序类似于另一个示例,但要简单许多。我们需要两台都可以签到到 PNM 服务器上的计算机,才能看到这个应用程序的执行情况,因为它枚举并显示了本地子网中的 PNM 对等机。



图 45-8

在至少可以发现一个对等机可用的情况下运行应用程序,结果如图 45-8 所示。

因为代码的结构与上一个例子相同,所以如果阅读了前面的代码,就会很熟悉这个例子的代码。这次在 Window_Loaded()事件处理程序中,除了签入之外,并没有太多的工作要做,因为没有要初始化的 WCF 服务,也没有要获得的对等机名称注册项:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Sign in to PNM
    PeerCollaboration.SignIn(PeerScope.NearMe);
}
```

为了使结果更美观,使用 ContactManager.LocalContact.Nickname 格式化窗口的标题:

```
// Get local peer name to display
this.Title = string.Format("PNMSample-{0}",
    ContactManager.LocalContact.Nickname);
}
```

在 Window_Closing()中,本地对等机自动签出 PNM:

```
private void Window_Closing(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    // Sign out of PNM
}
```

```

        PeerCollaboration.SignOut(PeerScope.NearMe);
    }

```

大多数工作都在 `RefreshButton_Click()` 事件处理程序中完成, 这使用 `PeerCollaboration.GetPeersNearMe()` 方法获得对等机列表, 并使用项目中定义的 `PeerEntry` 类, 把这些对等机添加到显示结果中, 如果没有找到任何对等机, 就显示一条失败消息。

```

private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
    // Get local peers
    PeerNearMeCollection peersNearMe = PeerCollaboration.GetPeersNearMe();

    // Prepare for new peers
    PeerList.Items.Clear();

    // Examine peers
    foreach (PeerNearMe peerNearMe in peersNearMe)
    {
        PeerList.Items.Add(
            new PeerEntry
            {
                PeerNearMe = peerNearMe,
                PresenceStatus = peerNearMe.GetPresenceInfo(
                    peerNearMe.PeerEndpoints[0]).PresenceStatus,
                DisplayString = peerNearMe.Nickname
            });
    }

    // Add failure message if necessary
    if (PeerList.Items.Count == 0)
    {
        PeerList.Items.Add(
            new PeerEntry
            {
                DisplayString = "No peers found."
            });
    }
}

```

从这个例子可以看出, 使用前面介绍的类, 可以非常方便地与 PNM 服务交互。

45.4 小结

本章介绍了如何在应用程序中使用 .NET 4 中的 P2P 类实现对等网(P2P)功能。

我们探讨了 P2P 可以实现的解决方案类型, 这些解决方案的构造方式, 如何使用 PNRP 和 PNM, 如何使用 `System.Net.PeerToPeer` 和 `System.Net.PeerToPeer.Collaboration` 名称空间中的类型, 还了解了作为 P2P 端点提供 WCF 服务的有用技术。

如果读者对开发 P2P 应用程序感兴趣, 就应进一步研究 PNM。还应了解对等机信道, WCF 服务利用该信道在多个客户端之间同时进行通信。

下一章介绍消息队列。

第 46 章

消息队列

本章内容:

- 消息队列概述
- 消息队列体系结构
- 消息队列管理工具
- 编程实现消息队列
- 课程订单示例应用程序
- 消息队列和 WCF

`System.Messaging` 名称空间包含的类可以用 Windows 操作系统的消息队列功能读写消息。消息传递功能可以在断开连接的环境下使用,在该环境下,客户端和服务端不需要同时运行。

本章介绍消息队列的体系结构和用法,探讨 `System.Messaging` 名称空间中用于创建队列和收发消息的类,学习如何使用确认队列和响应队列从服务器中获得应答,如何通过 WCF 消息队列绑定使用消息队列。

46.1 概述

在开始学习消息队列编程之前,本节先讨论消息传递的基本概念,将它与同步和异步编程比较。在同步编程中,调用一个方法时,调用者必须等待该方法执行完毕。在异步编程中,主调线程可以启动同时运行的方法。异步编程可以通过委托、支持异步方法的类库(例如,Web 服务代理, `System.Net` 和 `System.IO` 类),或使用自定义线程(详见第 20 章)来实现。在同步和异步编程中,客户端和服务端必须同时运行。

尽管消息队列是异步进行的,但因为客户端(发送者)不等待服务器(接收者)读取发送给它的数据,所以消息队列与异步编程有很大的区别。消息队列可以在断开连接的环境下进行。在发送数据时,接收者可以离线。在以后的某个时刻,接收者上线时,就会接收到数据,而无须来自发送应用程序的干预。

可以把打开连接的编程和断开连接的编程与给他人打电话和发送电子邮件做比较。在给他人打电话时,电话两端的人都必须同时在线;这种通信是同步的。而利用电子邮件,发送者不能确定处理电子邮件的时间。使用这个技术的人工作在断开连接的模式。当然,电子邮件可能从来不被处理,

而它可能被忽略。这就是断开连接的通信的本质。为了避免这个问题，可以要求发送一个应答，以确认已读取电子邮件。如果在指定的时间内没有收到应答，就要处理这个“异常”。这也可以使用消息队列来实现。

在某些方面，消息队列就是应用程序之间通信的电子邮件，而不是人与人之间通信的电子邮件。但是，它提供了邮件服务没有的许多功能，例如，有保证的交付、事务、确认、使用内存的快捷模式等。如下一节所述，消息队列有许多可用于应用程序之间通信的特性。

通过消息队列功能，可以在打开连接或断开连接的环境下发送、接收和路由消息。图 46-1 显示了使用消息队列的一种非常简单的方式。发送者给消息队列发送消息，接收者从队列中接收消息。

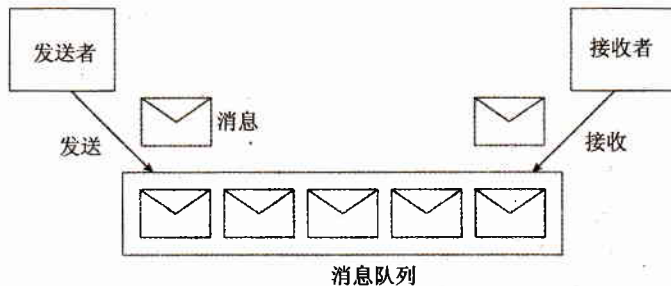


图 46-1

46.1.1 使用消息队列的场合

使用消息队列的一个场合是，客户端应用程序常常从网络上断开连接(例如，销售员在站点上访问顾客)。销售员可以直接在顾客的站点上输入订购数据。应用程序把每个订单的消息发送给位于客户端系统上的消息队列(如图 46-2 所示)。只要销售员回到办公室，订单就会自动从客户端系统的消息队列传输到目标系统的消息队列上，在目标系统上处理消息。

除了使用笔记本电脑之外，销售员还可以使用支持消息队列的 Pocket Windows 设备。

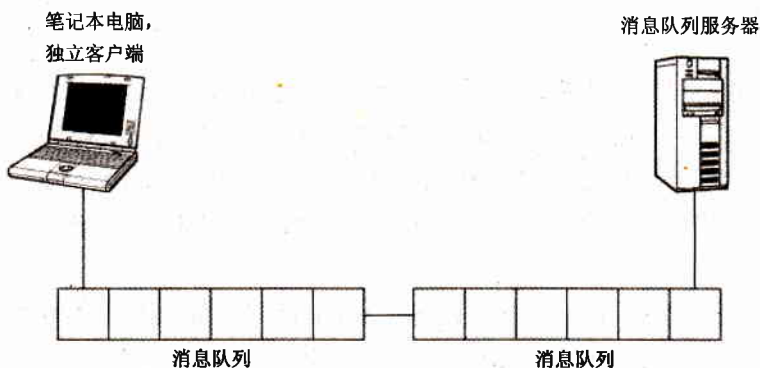


图 46-2

消息队列还可以在打开连接的环境下使用。假定在一个电子商务站点上(如图 46-3 所示)，其中服务器在某些时刻(如傍晚和周末)的订单事务负载是满的，但在晚上，其负载很低。一种解决方案是购买一台更快的服务器或在系统中添加更多服务器，以处理高峰时的订单。还有一种成本较低的解决方案：把事务从高负载的时段移动到低负载的时段，即削峰平谷。在这种方案中，把订单发送

到消息队列中,接收端按对数据库系统有利的速度读取订单。现在,系统的负载被平摊到各个时间段内,这样处理事务的服务器就可以比数据库系统升级的系统开销少。

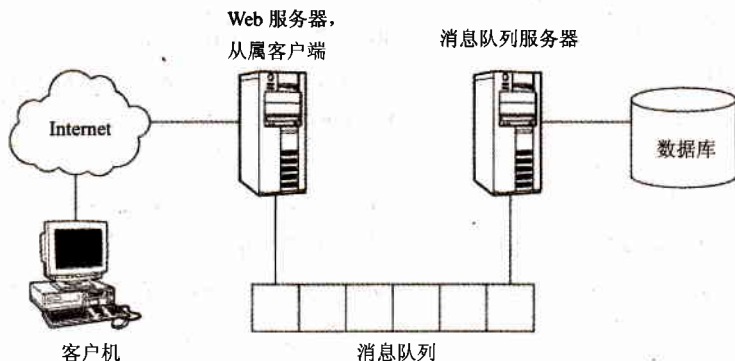


图 46-3

46.1.2 消息队列功能

消息队列是 Windows 操作系统的一部分。这个服务的主要功能如下:

- 消息可以在断开连接的环境下发送。不需要同时运行正在发送和正在接收的应用程序。
- 使用快捷模式,消息可以非常快地发送。在快捷模式下,消息存储在内存中。
- 对于可恢复的机制,消息可以使用有保证的交付方式发送。可恢复的消息存储在文件中。在服务器重新启动时发送它们。
- 用访问控制列表来保护消息队列,可以确定哪些用户可以发送或接收队列中的消息。消息还可以加密,避免网络嗅探器读取其中的数据。消息在发送时可以指定优先级,这样可以更快地处理高优先级的项。
- Message Queuing 3.0 支持多播消息的发送。
- Message Queuing 4.0 支持病毒消息。病毒消息不能解析。可以定义病毒队列中不能解析的消息是可以移动的。例如,如果从正常的队列中读取消息后,对应作业要把消息插入数据库中,但消息不能插入数据库,因此该作业失败,该消息就会发送到病毒队列中。有人负责处理病毒队列,这个人应以能解析病毒消息的方式来处理该消息。
- Message Queuing 5.0 支持更安全学身份验证算法,可以处理大量队列(Message Queuing 4.0 在处理几千个队列时有性能问题)。



因为消息队列是操作系统的一部分,所以不能在 Windows Server 2003 和 Windows XP 系统上安装 Message Queuing 5.0。Message Queuing 5.0 是 Windows Server 2008 R2 和 Windows 7 的一部分。

本章剩余的内容探讨这些功能的用法。

46.2 Message Queuing 产品

Message Queuing 5.0 是 Windows Server 2008 R2 和 Windows 7 的一部分。Windows 2000 和 Message Queuing 2.0 一起发布, 它不支持 HTTP 协议和多播消息。Message Queuing 3.0 是 Windows Server 2003 和 Windows XP 的一部分。Message Queuing 4.0 是 Windows Server 2008 和 Windows Vista 的一部分。

使用 Windows 7 的 Configuring Programs and Features 中的 Turn Windows Features on or off 链接时, 有一个单独用于 Message Queuing 选项的部分。在这个部分中, 可以选择如下组件:

- **Microsoft Message Queue(MSMQ) Server Core**——Core 子组件是消息队列基本功能所必需的。
- **Active Directory Domain Services Integration**——有了它, 消息队列名就会写入 Active Directory。利用这个选项可以使用 Active Directory Integration 查找队列, 用 Windows 用户和组保护队列。
- **MSMQ HTTP Support**——它允许使用 HTTP 协议发送和接收消息。
- **Triggers**——利用 Triggers, 可以在接收到新消息时实例化应用程序。
- **Multicast Support**——有了它, 就可以把一条消息发送给一组服务器。
- **MSMQ DCOM Proxy**——有了 DCOM 代理, 系统就可以使用 DCOM API 连接到远程服务器上。

在安装 Message Queuing 时, 必须启动 Message Queuing 服务(如图 46-4 所示)。这条服务读写消息, 与其他 Message Queuing 服务器通信, 把消息路由到网络上。

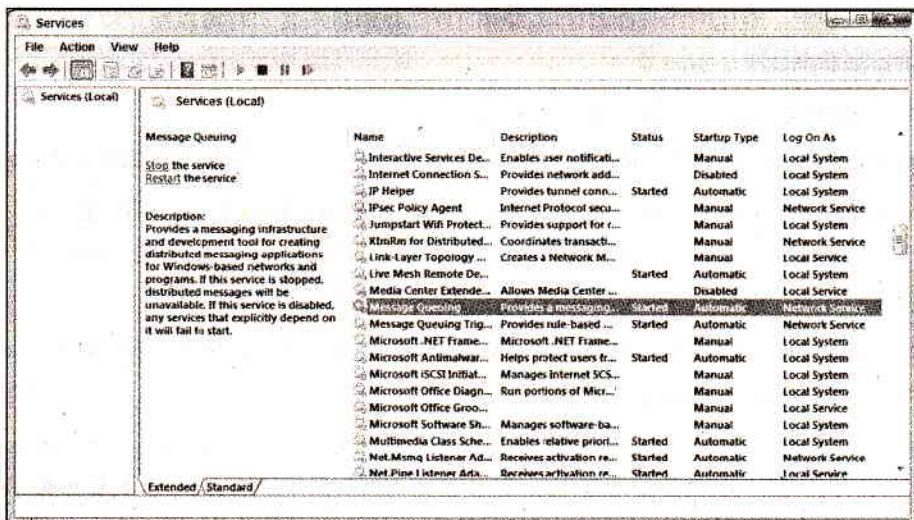


图 46-4

46.3 消息队列体系结构

通过消息队列, 可以从消息队列中读写消息。消息和消息队列有几个属性需要进一步阐述。

46.3.1 消息

把消息发送到消息队列中。消息包含正文(包含要发送的数据)和一个标签(消息的标题)。在消息的正文中可以放置任意信息。在.NET中,有几个格式化程序转换要放在正文中的数据。除了标签和正文之外,消息还包含发送者、超时配置、事务ID或优先级等信息。

消息队列有几种类型的消息:

- 一般消息——由应用程序发送。
- 确认消息——报告一般消息的状态。把确认消息发送到管理队列中,来报告一般消息的发送是否成功。
- 响应消息——当原始发送者需要某种特殊应答时,由接收应用程序发送响应消息。
- 报告消息——由消息队列系统生成。测试消息和路由跟踪消息属于此类。

消息可以有优先级,优先级定义从队列中读取消息的顺序。因为消息在队列中按照其优先级排序,所以从队列中读取的下一条消息就是优先级最高的那条消息。

消息有两种传递模式:快捷模式和可恢复模式。快捷消息的传送速度非常快,因为消息只使用消息存储器来存储。可恢复消息在路由的每一阶段都要存储在文件中,直到消息传递到目的地为止。这样,即使计算机重新启动或网络失败,消息的传递也能得到保证。

事务消息是可恢复消息的一种特殊版本。在事务消息传递过程中,可以确保消息只到达目的地一次,且按照它们发送的顺序到达目的地。优先级不能在事务消息中使用。

46.3.2 消息队列

消息队列是一个消息存储库。存储在磁盘上的消息位于<windir>\system32\msmq\storage目录。公共队列或私有队列通常用于发送消息,但还有其他队列类型:

- 公共队列在Active Directory中发布。这些队列的信息通过Active Directory域复制。可以使用浏览和搜索功能获得这些队列的信息。即使不知道放置队列的计算机名,也可以访问公共队列。还可以把这种队列从一个系统移动到另一个系统上,而无需通知客户。但不能在Workgroup环境下创建公共队列,因为需要Active Directory。Active Directory详见第52章(见光盘)。
- 私有队列不在Active Directory中发布。只有在知道队列的完整路径名时才能访问这些队列。私有队列可以在Workgroup环境下使用。
- 日志队列用于在发送或接收消息后,保存消息的副本。启动公共或私有队列的日志功能,就会自动创建一个日志队列。在日志队列中,可以有两种不同的队列类型:源日志队列和目标日志队列。通过消息的属性打开源日志功能,用源系统存储日志消息。用队列的属性打开目标日志功能,这些消息存储在目标系统的日志队列中。
- 如果消息没有在指定的超时前到达目标系统,该消息就存储在死信队列中。在同步编程中,错误会被立即检测出来,但使用消息队列处理错误的方式必须不同。死信队列可以用于检查未到达目的地的消息。

- 管理队列包含发送消息的确认消息。发送者可以指定一个管理队列，发送者从中接收消息是否成功发送的通知。
- 如果需要把多条简单的确认消息用作接收端的应答，就可以使用响应队列。接收应用程序可以把响应消息发送回原始发送者。
- 报告队列用于测试消息。把公共或私有队列的类型改为预定义的 ID{55EE8F33-CCE9-11CF-B108-0020AFD61CE9}，就可以创建报告队列。报告队列可以用作跟踪其路由中的消息的测试工具。
- 系统队列是私有的，由消息队列系统使用。这些队列用于管理消息，存储通知消息，保证事务消息的正确顺序。

46.4 Message Queuing 管理工具

在介绍如何以编程方式处理 Message Queuing 之前，本节先讨论 Windows 操作系统中的管理工具，以创建和管理队列和消息。



这里介绍的工具不仅能用于 Message Queuing。只有安装了 Message Queuing，才能使用这些工具的 Message Queuing 功能。

46.4.1 创建消息队列

消息队列可以使用 Computer Management MMC 插件创建。在 Windows 7 系统上，用 Start | Control Panel | Administrative Tools | Computer Management 菜单启动 Computer Management MMC 插件。在树型视图面板上，Message Queuing 位于 Services and Applications 项的下面。选择 Private Queues 或 Public Queues，就可以从 Action 菜单中创建新队列，如图 46-5 所示。只有在 Active Directory 模式下配置 Message Queuing，公共队列才可用。

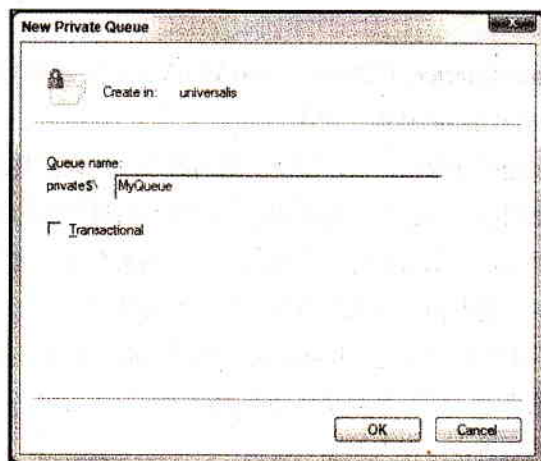


图 46-5

46.4.2 消息队列属性

在创建队列后, 可以使用 Computer Management 插件, 在树型面板上选择队列, 选择 Action | Properties 菜单, 来修改队列的属性, 如图 46-6 所示。

可以配置几个选项:

- Label 是队列的名称, 它可用于搜索队列。
- Type ID 默认设置为 {00000000-0000-0000-0000-000000000000}, 把多个队列映射到一个类别或类型上。报告队列使用特定的类型 ID, 如前所述。类型 ID 是通用唯一 ID(UUID)或 GUID。



图 46-6



可以使用 uuidgen.exe 或 guidgen.exe 实用程序创建自定义类型标识符。uuidgen.exe 是一个命令行实用程序, 用于创建唯一的 ID, guidgen.exe 是创建 UUID 的图形版本。

- 可以限制队列中所有消息的最大长度, 避免填满整个磁盘。
- 勾选 Authenticated 复选框时, Authenticated 选项只允许通过身份验证的用户读写队列中的消息。
- 使用 Privacy level 选项, 可以加密消息的内容。可以设置的值有 None、Optional 和 Body。None 表示不接收加密的消息, Body 只接收加密的消息, 默认的 Optional 值接收两者。
- 使用 Journal 设置可以配置目标日志功能。使用这个选项, 可以把接收的消息副本存储到日志中。可以为队列的日志消息配置能使用的磁盘空间的最大容量。当到达这个最大容量时, 就停止目标日志功能。
- 配置选项 Multicast 用于定义队列的多播 IP 地址。同一个多播 IP 地址可以用于网络上的不同节点, 这样发送给一个地址的消息就可以用多个队列接收。

46.5 消息队列的编程实现

既然理解了消息队列的体系结构之后, 就可以探讨其编程了。下面几节将学习如何创建和控制队列, 如何发送和接收消息。

还要构建一个小型课程订单应用程序, 它由发送部分和接收部分组成。

46.5.1 创建消息队列

前面了解了如何使用 Computer Management 实用程序创建消息队列。消息队列还可以用 MessageQueue 类的 Create() 方法以编程方式创建。

在 Create() 方法中, 必须传递新队列的路径。路径包括队列所在主机的名称和队列的名称。在下面的例子中, 要在本地主机上创建 MyNewPublicQueue 队列。为了创建私有队列, 路径名必须包含 Private\$, 如 \Private\$\MyNewPrivateQueue。

调用 `Create()` 方法之后, 就可以修改队列的属性。例如, 使用 `Label` 属性, 把队列的标签设置为 `Demo Queue`。示例程序把队列的路径和格式名写到控制台上。格式名用 `UUID` 自动创建, `UUID` 可用于访问队列, 且无须服务器名:



可从
wrox.com
下载源代码

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            using (var queue = MessageQueue.Create(@".\MyNewPublicQueue"))
            {
                queue.Label = "Demo Queue";
                Console.WriteLine("Queue created:");
                Console.WriteLine("Path: {0}", queue.Path);
                Console.WriteLine("FormatName: {0}", queue.FormatName);
            }
        }
    }
}
```

代码段 CreateMessageQueue/Program.cs



创建队列时需要管理权限。通常不希望应用程序的用户拥有管理权限。这就是队列通常用安装程序创建的原因。本章后面将介绍如何使用 `MessageQueueInstaller` 类创建消息队列。

46.5.2 查找队列

路径名和格式名可以用于标识队列。要查找队列, 必须区分公共队列和私有队列。公共队列在 `Active Directory` 中发布。对于这些队列, 无须知道它们所在的系统。只有在已知队列所在系统私有队列名称时才能找到私有队列。

在 `Active Directory` 域中搜索队列的标签、类别或格式名, 就可以找到公共队列。还可以获得计算机上的所有队列。`MessageQueue` 类的静态方法 `GetPublicQueuesByLabel()`、`GetPublicQueuesByCategory()` 和 `GetPublicQueuesByMachine()` 可以搜索队列。`GetPublicQueues()` 方法返回包含域中所有公共队列的数组。



可从
wrox.com
下载源代码

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {

```

```

static void Main()
{
    foreach (var queue in MessageQueue.GetPublicQueues())
    {
        Console.WriteLine(queue.Path);
    }
}

```

代码段 FindQueues/Program.cs

GetPublicQueues()方法是重载的。它的一个重载版本允许传递 MessageQueueCriteria 类的一个实例。利用这个类可以搜索在某个时刻之前或之后创建或修改的队列，还可以查找队列的类别、标签或计算机名。

可以使用静态方法 GetPrivateQueuesByMachine()搜索私有队列。这个方法返回指定系统中的所有私有队列。

46.5.3 打开已知队列

如果队列名已知，就不需要搜索它。使用路径或格式名就可以打开队列。路径或格式名都在 MessageQueue 类的构造函数中设置。

1. 路径名

路径指定了打开队列需要的计算机名和队列名。下面的代码示例打开本地主机上的 MyPublicQueue 队列。为了确定队列是否存在，可以使用静态方法 MessageQueue.Exists()：



可从
wrox.com
下载源代码

```

using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            if (MessageQueue.Exists(@".\MyPublicQueue"))
            {
                var queue = new MessageQueue(@".\MyPublicQueue");
                //.
            }
            else
            {
                Console.WriteLine("Queue .\MyPublicQueue not existing");
            }
        }
    }
}

```

代码段 OpenQueue/Program.cs

根据队列的类型，在打开队列时需要不同的标识符。表 46-1 列出了指定类型的队列名的语法。

表 46-1

队 列 类 型	语 法
公共队列	MachineName\QueueName
私有队列	MachineName\Private\$\QueueName
日志队列	MachineName\QueueName\Journal\$
计算机日志队列	MachineName\Journal\$
计算机死信队列	MachineName\DeadLetter\$
计算机事务死信队列	MachineName\XactDeadLetter\$

在使用路径名打开公共队列时，需要传递计算机名。如果计算机名未知，则可以使用格式名代替。私有队列的路径名只能在本地系统上使用，必须使用格式名远程访问私有队列。

2. 格式名

除了路径名之外，还可以使用格式名打开队列。格式名用于在 Active Directory 中搜索队列，获得队列所在的主机。在断开连接的环境下，在发送消息时队列不能到达，此时就需要使用格式名：

```
MessageQueue queue = new MessageQueue(
    @"FormatName:PUBLIC=09816AFF-3608-4c5d-B892-69754BA151FF");
```

格式名还有一些其他用途。它可以用于打开私有队列，并指定要使用的协议：

- 要访问私有队列，必须给构造函数传递字符串 `FormatName:PRIVATE=MachineGUID\QueueNumber`。在创建队列时，会生成私有队列的队列号。队列号在<windows>\System32\msmq\storage\lqs 目录下。
- 使用 `FormatName:DIRECT=Protocol:MachineAddress\QueueName`，可以指定用于发送消息的协议。Message Queuing 3.0 及以后版本支持 HTTP 协议。
- `FormatName:DIRECT=OS:MachineName\QueueName` 是使用格式名指定队列的另一种方式。此时不需要指定协议，但仍可以使用计算机名和格式名。

46.5.4 发送消息

可以使用 MessageQueue 类的 Send()方法给队列发送消息。作为参数传递给 Send()方法的对象序列化到相关联的队列上。Send()方法是重载的，这样才能传递标签和 MessageQueueTransaction 对象。Message Queuing 的事务行为在后面论述。

下面的代码示例先检查队列是否存在，如果不存在，就创建一个队列。接着打开队列，使用 Send()方法给队列发送 Sample Message 消息。

路径名给服务器名指定“.”，表示它是本地系统。私有队列的路径名只能在本地使用。



可从
wrox.com
下载源代码

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
```

```

    {
        static void Main()
        {
            try
            {
                if (!MessageQueue.Exists(@".\Private$\MyPrivateQueue"))
                {
                    MessageQueue.Create(@".\Private$\MyPrivateQueue");
                }
                var queue = new MessageQueue(@".\Private$\MyPrivateQueue");

                queue.Send("Sample Message", "Label");
            }
            catch (MessageQueueException ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}

```

代码段 SendMessage/Program.cs

图 46-7 显示了 Computer Management 管理工具，其中可以看到到达队列的消息。

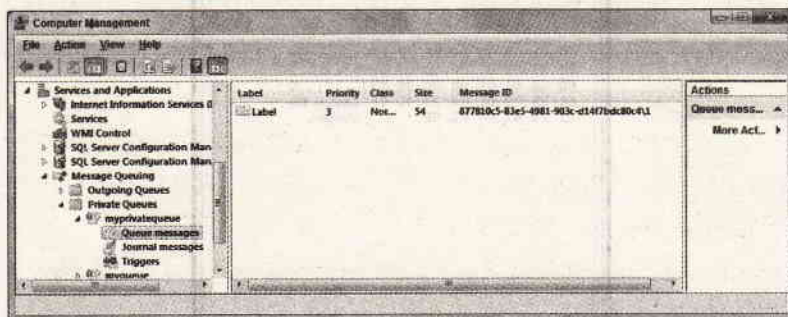


图 46-7

打开消息，选择对话框的 Body 选项卡(如图 46-8 所示)，就可以看到用 XML 格式化后的消息。判断消息的格式化方式是消息队列相关联的格式化程序的功能。

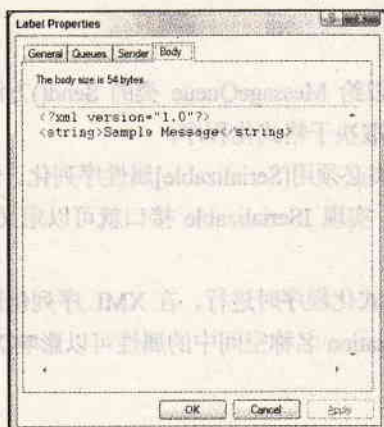


图 46-8

1. 消息格式化程序

消息传输给队列的格式取决于格式化程序。`MessageQueue` 类有一个 `Formatter` 属性, 通过它可以指定格式化程序。默认的格式化程序 `XmlMessageFormatter` 会用 XML 语法格式化消息, 如前面的例子所示。

消息格式化程序实现 `IMessageFormatter` 接口。`System.Messaging` 名称空间中有 3 个消息格式化程序:

- `XmlMessageFormatter` 是默认的格式化程序, 它使用 XML 序列化对象, XML 格式的内容详见第 33 章。
- 使用 `BinaryMessageFormatter`, 可以用二进制格式对消息进行序列化。这些消息比使用 XML 格式化的消息短。
- `ActiveXMessageFormatter` 是一个二进制格式化程序, 这样可以用 COM 对象读写消息。使用这个格式化程序, 可以用 .NET 类把消息写入队列中, 使用 COM 对象从队列中读取消息, 反之亦然。

图 46-8 中用 XML 显示的示例消息是用图 46-9 中的 `BinaryMessageFormatter` 格式化的。

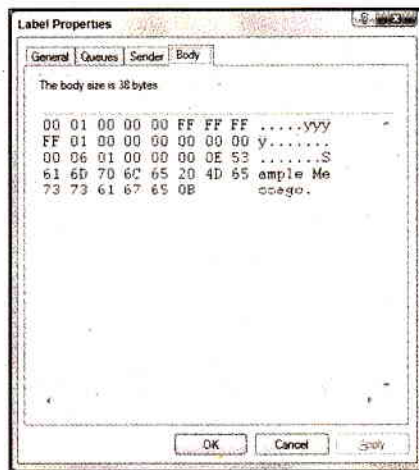


图 46-9

2. 发送复杂的消息

除了传递字符串之外, 还可以给 `MessageQueue` 类的 `Send()` 方法传递对象。虽然该类的类型必须满足一些特定的要求, 但它们取决于格式化程序。

对于二进制格式化程序, 该类必须用 `[Serializable]` 属性序列化。使用 .NET 运行库的序列化功能, 序列化所有字段(包括私有字段)。实现 `ISerializable` 接口就可以定义自定义序列化。 .NET 运行库的序列化功能详见第 29 章。

XML 序列化在使用 XML 格式化程序时进行。在 XML 序列化过程中, 会序列化所有公共字段和属性。使用 `System.Xml.Serialization` 名称空间中的属性可以影响 XML 序列化。XML 序列化详见第 33 章。

46.5.5 接收消息

要读取消息，也可以使用 `MessageQueue` 类。通过 `Receive()` 方法可以读取一条消息，再将该消息从队列中删除。如果使用不同的优先级发送消息，就读取优先级最高的消息。读取优先级相同的消息时，第一条发送的消息不一定是第一条读取的消息，因为消息在网络中的传递顺序无法保证。要保证发送顺序和读取顺序相同，可以使用事务消息队列。

在下面的例子中，要从私有队列 `MyPrivateQueue` 中读取一条消息。之前把一个简单的字符串传递给该消息。在使用 `XmlMessageFormatter` 格式化程序读取消息时，必须把要读取的对象的类型传递给该格式化程序的构造函数。在本例中，将 `System.String` 类型传递给 `XmlMessageFormatter` 的构造函数的参数数组。这个构造函数可以接收一个 `String` 数组，该数组包含要作为字符串传递的类型；也可以接收一个 `Type` 数组。

用 `Receive()` 方法读取消息，再把消息正文写入控制台中：



可从
wrox.com
下载源代码

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
            queue.Formatter = new XmlMessageFormatter(
                new string[] { "System.String" });

            Message message = queue.Receive();
            Console.WriteLine(message.Body);
        }
    }
}
```

代码段 `SendMessage/Program.cs`

`Receive()` 方法将同步执行，如果队列中没有消息，它就会等待队列中有消息时再执行。

1. 枚举消息

除了使用 `Receive()` 方法逐条消息地读取之外，还可以使用枚举器遍历所有消息。因为 `MessageQueue` 类实现 `IEnumerable` 接口，所以可以在 `foreach` 语句中使用。使用迭代器时，虽然消息不会从队列中删除，但可以查看消息从而获得它们的内容：

```
var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
    new string[] { "System.String" });

foreach (Message message in queue)
{
    Console.WriteLine(message.Body);
}
```

除了使用 `IEnumerable` 接口外，还可以使用 `MessageEnumerator` 类。虽然 `MessageEnumerator` 类

实现 `IEnumerator` 接口,但它有更多功能。实现 `IEnumerable` 接口,就表示不从队列中删除消息。`MessageEnumerator` 类的 `RemoveCurrent()`方法可以从枚举器的当前光标位置删除消息。

在下面的例子中,使用 `MessageQueue` 类的 `GetMessageEnumerator()`方法访问 `MessageEnumerator` 类。通过 `MessageEnumerator` 类的 `MoveNext()`方法,可以逐条查看消息。`MoveNext()`方法重载为允许把一个时间段作为参数。这是使用这个枚举器的一个主要优点。现在,线程可以在指定的时间段内等待消息到达队列,之后就不等待了。`IEnumerator` 接口定义的 `Current` 属性返回消息的一个引用:

```
var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
    new string[] { "System.String" });

using (MessageEnumerator messages = queue.GetMessageEnumerator())
{
    while (messages.MoveNext(TimeSpan.FromMinutes(30)))
    {
        Message message = messages.Current;
        Console.WriteLine(message.Body);
    }
}
```

2. 异步读取

`MessageQueue` 类的 `Receive()`方法会等到队列中的消息可以读取为止。为了避免阻碍线程的执行,可以在 `Receive()`方法的一个重载版本中指定一个超时期限。要在超时后读取队列中的消息,必须再次调用 `Receive()`方法。除了轮询消息外,还可以调用 `BeginReceive()`异步方法。在使用 `BeginReceive()`开始异步读取消息之前,应设置 `ReceiveCompleted` 事件。`ReceiveCompleted` 事件需要 `ReceiveCompletedEventHandler` 委托,在消息到达队列并可以读取时该委托引用要调用的方法。在下面的例子中,把 `MessageArrived()`方法传递给 `ReceivedCompletedEventHandler` 委托:



可从
wrox.com
下载源代码

```
var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
    new string[] { "System.String" });

queue.ReceiveCompleted += MessageArrived;
queue.BeginReceive();
// thread does not wait
```

代码段 `ReceiveMessageAsync/Program.cs`

`MessageArrived()`处理程序方法需要两个参数。第一个参数是 `MessageQueue` 事件源。第二个参数是 `ReceiveCompletedEventArgs` 类型,它包含消息和异步结果。在下面的例子中,调用队列中的 `EndReceive()`方法,以获得异步方法的结果,即消息:

```
public static void MessageArrived(object source, ReceiveCompletedEventArgs e)
{
    MessageQueue queue = (MessageQueue)source;
    Message message = queue.EndReceive(e.AsyncResult);
    Console.WriteLine(message.Body);
}
```

如果不应从队列中删除消息, `BeginPeek()` 和 `EndPeek()` 方法就可以与异步 I/O 一起使用。

46.6 课程订单应用程序

为了演示消息队列的用法, 本节将创建一个示例解决方案, 用于订购课程。示例解决方案由 3 个程序集组成:

- 组件库(`CourseOrder`), 它包含在队列中发送和接收的消息的实体类
- WPF 应用程序(`CourseOrderSender`), 它给消息队列发送消息
- WPF 应用程序(`CourseOrderReceiver`), 它从消息队列中接收消息

46.6.1 课程订单类库

发送和接收应用程序都需要订单信息。所以, 把实体类放在一个单独的程序集中。`CourseOrder` 程序集包含 3 个实体类: `CourseOrder`、`Course` 和 `Customer`。在示例应用程序中, 并不像实际应用程序中的实现方式那样实现所有属性, 而只实现用来揭示概念的属性。

在 `Course.cs` 文件中定义 `Course` 类。这个类只有一个属性, 表示课程的名称:



```
namespace Wrox.ProCSharp.Messaging
{
    public class Course
    {
        public string Title { get; set; }
    }
}
```

代码段 `CourseOrder/Course`

`Customer.cs` 文件包含 `Customer` 类, 该类的属性用于表示公司名称和联系人姓名:



```
namespace Wrox.ProCSharp.Messaging
{
    public class Customer
    {
        public string Company { get; set; }
        public string Contact { get; set; }
    }
}
```

代码段 `CourseOrder/Customer.cs`

在 `CourseOrder.cs` 文件中的 `CourseOrder` 类映射订单中的一个顾客和一门课程, 并确定订单是否有高优先级。这个类还定义了队列名, 把队列名设置为公共队列的格式名。即使当前不能访问队列, 也可以使用格式名发送消息。我们可以使用 `Computer Management` 插件获得格式名, 来读取消息队列的 ID。如果不能访问 `Active Directory` 来创建公共队列, 就可以轻松地修改代码来使用私有队列。



```
namespace Wrox.ProCSharp.Messaging
{
    public class CourseOrder
    {
    }
}
```

```

public const string CourseOrderQueueName =
"FormatName:Public=D99CE5F3-4282-4a97-93EE-E9558B15EB13";

public Customer Customer { get; set; }
public Course Course { get; set; }
}

```

代码段 CourseOrder/CourseOrder.cs

46.6.2 课程订单消息发送程序

解决方案的第二部分是一个 Windows 应用程序 `CourseOrderSender`。在这个应用程序中,把课程订单发送给消息队列。必须引用 `System.Messaging` 和 `CourseOrder` 程序集。

这个应用程序的用户界面如图 46-10 所示。`comboBoxCourses` 组合框的项包括几门课程,例如, `Advanced .NET Programming`、`Programming with LINQ` 和 `Distributed Application Development using WCF`。

单击 `Submit the Order` 按钮,就调用 `buttonSubmit_Click()` 处理程序方法。在这个方法中,创建一个 `CourseOrder` 对象,并用 `TextBox` 和 `ComboBox` 控件的内容填充它。接着创建一个 `MessageQueue` 实例,以打开带有格式名的公共队列。使用 `Send()` 方法,传递 `CourseOrder` 对象,用默认的 `XmlMessageFormatter` 序列化它,再把它写入队列中:

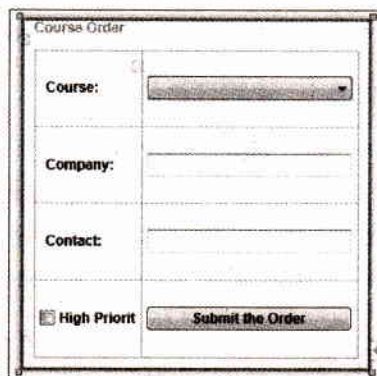


图 46-10



可从
wrox.com
下载源代码

```

private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        var order = new CourseOrder();
        order.Course = new Course()
        {
            Title = comboBoxCourses.SelectedItem.ToString()
        };
        order.Customer = new Customer()
        {
            Company = textCompany.Text,
            Contact = textContact.Text
        };

        using (var queue = new MessageQueue(CourseOrder.CourseOrderQueueName))
        {
            queue.Send(order, String.Format("Course Order {0}",
                order.Customer.Company));
        }
        MessageBox.Show("Course Order submitted", "Course Order",
            MessageBoxButton.OK, MessageBoxImage.Information);
    }
    catch (MessageQueueException ex)
    {
        MessageBox.Show(ex.Message, "Course Order Error",

```

```
MessageBoxButton.OK, MessageBoxImage.Error);
```

代码段 CourseOrderSender/CourseOrderWindow.xaml.cs

46.6.3 发送优先级和可恢复的消息

通过设置 Message 类的 Priority 属性, 就可以给消息指定优先级。如果消息是特别配置的, 就必须创建一个 Message 对象, 其中消息的正文在构造函数中传递。

在这个例子中, 如果选中 checkBoxPriority 复选框, 优先级就设置为 MessagePriority.High。MessagePriority 是一个枚举, 可以把值设置为从 Lowest (0) 到 Highest (7), 默认值 Normal 的优先级是 3。

为了使消息可以恢复, 应把 Recoverable 属性设置为 true:



可从
wrox.com
下载源代码

```
private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        var order = new CourseOrder
        {
            Course = new Course
            {
                Title = comboBoxCourses.Text
            },
            Customer = new Customer
            {
                Company = textCompany.Text,
                Contact = textContact.Text
            }
        };

        using (var queue = new MessageQueue(CourseOrder.CourseOrderQueueName))
        using (var message = new Message(order))
        {
            if (checkBoxPriority.IsChecked == true)
            {
                message.Priority = MessagePriority.High;
            }

            message.Recoverable = true;
            queue.Send(message, String.Format("Course Order {{0}}",
                order.Customer.Company));
        }

        MessageBox.Show("Course Order submitted");
    }
    catch (MessageQueueException ex)
    {
        MessageBox.Show(ex.Message, "Course Order Error",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

代码段 CourseOrderSender/CourseOrderWindow.xaml.cs

运行应用程序，就可以把课程订单添加到消息队列中，如图 46-11 所示。

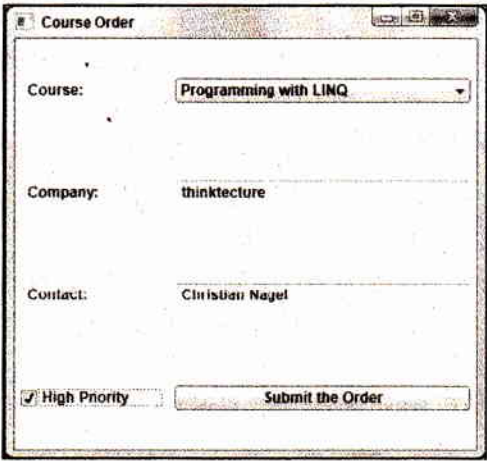


图 46-11

46.6.4 课程订单消息接收程序

课程订单接收应用程序从队列中读取消息，其设计视图如图 46-12 所示。这个应用程序在 `listOrders` 列表框中显示每个订单的标签。选中一个订单后，订单的内容就会显示在应用程序右边的控件中。

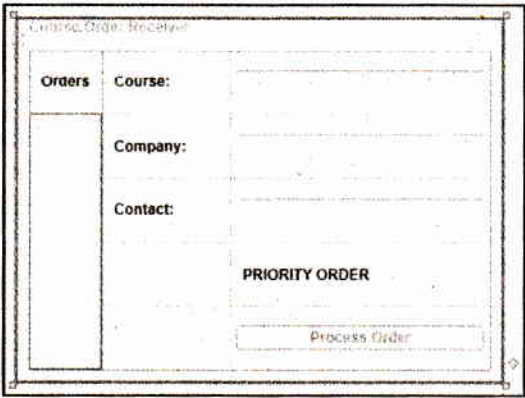


图 46-12

在窗体类 `CourseOrderReceiverWindow` 的构造函数中，创建一个 `MessageQueue` 对象，它引用由发送应用程序使用的队列。要读取消息，应使用 `Formatter` 属性把 `XmlMessageFormatter`、读取的类型和队列关联起来。

要在列表中显示可用的消息，应创建一个新任务，该任务在后台中读取消息。该任务的主方法是 `PeekMessages()`。

 任务的更多内容详见第 30 章。



可从
wrox.com
下载源代码

```
using System;
using System.Messaging;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Threading;

namespace Wrox.ProCSharp.Messaging
{
    public partial class CourseOrderReceiverWindow: Window
    {
        private MessageQueue orderQueue;

        public CourseOrderReceiverWindow()
        {
            InitializeComponent();

            string queueName = CourseOrder.CourseOrderQueueName;
            orderQueue = new MessageQueue(queueName);
            orderQueue.Formatter = new XmlMessageFormatter(
                new Type[]
                {
                    typeof(CourseOrder),
                    typeof(Customer),
                    typeof(Course)
                });

            // start the task that fills the ListBox with orders
            Task t1 = new Task(PeekMessages);
            t1.Start();
        }
    }
}
```

代码段 CourseOrderReceiver/CourseOrderReceiverWindow.xaml.cs

任务的主方法 `PeekMessages()` 使用消息队列的枚举器显示所有消息。在 `while` 循环中，`MessageEnumerator` 检查队列中是否还有新消息。如果队列中没有消息，线程就等待下一条消息进入队列中，等待 3 个小时后它才会退出。

要在列表框中显示队列中的每条消息，虽然线程不能直接把文本写入列表框中，但必须把调用传递给列表框的创建线程。因为 WPF 窗体控件绑定到一个线程上，所以只能使用该创建线程访问方法和属性。`Dispatcher.Invoke()` 方法把请求写入创建线程中：

```
private void PeekMessages()
{
    using (MessageEnumerator messagesEnumerator =
        orderQueue.GetMessageEnumerator2())
    {
        while (messagesEnumerator.MoveNext(TimeSpan.FromHours(3)))
        {
            var labelId = new LabelIdMapping()
            {
                Id = messagesEnumerator.Current.Id,
                Label = messagesEnumerator.Current.Label
            };
            Dispatcher.Invoke(DispatcherPriority.Normal,
```

```

        new Action<LabelIdMapping> (AddListItem),
        labelId);
    }
    MessageBox.Show("No orders in the last 3 hours. Exiting thread",
        "Course Order Receiver", MessageBoxButton.OK,
        MessageBoxImage.Information);
}
private void AddListItem(LabelIdMapping labelIdMapping)
{
    listOrders.Items.Add(labelIdMapping);
}

```

ListBox 控件包含 LabelIdMapping 类的元素。虽然这个类用于在列表框中显示消息的标签，但这里它隐藏消息的 ID。消息的 ID 可以用于以后读取消息：

```

private class LabelIdMapping
{
    public string Label { get; set; }
    public string Id { get; set; }

    public override string ToString()
    {
        return Label;
    }
}

```

ListBox 控件的 SelectedIndexChanged 事件与 OnOrders_SelectionChanged() 方法相关联。这个方法从当前选中的项中获得 LabelIdMapping 对象，并使用 ID 通过 PeekById() 方法再次查看消息。接着在 TextBox 控件中显示消息的内容。因为默认情况下不读取消息的优先级，所以 MessageReadPropertyFilter 属性必须设置为接收 Priority：

```

private void listOrders_SelectionChanged(object sender,
    RoutedEventArgs e)
{
    LabelIdMapping labelId = listOrders.SelectedItem as LabelIdMapping;
    if (labelId == null)
        return;

    orderQueue.MessageReadPropertyFilter.Priority = true;
    Message message = orderQueue.PeekById(labelId.Id);

    CourseOrder order = message.Body as CourseOrder;
    if (order != null)
    {
        textCourse.Text = order.Course.Title;
        textCompany.Text = order.Customer.Company;
        textContact.Text = order.Customer.Contact;
        buttonProcessOrder.IsEnabled = true;

        if (message.Priority > MessagePriority.Normal)
        {
            labelPriority.Visibility = Visibility.Visible;
        }
        else
    }
}

```

```

        {
            labelPriority.Visibility = Visibility.Hidden;
        }
    }
    else
    {
        MessageBox.Show("The selected item is not a course order",
            "Course Order Receiver", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}

```

单击 **Process Order** 按钮时，会调用 `OnProcessOrder()` 处理程序方法。这里会再次引用列表框中当前选择的消息，并调用 `ReceiveById()` 方法从队列中删除该消息：

```

private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
    LabelIdMapping labelId = listOrders.SelectedItem as LabelIdMapping;
    Message message = orderQueue.ReceiveById(labelId.Id);

    listOrders.Items.Remove(labelId);
    listOrders.SelectedIndex = -1;
    buttonProcessOrder.IsEnabled = false;
    textCompany.Text = string.Empty;
    textContact.Text = string.Empty;
    textCourse.Text = string.Empty;

    MessageBox.Show("Course order processed", "Course Order Receiver",
        MessageBoxButton.OK, MessageBoxImage.Information);
}

```

图 46-13 显示了正在运行的接收程序，该程序列出了队列中的 3 个订单，且当前选中了一个订单。

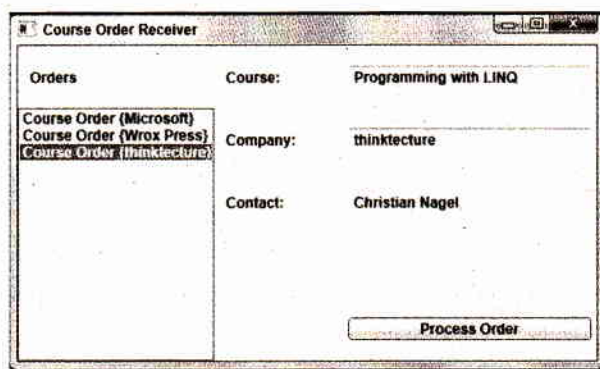


图 46-13

46.7 接收结果

在示例应用程序的当前版本中，发送应用程序并不知道消息是否已处理。为了得到接收程序的

结果，可以使用确认队列或响应队列。

46.7.1 确认队列

使用确认队列，发送应用程序可以获得消息的状态信息。在确认消息中，可以定义是否要接收应答，来判断是一切正常，还是某些地方出错。例如，可以在消息到达目标队列或读取消息时，发送确认消息；或在指定的超时过后，消息未到达目标队列或未读取消息时，发送确认消息。

在下面的例子中，将 `Message` 类的 `AdministrationQueue` 设置为 `CourseOrderAck` 队列，这个队列的创建方式类似于一般队列，但它以另外一种方式使用：原始发送程序接收到确认消息时使用它。把 `AcknowledgementType` 属性设置为 `AcknowledgementTypes.FullReceive`，会在读取消息时获得一条确认消息：

```
Message message = new Message(order);

message.AdministrationQueue = new MessageQueue(@".\CourseOrderAck");
message.AcknowledgeType = AcknowledgementTypes.FullReceive;

queue.Send(message, String.Format("Course Order {{0}}",
    order.Customer.Company));

string id = message.Id;
```

关联 ID 用于确定什么确认消息属于发送的哪条消息。发送的每条消息都有一个 ID，响应该消息所发送的确认消息把源消息的 ID 作为其关联 ID。可以用 `MessageQueue.ReceiveByCorrelationId()` 方法读取确认队列中的消息，以接收相关的确认消息。

除了使用确认消息之外，还可以为没有到达其目的地的消息使用死信队列。把 `Message` 类的属性 `UseDeadLetterQueue` 设置为 `true`，如果消息在超时过后没有到达目标队列，该消息就会复制到死信队列中。

超时用 `Message` 类的 `TimeToReachQueue` 和 `TimeToBeReceived` 属性设置。

46.7.2 响应队列

如果需要从接收程序中获得比确认消息更多的信息，就可以使用响应队列。响应队列类似于一般队列，但原始发送程序把该队列用作接收程序，原始接收程序把响应队列用作发送程序。

发送程序必须用 `Message` 类的 `ResponseQueue` 属性指定响应队列。下面的示例代码揭示了接收程序如何使用响应队列返回一条响应消息。在响应消息 `responseMessage` 中，把 `CorrelationId` 属性设置为原始消息的 ID。这样，客户端应用程序就知道该应答属于哪条消息。这类似于确认队列。响应消息用 `MessageQueue` 对象的 `Send()` 方法发送，`MessageQueue` 对象从 `ResponseQueue` 属性中返回：

```
public void ReceiveMessage(Message message)
{
    Message responseMessage = new Message("response");
    responseMessage.CorrelationId = message.Id;

    message.ResponseQueue.Send(responseMessage);
}
```

46.8 事务队列

对于可恢复的消息，不能保证消息的到达顺序，也不能保证消息只到达一次。网络失败可能会使消息到达多次，如果发送程序和接收程序安装了供消息队列使用的多个网络协议，那么也会发生这种情况。

在需要确保如下条件的情况下，可以使用事务队列：

- 消息的到达顺序与其发送顺序相同。
- 消息只到达一次。

对于事务队列，一个事务不能横跨消息的发送和接收过程。消息队列的本质是发送和接收的时间间隔可能非常长。而事务应该很短。在消息队列中，第一个事务用于把消息发送到队列中，第二个事务用于把消息转发到网络上，第三个事务用于接收消息。

下一个例子揭示了如何创建事务消息队列，如何使用事务发送消息。

给 `MessageQueue.Create()` 方法的第二个参数传递 `true`，就会创建事务消息队列。

如果要在一个事务中把多条消息写入队列中，就必须实例化 `MessageQueueTransaction` 对象，并调用 `Begin()` 方法。在发送完属于该事务的所有消息后，必须调用 `MessageQueueTransaction` 对象的 `Commit()` 方法。要取消一个事务(且不把消息写入队列中)，就必须在 `catch` 块中调用 `Abort()` 方法，如下所示：

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            if (!MessageQueue.Exists(@".\MyTransactionalQueue"))
            {
                MessageQueue.Create(@".\MyTransactionalQueue", true);
            }

            var queue = new MessageQueue(@".\MyTransactionalQueue");
            var transaction = new MessageQueueTransaction();

            try
            {
                transaction.Begin();
                queue.Send("a", transaction);
                queue.Send("b", transaction);
                queue.Send("c", transaction);
                transaction.Commit();
            }
            catch
            {
                transaction.Abort();
            }
        }
    }
}
```

46.9 消息队列和 WCF

第 43 章介绍了 WCF 的体系结构和核心功能。使用 WCF 可以配置一个使用 Windows Message Queuing 体系结构的消息队列绑定。WCF 通过它为消息队列提供了一个抽象层。图 46-14 用一幅简单的图片解释了该体系结构。客户端应用程序调用 WCF 代理的一个方法，来给队列发送一条消息。该消息由代理创建：客户端开发人员不需要知道消息发送给了队列，只调用代理的方法即可。代理抽象了处理 System.Messaging 名称空间中的类的过程，并给队列发送一个消息。服务器端的 MSMQ 侦听器信道从队列中读取消息，把它们转换为方法调用，再用服务调用这些方法调用。

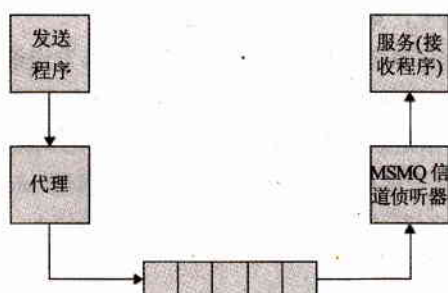


图 46-14

接着，从 WCF 的角度来看，把课程预订应用程序转换为使用消息队列。在这个解决方案中，修改前面的 3 个项目，再添加一个程序集，其中包含 WCF 服务的协定。

- 组件库(CourseOrder)包含在网络上发送消息的实体类。这些实体类改为满足数据协定的要求，来与 WCF 进行序列化。
- 添加一个新库(CourseOrderService)，它定义了服务提供的协定。
- 修改 WPF 发送应用程序(CourseOrderSender)，不发送消息，而调用 WCF 代理的方法。
- 修改 WPF 接收应用程序(CourseOrderReceiver)，使用实现了协定的 WCF 服务。

46.9.1 带数据协定的实体类

在 CourseOrder 库中，修改 Course、Customer 和 CourseOrder 类，以应用数据协定和属性 [DataContract] 及 [DataMember]。要使用这些属性，必须引用 System.Runtime.Serialization 程序集，并导入 System.Runtime.Serialization 名称空间。



```
using System.Runtime.Serialization;

namespace Wrox.ProCSharp.Messaging
{
    [DataContract]
    public class Course
    {
        [DataMember]
        public string Title { get; set; }
    }
}
```

}

代码段 CourseOrder/Course.cs

Customer 类还需要数据协定属性:



可从
wrox.com
下载源代码

```
[DataContract]
public class Customer
{
    [DataMember]
    public string Company { get; set; }

    [DataMember]
    public string Contact { get; set; }
}
```

代码段 CourseOrder/Customer.cs

对于 CourseOrder 类, 不仅要添加数据协定属性, 还要添加 ToString() 方法的一个重写版本, 以及包含这些对象的默认字符串表示:



可从
wrox.com
下载源代码

```
[DataContract]
public class CourseOrder
{
    [DataMember]
    public Customer Customer { get; set; }

    [DataMember]
    public Course Course { get; set; }
    public override string ToString()
    {
        return String.Format("Course Order {{0}}", Customer.Company);
    }
}
```

代码段 CourseOrder/CourseOrder.cs

46.9.2 WCF 服务协定

为了给服务提供 WCF 服务协定, 需要添加一个 WCF 服务库 CourseOrderServiceContract。这个协定由 ICourseOrderService 接口定义。该协定需要 [ServiceContract] 特性。如果希望仅把这个接口用于消息队列, 就可以应用 [DeliveryRequirements] 特性, 并指定 QueuedDeliveryRequirements 属性。QueuedDeliveryRequirementsMode 枚举的值有 Required、Allowed 和 NotAllowed。AddCourseOrder() 方法由服务提供。消息队列使用的方法只能有输入参数。因为发送程序和接收程序都可以彼此独立地运行, 所以发送程序不能期望立即得到结果。使用 [OperationContract] 属性设置 IsOneWay 特性。这个操作的调用者不等待服务的应答:



可从
wrox.com
下载源代码

```
using System.ServiceModel;

namespace Wrox.ProCSharp.Messaging
{
    [ServiceContract]
    [DeliveryRequirements(
        QueuedDeliveryRequirements=QueuedDeliveryRequirementsMode.Required)]
    public interface ICourseOrderService
```

```
[OperationContract(IsOneWay = true)]
void AddCourseOrder(CourseOrder courseOrder);
```

代码段 CourseOrderServiceContract/ICourseOrderService.cs



可以使用确认队列和响应队列给客户供应答。

46.9.3 WCF 消息接收应用程序

WPF 应用程序 CourseOrderReceiver 现在修改为实现 WCF 服务，并接收消息。它需要引用 System.ServiceModel 程序集和 WCF 协定程序集 CourseOrderServiceContract。

CourseOrderService 类实现 ICourseOrderService 接口。在实现代码中，触发 CourseOrderAdded 事件。WPF 应用程序注册这个事件，来接收 CourseOrder 对象。

因为把 WPF 控件绑定到单个线程上，所以 UseSynchronizationContext 属性用[ServiceBehavior] 特性设置。这是 WCF 运行库的一个功能，可以把方法调用传递给 WPF 应用程序的同步上下文定义的线程。



可从
wrox.com
下载源代码

```
using System.ServiceModel;

namespace Wrox.ProCSharp.Messaging
{
    [ServiceBehavior(UseSynchronizationContext=true)]
    public class CourseOrderService: ICourseOrderService
    {
        public static event EventHandler<CourseOrderEventArgs>
            CourseOrderAdded;

        public void AddCourseOrder(CourseOrder courseOrder)
        {
            if (CourseOrderAdded != null)
                CourseOrderAdded(this, new CourseOrderEventArgs(courseOrder));
        }
    }

    public class CourseOrderEventArgs : EventArgs
    {
        public CourseOrderEventArgs(CourseOrder courseOrder)
        {
            this.CourseOrder = courseOrder;
        }
        public CourseOrder CourseOrder { get; private set; }
    }
}
```

代码段 CourseOrderReceiver/CourseOrderService.cs



第 20 章介绍了同步上下文。

在 `CourseReceiverWindow` 类的构造函数中, 实例化一个 `ServiceHost` 对象, 并打开它, 以启动侦听器。侦听器的绑定在应用程序配置文件中完成。

在构造函数中, 订阅 `CourseOrderReceiver` 类的 `CourseOrderAdded` 事件。因为这里只需把接收到的 `CourseOrder` 对象添加到集合中, 所以使用了一个简单的 Lambda 表达式。



第8章介绍了 Lambda 表达式。

这里使用的集合类是 `System.Collections.ObjectModel` 名称空间中的 `ObservableCollection<T>`。由于这个集合类实现 `INotifyCollectionChanged` 接口, 因此绑定到集合上的 WPF 控件知道列表的动态变化。



可从
wrox.com
下载源代码

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.Windows;

namespace Wrox.ProCSharp.Messaging
{
    public partial class CourseOrderReceiverWindow: Window
    {
        private ObservableCollection<CourseOrder> courseOrders =
            new ObservableCollection<CourseOrder>();

        public CourseOrderReceiverWindow()
        {
            InitializeComponent();
            CourseOrderService.CourseOrderAdded += (sender, e) =>
            {
                courseOrders.Add(e.CourseOrder);
                buttonProcessOrder.IsEnabled = true;
            }

            var host = new ServiceHost(typeof(CourseOrderService));
            try
            {
                host.Open();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }

            this.DataContext = courseOrders;
        }
    }
}
```

代码段 `CourseOrderReceiver/CourseOrderReceiverWindow.xaml.cs`

XAML 代码中的 WPF 元素现在使用数据绑定。把 `ListBox` 绑定到数据上下文中, 把单项控件绑定到数据上下文的当前项的属性上。



可从
wrox.com
下载源代码

```
<ListBox Grid.Row="1" x:Name="listOrders" ItemsSource="{Binding}"
        IsSynchronizedWithCurrentItem="true" />

<!-- ... -->

<TextBox x:Name="textCourse" Grid.Row="0" Grid.Column="1"
        Text="{Binding Path=Course.Title}" />
<TextBox x:Name="textCompany" Grid.Row="1" Grid.Column="1"
        Text="{Binding Path=Customer.Company}" />
<TextBox x:Name="textContact" Grid.Row="2" Grid.Column="1"
        Text="{Binding Path=Customer.Contact}" />
```

代码段 CourseOrderReceiver/CourseOrderReceiverWindow.xaml

应用程序配置文件定义 **netMsmqBinding**。为了可靠地传递消息，需要事务队列。要接收和发送非事务队列中的消息，必须把 **exactlyOnce** 属性设置为 **false**。



如果接收应用程序和发送应用程序都是 WCF 程序，就可以使用 **netMsmqBinding** 绑定。如果其中一个程序正在使用 **System.Messaging** API 发送或接收消息，或者其中一个程序是较旧的 COM 应用程序，就可以使用 **msmqIntegrationBinding**。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding name="NonTransactionalQueueBinding" exactlyOnce="false">
          <security mode="None" />
        </binding>
      </netMsmqBinding>
    </bindings>
    <services>
      <service name="Wrox.ProCSharp.Messaging.CourseOrderService">
        <endpoint address="net.msmq://localhost/private/courseorder"
          binding="netMsmqBinding"
          bindingConfiguration="NonTransactionalQueueBinding"
          name="OrderQueueEP"
          contract="Wrox.ProCSharp.Messaging.ICourseOrderService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

buttonProcessOrder 按钮的 Click 事件处理程序从集合类中删除选中的课程订单：



可从
wrox.com
下载源代码

```
private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
    CourseOrder courseOrder = listOrders.SelectedItem as CourseOrder;
    courseOrders.Remove(courseOrder);
    listOrders.SelectedIndex = -1;
    buttonProcessOrder.IsEnabled = false;
}
```

```

        MessageBox.Show("Course order processed", "Course Order Receiver",
            MessageBoxButton.OK, MessageBoxImage.Information);
    }

```

代码段 CourseOrderReceiver/CourseOrderReceiverWindow.xaml.cs

46.9.4 WCF 消息发送应用程序

把发送应用程序修改为使用 WCF 代理类。对于服务协议定, 需要引用 `CourseOrderServiceContract` 程序集, 而使用 WCF 类, 需要引用 `System.ServiceModel` 程序集。

在 `buttonSubmit` 控件的 `Click` 事件处理程序中, `ChannelFactory` 类返回一个代理。通过调用 `AddCourseOrder()` 方法, 该代理给队列发送一条消息:



可从
wrox.com
下载源代码

```

private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        var order = new CourseOrder
        {
            Course = new Course()
            {
                Title = comboCourses.Text
            },
            Customer = new Customer()
            {
                Company = textCompany.Text,
                Contact = textContact.Text
            }
        };

        var factory = new ChannelFactory<ICourseOrderService>("queueEndpoint");
        ICourseOrderService proxy = factory.CreateChannel();
        proxy.AddCourseOrder(order);
        factory.Close();

        MessageBox.Show("Course order submitted", "Course Order",
            MessageBoxButton.OK, MessageBoxImage.Information);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Course Order Error",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}

```

代码段 CourseOrderSender/CourseOrderWindow.xaml.cs

应用程序配置文件定义 WCF 连接的客户端部分。这里也使用 `netMsmqBinding`:



可从
wrox.com
下载源代码

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <bindings>
            <netMsmqBinding>
                <binding name="nonTransactionalQueueBinding"

```

```

        exactlyOnce="false">
        <security mode="None" />
    </binding>
</netMsmqBinding>
</bindings>
<client>
    <endpoint address="net.msmq://localhost/private/courseorder"
        binding="netMsmqBinding"
        bindingConfiguration="nonTransactionalQueueBinding"
        contract="Wrox.ProCSharp.Messaging.ICourseOrderService"
        name="queueEndpoint" />
</client>
</system.serviceModel>
</configuration>

```

代码段 CourseOrderSender/app.config

现在启动应用程序，它的工作方式与以前类似。但不再需要使用 `System.Messaging` 名称空间中的类发送和接收消息。而是使用 TCP 或 HTTP 信道和 WCF 以类似方式编写应用程序。

但是，要创建消息队列和要删除消息，仍需要 `MessageQueue` 类。WCF 只是发送和接收消息的一个抽象。



如果需要使用 `System.Messaging` 应用程序与 WCF 应用程序通信, 就可以使用 `Binding` 替代 `netMsmqBinding`。这个绑定使用用于 COM 和 `System.Messaging` 的消息格式。

46.10 消息队列的安装

可以用 `MessageQueue.Create()` 方法创建消息队列。但运行应用程序的用户通常没有创建消息队列所需的管理特权。

一般, 可以通过安装程序使用 `MessageQueueInstaller` 类创建消息队列。如果安装程序类是应用程序的一部分, 命令行实用程序 `installutil.exe` (或 Windows 安装软件包) 就会调用安装程序的 `Install()` 方法。

Visual Studio 专门支持在 Windows 窗体应用程序中使用 `MessageQueueInstaller` 类。如果把 `MessageQueue` 组件从工具箱拖放到窗体上, 该组件的智能标记就允许添加一个安装程序及其菜单项 `Add Installer`。可以使用属性编辑器配置 `MessageQueueInstaller` 对象, 以定义事务队列、日志队列、格式化程序的类型、基本优先级等。



安装程序详见第 17 章。

46.11 小结

本章介绍了消息队列的用法。消息队列是一种重要的技术, 它不仅提供异步通信, 还提供断开

连接的通信。发送程序和接收程序可以在不同的时间内运行，此时智能客户端就可以使用消息队列，消息队列还可以把负载分布到不同的时间段上，以充分利用服务器的潜能。

消息队列最重要的类是 `Message` 和 `MessageQueue`。`MessageQueue` 类可以发送、接收和查看消息，`Message` 类定义发送的内容。

WCF 提供消息队列的一个抽象。可以使用 WCF 提供的概念，调用代理的方法来发送消息，实现一个服务来接收消息。

下一章介绍 Syndication，以及如何显示 RSS 和 Atom feed 中的数据。

第 47 章

Syndication

本章内容:

- System.ServiceModel.Syndication 名称空间
- Syndication 阅读器
- Syndication Feed

我们有时要提供一些结构化的且常常改变的数据。在许多 Web 站点上, RSS 或 Atom 允许通过 feed 阅读器订阅。RSS(Really Simple Syndication, 真正简单的联合)是一种 XML 格式, 它允许联合信息。RSS 随着博客变得非常流行。这种 XML 信息非常便于使用 RSS 阅读器订阅。

目前, RSS 不仅用于博客, 还用于许多不同的数据源, 如在线新闻杂志。常常改变的任何数据都可以由 RSS 或其后继协议 Atom 提供。Microsoft Internet Explorer 和 Microsoft Outlook 包含 RSS 阅读器和 Atom 阅读器, 它们集成到了产品中。

WCF(Windows Communication Foundation)在 System.ServiceModel.Syndication 名称空间中包含了对联合功能的扩展。这个名称空间中的类可以用于读写 RSS 和 Atom feed。

本章介绍如何创建 Syndication 阅读器, 如何提供数据。

47.1 System.ServiceModel.Syndication 名称空间概述

对于联合, 可以使用 System.ServiceModel.Syndication 名称空间, 它提供的类用于以 RSS 或 Atom 格式联合。

RSS 2.0 版本发布后, RSS 现在已经是 Really Simple Syndication 的缩写。在以前的版本中, 它的名称是 Resource Description Framework(RDF) Site Summary 和 Rich Site Summary。RDF 的第一个版本由 Netscape 创建; 用来描述其门户网站的内容。在 2002 年《纽约时报》开始为其读者提供 RSS 新闻源的订阅, 之后 RSS 变得非常成功。



图 47-1

Atom 联合格式是 RSS 的后继格式, 并通过 RFC 4287(www.ietf.org/rfc/rfc4287.txt)成为一个推荐标准。RSS 和 Atom 之间的主要区别是可以用一项定义的内容。在 RSS 上, 描述元素可以包含简单的文本或 HTML 元素, 在其中阅读应用程序并不关心这些内容。而 Atom 不仅要求用 type 属性为内容定义特定的类型, 它还允许包含带指定名称空间的 XML 内容。

图 47-1 显示了 RSS 和 Atom feed 使用的徽标。如果站点显示这个徽标, 就提供 RSS 或 Atom feed。

表 47-1 列出了允许创建联合源的类和元素。这些类独立于联合类型 RSS 或 Atom。

表 47-1

类	说 明
SyndicationFeed	SyndicationFeed 类表示源的顶级元素。在 Atom 中，顶级元素是<feed>，RSS 把<rss>定义为顶级元素。 使用静态方法 Load()，可以通过 XmlReader 读取源 这个类的 Authors、Categories、Contributors、Copyright、ImageUrl、Links、Title 和 Items 属性可以定义子元素
SyndicationPerson	SyndicationPerson 类表示一个人，其中 Name、Email 和 Uri 可以赋予 Authors 和 Contributors 集合
SyndicationItem	一个源由多项组成。一项的属性有 Authors、Contributors、Copyright 和 Content
SyndicationLink	SyndicationLink 类表示源或项的链接。这个类定义 Title 和 Uri 属性
SyndicationCategory	源可以把项组合到类别中。类别的关键字可以设置为 SyndicationCategory 类的 Name 和 Label 属性
SyndicationContent	SyndicationContent 类是一个抽象基类，它描述某一项的内容。内容的类型可以是 HTML、纯文本、XHTML、XML 或 URL，分别用具体的类 TextSyndicationContent、UrlSyndicationContent 和 XmlSyndicationContent 来描述
SyndicationElement-Extension	对于扩展元素，可以添加额外的内容。SyndicationElementExtension 类可用于给源、类别、人、链接和项添加信息

为了把源转换为 RSS 和 Atom 格式，可以使用派生自抽象基类 SyndicationFeedFormatter 和 SyndicationItemFormatter 的类。在.NET 4 中，用于源的格式化程序有 Atom10FeedFormatter 和 Rss20FeedFormatter，用于项的格式化程序有 Atom10ItemFormatter 和 Rss20ItemFormatter。

47.2 读取联合源的示例

第一个例子是一个 Syndication 阅读器应用程序，其用户界面用 WPF 开发，该 WPF 应用程序的用户界面如图 47-2 所示。

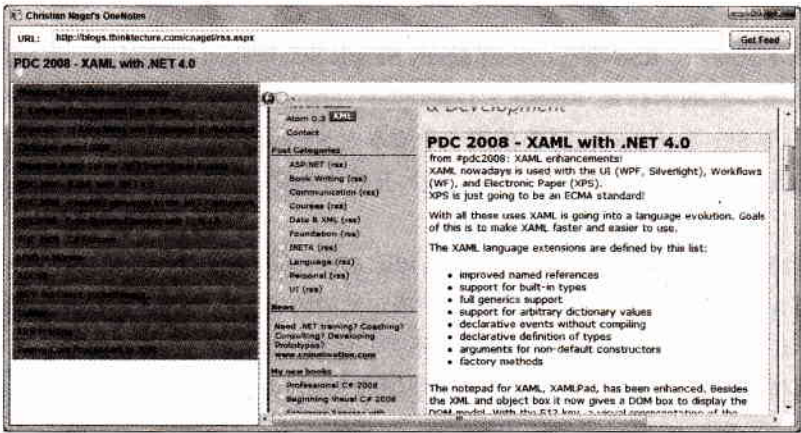


图 47-2

要使用 Syndication API, 必须通过应用程序引用 System.ServiceModel 程序集。把 OnGetFeed() 事件处理程序方法设置为 Get Feed 按钮的 Click 事件。读取应用程序所需的代码非常简单。首先, 把 RSS 源中的 XML 内容读入 System.Xml 名称空间的 XmlReader 类中。Rss20FeedFormatter 格式化程序在 ReadFrom() 方法中接收一个 XmlReader 参数。对于数据绑定, 把返回一个 SyndicationFeed 的 Feed 属性赋予 Window 的 DataContext, 把返回 IEnumerable<SyndicationItem> 的 Feed.Items 属性赋予 DockPanel 容器的 DataContext。



可从
wrox.com
下载源代码

```
private void OnGetFeed(object sender, RoutedEventArgs e)
{
    try
    {
        using (var reader = XmlReader.Create(textUrl.Text))
        {
            var formatter = new Rss20FeedFormatter();
            formatter.ReadFrom(reader);
            this.DataContext = formatter.Feed;
            this.feedContent.DataContext = formatter.Feed.Items;
        }
    }
    catch (WebException ex)
    {
        MessageBox.Show(ex.Message, "Syndication Reader");
    }
}
```

代码段 SyndicationReader/SyndicationReader.xaml.cs

定义用户界面的 XAML 代码如下所示。Window 类的 Title 属性绑定到 SyndicationFeed 的 Title.Text 属性上, 以显示源的标题。

在 XAML 代码中, 定义 DockPanel 容器 heading, 它包含一个绑定到 Title.Text 上的标签和一个绑定到 Description.Text 上的标签。因为这些标签都包含在 DockPanel 容器 feedContent 中, 而 feedContent 绑定到 Feed.Item 属性上, 所以这些标签获得了当前选中项的标题和描述信息。

项列表显示在 ListBox 中, 该 ListBox 使用 ItemTemplate 把标签绑定到 Title 上。

DockPanel 容器 content 包含一个 Frame 元素, 该元素把 Source 属性绑定到项的第一个链接上。之后, Frame 控件使用 Web 浏览器控件显示链接中的内容。



可从
wrox.com
下载源代码

```
<Window x:Class="Wrox.ProCSharp.Syndication.SyndicationReaderWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="{Binding Path=Title.Text}" Height="300" Width="450">

    <DockPanel x:Name="feedContent">
        <Grid DockPanel.Dock="Top">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="50" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="90" />
            </Grid.ColumnDefinitions>
            <Label Grid.Column="0" Margin="5">URL:</Label>
            <TextBox Grid.Column="1" x:Name="textUrl" MinWidth="150"
                Margin="5">http://blogs.thinkitecture.com/cnagel/rss.aspx
```

```

</TextBox>
<Button Grid.Column="2" Margin="5" MinWidth="80"
        Click="OnGetFeed">Get Feed</Button>
</Grid>
<StackPanel Orientation="Vertical" Background="LightGreen"
        DockPanel.Dock="Top" x:Name="heading">
    <Label DockPanel.Dock="Top" Content="{Binding Path=Title.Text}"
        FontSize="16" />
    <Label DockPanel.Dock="Top" Content="{Binding Path=Description.Text}" />
</StackPanel>
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}"
        Style="{StaticResource listTitleStyle}"
        IsSynchronizedWithCurrentItem="True"
        HorizontalContentAlignment="Stretch" />
<DockPanel x:Name="content">
    <Label DockPanel.Dock="Top" Content="{Binding Path=Description.Text}" />
    <Frame Source="{Binding Path=Links[0].Uri}" />
</DockPanel>
</DockPanel>
</Window>

```

代码段 SyndicationReader/SyndicationReader.xaml

47.3 提供联合源的示例

读取联合源的示例时可以使用 Syndication API。Syndication API 还可以给 RSS 和 Atom 客户提供联合源的示例。

为此，Visual Studio 2010 提供了可用作起点的 Syndication Service Library 模板。这个模板定义了对 System.ServiceModel 库的引用，并添加了一个应用程序配置文件，来定义 WCF 端点。

为了给联合源的示例提供数据，可以使用 ADO.NET Entity Framework。在示例应用程序中，使用 Formula-1 数据库，它可以和本书的示例应用程序一起从 Wrox 网站(www.wrox.com)和本书附赠光盘上找到。把名为 Formula1Model.edmx 的 ADO.NET Entity Data Model 项添加到项目中。其中，Racers、RaceResults、Races 和 Circuits 表映射到 Racer、RaceResult、Race 和 Circuit 实体类上，如图 47-3 所示。



图 47-3



ADO.NET Entity Framework 详见第 31 章。

项目模板创建了一个 `IService1.cs` 文件，该文件包含 WCT 服务的协定。该接口包含 `CreateFeed()` 方法，这个方法返回一个 `SyndicationFeedFormatter`。因为 `SyndicationFeedFormatter` 是一个抽象类，所返回的实际类型是 `Atom10FeedFormatter` 或 `Rss20FeedFormatter`，而这些类型用 `ServiceKnownTypeAttribute` 列出，所以序列化时类型是已知的。

`WebGet` 属性定义了可以从一个简单的 HTTP GET 请求中调用的操作，该操作可用于请求联合源。`WebMessageBodyStyle.Bare` 指定，发送结果(联合源中的 XML)时，就好像这些 XML 没有添加 XML 包装元素一样。



可从
wrox.com
下载源代码

```
using System.ServiceModel;
using System.ServiceModel.Syndication;
using System.ServiceModel.Web;

namespace Wrox.ProCSharp.Syndication
{
    [ServiceContract]
    [ServiceKnownType(typeof(Atom10FeedFormatter))]
    [ServiceKnownType(typeof(Rss20FeedFormatter))]
    public interface IFormula1Feed
    {
        [OperationContract]
        [WebGet(UriTemplate = " * ", BodyStyle = WebMessageBodyStyle.Bare)]
        SyndicationFeedFormatter CreateFeed();
    }
}
```

代码段 `SyndicationService/IFormula1Feed.cs`

该服务的实现在类 `Formula1Feed` 中完成。其中，创建了一个 `SyndicationFeed` 项，这个类还指定了各个属性，如 `Generator`、`Language`、`Title`、`Categories` 和 `Authors`。从一个 LINQ 查询填充 `Items` 属性，该查询请求指定日期的一级方程式大赛的获奖者。这个查询的 `select` 子句创建了一个新的匿名类型，该匿名类型用几个属性来填充，接着 `Select()` 方法使用该匿名类型为每个获奖者创建了一个 `SyndicationItem`。在 `SyndicationItem` 中，把 `Title` 属性赋予包含举办国的纯文本。`Content` 属性使用 LINQ to XML 来填充。`XElement` 类用于创建能由浏览器解释的 XHTML 代码。这项内容显示了比赛的日期、举办国和获奖者的姓名。

根据请求联合的查询字符串，用 `Atom10FeedFormatter` 或 `Rss20FeedFormatter` 格式化程序格式化 `SyndicationFeed`。



可从
wrox.com
下载源代码

```
using System;
using System.Linq;
using System.ServiceModel.Syndication;
using System.ServiceModel.Web;
using System.Xml.Linq;

namespace Wrox.ProCSharp.Syndication
{
    public class Formula1Feed: IFormula1Feed
```

```

public SyndicationFeedFormatter CreateFeed()
{
    DateTime fromDate = DateTime.Today - TimeSpan.FromDays(365);
    DateTime toDate = DateTime.Today;
    string from = WebOperationContext.Current.IncomingRequest.UriTemplateMatch
        .QueryParameters["from"];
    string to = WebOperationContext.Current.IncomingRequest.UriTemplateMatch
        .QueryParameters["to"];

    if (from != null && to != null)
    {
        try
        {
            fromDate = DateTime.Parse(from);
            toDate = DateTime.Parse(to);
        }
        catch (FormatException)
        {
            // keep the default dates
        }
    }

    // Create a new Syndication Feed.
    var feed = new SyndicationFeed();
    feed.Generator = "Pro C# 4.0 Sample Feed Generator";
    feed.Language = "en-us";
    feed.LastUpdatedTime = new DateTimeOffset(DateTime.Now);
    feed.Title = SyndicationContent.CreatePlaintextContent(
        "Formula1 results");
    feed.Categories.Add(new SyndicationCategory("Formula1"));
    feed.Authors.Add(new SyndicationPerson("web@christiannagel.com",
        "Christian Nagel", "http://www.christiannagel.com"));
    feed.Description = SyndicationContent.CreatePlaintextContent(
        "Sample Formula 1");
    using (var data = new Formula1Entities())
    {
        var races = (from racer in data.Racers
            from raceResult in racer.RaceResults
            where raceResult.Race.Date > fromDate &&
                raceResult.Race.Date < toDate &&
                raceResult.Position == 1
            orderby raceResult.Race.Date
            select new
            {
                Country = raceResult.Race.Circuit.Country,
                Date = raceResult.Race.Date,
                Winner = racer.Firstname + " " + racer.Lastname
            }).ToArray();
        feed.Items = races.Select(race =>
        {
            return new SyndicationItem
            {
                Title = SyndicationContent.CreatePlaintextContent(
                    String.Format("G.P. {0}", race.Country)),
            }
        });
    }
}

```


通过服务的默认请求, 返回 RSS 源。之后使用 rss 根元素提取 RSS 源。在 RSS 中, 把 Title 属性转换为 title 元素, 把 Description 属性变成 description 元素。SyndicationFeed 的 Authors 属性包含 SyndicationPerson, SyndicationPerson 使用电子邮件地址创建 managingEditor 元素。通过 RSS, 对 item 的 description 元素内容进行编码:

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0" xmlns:a10="http://www.w3.org/2005/Atom">
  <channel>
    <title>Formulal results</title>
    <description>Sample Formula 1</description>
    <language>en-us</language>
    <managingEditor>web@christiannagel.com</managingEditor>
    <lastBuildDate>Sat, 25 Jul 2009 11:25:31+0200</lastBuildDate>
    <category>Formulal</category>
    <generator>Pro C# 4.0 Sample Feed Generator</generator>
    <item>
      <title>G.P. South Africa</title>
      <description>&lt;p&gt;&#xD; &lt;h3&gt;South Africa, 07.03.1970&lt;/h3&gt;
        &#xD; &lt;b&gt;Winner: Jack Brabham&lt;/b&gt;&#xD; &lt;/p&gt;
      </description>
    </item>
    <item>
      <!-- -->
    </channel>
  </rss>
```

用?from=1970/1/1&to=1971/1/1&format=atom 查询返回 Atom 格式化的来源, 结果如下所示。根元素现在是 feed 元素, Description 属性变成 subtitle 元素, Authors 属性的值现在完全不同于前面的 RSS 源。Atom 允许不解码内容, 也很容易找出 XHTML 元素。

```
<feed xml:lang="en-us" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Formulal results</title>
  <subtitle type="text">Sample Formula 1</subtitle>
  <id>uuid:c19284e7-aa40-4bc2-9be8-f1960b0f747e;id=1</id>
  <updated>2009-07-25T11:44:01+02:00</updated>
  <category term="Formulal"/>
  <author>
    <name>Christian Nagel</name>
    <uri>http://www.christiannagel.com</uri>
    <email>web@christiannagel.com</email>
  </author>
  <generator>Pro C# 4.0 Sample Feed Generator</generator>
  <entry>
    <id>uuid:c19284e7-aa40-4bc2-9be8-f1960b0f747e;id=2</id>
    <title type="text">G.P. South Africa</title>
    <updated>2009-07-25T09:44:05Z</updated>
    <content type="xhtml">
      <p><h3>South Africa, 07.03.1970</h3><b>Winner: Jack Brabham</b></p>
    </content>
  </entry>
  <!-- -->
</feed>
```

47.4 小结

本章介绍了 `System.ServiceModel.Syndication` 名称空间中的类如何用于创建接收源的应用程序和提供源的应用程序。`Syndication` API 支持 RSS 2.0 和 Atom 1.0。在这些联合标准出现时，可以使用新的格式化程序。`SyndicationXXX` 类独立于所生成的格式。抽象类 `SyndicationFeedFormatter` 的具体实现方式定义了使用什么属性，以及它们如何转换为特定的格式。

本章完成了本书的通信部分。我们学习了直接使用套接字和所提供的抽象层的通信技术。`Windows Communication Foundation(WCF)` 技术在前面几章中讨论过。`WCF` 利用第 46 章介绍的消息队列技术，提供了断开连接的通信模型。

还有许多相关的内容。附录 A 介绍了 `Windows API` 代码包，这是 `Windows 7` 和 `Windows Server 2008 R2` 的一个扩展，以及在这些平台上开发应用程序的指南。

第VI部分

附 录

- 附录 A Windows 7 和 Windows Server 2008 R2 指南

附录 A

Windows 7 和 Windows Server 2008 R2 指南

本附录内容:

- Windows API 代码包
- 重启管理器
- 用户账户控制
- Windows 7 目录结构
- 命令链接和任务对话框
- 任务栏和跳转列表

本附录介绍为 Windows 7 和 Windows Server 2008 R2 开发应用程序所需要了解的内容, 以及如何在 .NET 应用程序中使用新的 Windows 功能。



本附录不介绍对 Windows 7 用户或 Windows Server 2008 管理员有用的特性, 而只讨论对开发人员很重要的特性。

如果应用程序不只面向 Windows Vista 或更高版本, 就应注意, WPF、WCF、WF 和 LINQ 也可以用于 Windows XP, 但本章不讨论这些主题。另外, 一些特性只能用于 Windows 7。如果仍面向 Windows XP, 那么还要考虑在 Windows 7 上运行应用程序的问题, 应特别关注用户账户控制和目录的变化。

A.1 概述

Windows API 有许多新的调用, 这些调用提供了可用于 Windows 7 和 Windows Server 2008 R2 的新功能; 但 .NET 4 发布时, 其中的许多调用不能用于 .NET Framework。但是, Windows API 代码包所包含的 .NET 类包装了本地 API 调用, 使它们可用于 .NET 库。可以在 .NET 应用程序中使用这个库。Windows API 代码包可以从 <http://code.msdn.microsoft.com/WindowsAPICodePack> 上下载。这个工具包可用于本附录的大多数示例。可以使用这个工具包附带的库, 也可以把源代码复制到应用程序中。这个工具包中的类一般是 Windows API 的一个瘦包装器。

可以用于面向 Windows 平台的应用程序的另一个工具包是 Windows Software Logo Kit。在 Microsoft 下载页面上, 可以找到 Windows 7 Client Software Logo 程序文档和工具包, 其中包含了应

用程序获取 Windows 7 软件徽标所需的信息。即使对应用程序接收徽标不感兴趣,这个文档和工具包仍很有趣,因为它解释了应用程序在 Windows 操作系统上正常执行所需的条件。

应用程序添加徽标的要求是什么?下面是这些要求的简短列表,Windows 7 Client Software Logo 文档提供了更多细节。

- 应用程序不能包含间谍软件或恶意软件。当然,每个应用程序都应满足这个要求。有趣的是,这个要求用于验证徽标的一致性。
- Windows 资源受保护的文件不能被替代。
- 必须使用 Windows 错误报告(Windows Error Reporting, WER)报告错误。
- 应用程序的安装必须完整,卸载必须干净。参见第 17 章中关于创建安装程序的信息。应用程序在安装时不应强制重新启动。
- 应用程序应安装到正确的文件夹中。用户数据不应写入应用程序文件夹。A.4 节将讨论如何寻找用于应用程序和用户数据的正确文件夹。
- 文件和驱动器必须进行数字签名。参见第 21 章中如何给应用程序签名的信息。
- 必须支持 64 位系统。这意味着不允许使用 16 位代码,可以在 64 位系统上运行 32 位应用程序。对于 .NET 应用程序,可以把平台配置设置为 Any CPU、x86、x64 或 Itanium。Any CPU 允许应用程序在 32 位系统上运行 32 位代码,在 64 位系统上运行 64 位代码。把配置设置为 Any CPU,就不能直接在程序集中利用平台调用包装本地代码。使用本地代码的限制是,本地 API 调用只能使用本地代码,因此 Any CPU 不能在两个版本上工作。
- 必须检查安装的版本,检查操作系统是否有所需的最低版本,但不允许检查特定的版本。
- 必须遵守用户账户控制规则,本附录会详细讨论相关内容。
- 应用程序不能阻止关机。在重新启动或失败后,可以使用 Restart Manager 把应用程序返回到工作状态。参见 A.2 节。
- 必须支持多用户会话。一个系统上的不同用户应能使用应用程序。这是因为 Windows 具备快速用户切换功能。

A.2 应用程序恢复

也许读者见过,一些应用程序在崩溃或系统重新启动时可以恢复到前一个状态。这类应用程序(如 Microsoft Office 和 Visual Studio)使用 Application Recovery and Restart (应用程序恢复和重新启动,ARR) API,这个 API 自从 Windows Vista 以来就是 Windows API 的一部分。

应用程序可以使用 Restart Manager 注册,如果应用程序崩溃或者不响应(至少运行 60 秒后),Windows 错误报告(WER)就重新启动该应用程序。应用程序也可以注册为恢复,这样如果该应用程序挂起,就可以恢复它。

为了说明这一点,下面的例子将创建一个简单的 WPF 编辑器。这个应用程序注册为如果失败就重新启动并恢复它。为了读写文件,CurrentFile 类定义了 IsDirty 和 Content 属性,以及 Load()和 Save()方法。这个应用程序使用跟踪功能,把详细消息写入一个跟踪侦听器中,以便重新跟踪应用程序中的事件。



跟踪功能参见第 19 章。



可从
wrox.com
下载源代码

```
using System;
using System.Diagnostics;
using System.IO;

namespace Wrox.ProCSharp.Windows7
{
    public class CurrentFile
    {
        public bool IsDirty { get; set; }
        public string Content { get; set; }

        public void Load (string fileName)
        {
            App.Source.TraceEvent (TraceEventType.Verbose, 0,
                                   String.Format("begin Load {0}", fileName));
            Content = File.ReadAllText(fileName);
            IsDirty = false;
            App.Source.TraceEvent (TraceEventType.Verbose, 0,
                                   String.Format("end Load {0}", fileName));
        }

        public void Save (string fileName)
        {
            App.Source.TraceEvent (TraceEventType.Verbose, 0,
                                   String.Format("begin Save {0}", fileName));
            File.WriteAllText(fileName, Content);
            IsDirty = false;
            App.Source.TraceEvent (TraceEventType.Verbose, 0,
                                   String.Format("end Save {0}", fileName));
        }
    }
}
```

代码段 EditorDemo/CurrentFile.cs

如果应用程序因为崩溃而重新启动，就把/restart:[filename]命令行参数传递给可执行程序。为了检索命令行信息，把 OnStartup() 处理程序方法赋予 App 类的 Startup 事件。利用第二个 StartupEventArgs 类型的参数，就可以检索命令行参数。使用/restart:参数会把其后的文件名赋予 RestartPath 属性。App 类还包含 TraceSource 对象，它用于跟踪。



可从
wrox.com
下载源代码

```
using System;
using System.Diagnostics;
using System.Windows;

namespace Wrox.ProCSharp.Windows7
{
    public partial class App : Application
    {
        public static TraceSource Source = new TraceSource("RestartDemo");

        private void OnStartup(object sender, StartupEventArgs e)
        {

```

```

        Source.TraceEvent(TraceEventType.Verbose, 0,
            "RestartDemo begin OnStartup");
        foreach (var arg in e.Args)
        {
            Source.TraceEvent(TraceEventType.Verbose, 0,
                String.Format("argument {0}", arg));
            if (arg.StartsWith("/restart:",
                StringComparison.InvariantCultureIgnoreCase))
            {
                Source.TraceEvent(TraceEventType.Verbose, 0,
                    "/restart: argument found");
                RestartPath = arg.Substring(9);
                Source.TraceEvent(TraceEventType.Verbose, 0,
                    String.Format("RestartPath: {0}",
                        RestartPath));
            }
        }
    }
    public string RestartPath { get; private set; }
}
}

```

代码段 EditorDemo/App.xaml.cs

主窗口(如图 A-1 所示)仅在中间区域包含一个用于编辑文本的文本框,在底部区域包含一个显示状态消息的 TextBlock 控件,还包含一个 Menu 控件。

MainWindow 类的构造函数检查 App 类的 RestartPath 属性,以确定应用程序是否从恢复中启动。如果应用程序从恢复中启动,就使用赋予 RestartPath 属性的文件名加载这个文件,并把它赋予 TextBox 的 DataContext。如果应用程序正常启动,就创建一个用于恢复的新文件名。

在构造函数内部,还创建了一个 DispatcherTimer,它在 60 秒后触发。这用于通知用户,应用程序在运行至少 60 秒后,只能从一次崩溃中恢复。

在这种情况下,最重要的调用是 RegisterForRestart()和 RegisterForRecovery()辅助方法,调用它们便于通过 Restart Manager 注册。

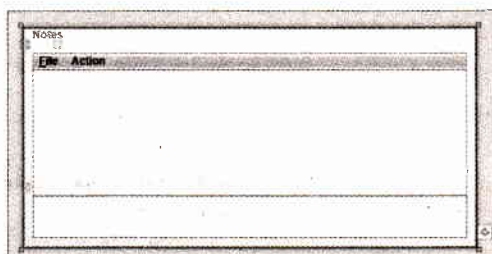


图 A-1



可从
wrox.com
下载源代码

```

using System;
using System.Diagnostics;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Threading;
using Microsoft.WindowsAPICodePack.ApplicationServices;

namespace Wrox.ProCSharp.Windows7
{
    public partial class MainWindow : Window
    {

```

```

private DispatcherTimer minuteTimer;
private CurrentFile currentFile = new CurrentFile();
private string tempPath;

public MainWindow()
{
    InitializeComponent();
    if ((Application.Current as App).RestartPath != null)
    {
        tempPath = (Application.Current as App).RestartPath;
        currentFile.Load(tempPath);
        App.Source.TraceEvent(TraceEventType.Verbose, 0,
            String.Format("Application recovered data from the file {0}",
                tempPath));
    }
    else
    {
        string tempFileName = String.Format("{0}.txt", Guid.NewGuid());
        tempPath = System.IO.Path.Combine(Environment.GetFolderPath(
            Environment.SpecialFolder.LocalApplicationData),
            tempFileName);
        App.Source.TraceEvent(TraceEventType.Verbose, 0,
            String.Format("Normal start with temp - filename {0}",
                tempPath));
    }
    this.text1.DataContext = currentFile;

    minuteTimer = new DispatcherTimer(TimeSpan.FromSeconds(60),
        DispatcherPriority.Normal, On60Seconds,
        Dispatcher.CurrentDispatcher);
    minuteTimer.Start();

    RegisterForRestart();
    RegisterForRecovery();
}

```

代码段 EditorDemo/MainWindow.xaml.cs



如果应用程序运行了至少 60 秒，就只能重新启动。这是有意义的，因为如果应用程序在第一个 60 秒内崩溃，那么最好不要重新启动；用户可能在这个时间内不会丢失很多数据。测试这个功能时，至少需要在应用程序崩溃之前等待 1 分钟。

为了标记已改变的内容，使用 TextBox 控件的 TextChanged 事件的处理程序设置 currentFile 的 IsDirty 属性：

```

private void OnTextChanged(object sender, TextChangedEventArgs e)
{
    if (!currentFile.IsDirty)
    {
        currentFile.IsDirty = true;
        textStatus.Text += String.Format("{0:T}: Set IsDirty\n",
            DateTime.Now);
    }
}

```

DispatcherTimer 类的处理程序方法 On60Seconds()把状态信息写入 textStatus 文本块中,说明 60 秒的期限已过,为了恢复现在可以使用应用程序崩溃。

```
private void On60Seconds(object sender, EventArgs e)
{
    this.textStatus.Text += String.Format("{0:T}: Application can crash " +
        "now\n", DateTime.Now);
    minuteTimer.Stop();
}
```

RegisterForRestart()方法调用 ApplicationRestartRecoveryManager 类的 RegisterFor ApplicationRestart()方法。ApplicationRestartRecoveryManager 类在 Microsoft.WindowsAPICodePack.ApplicationServices 名称空间中定义。RegisterForApplicationRestart()方法包装 Windows API 函数 RegisterApplicationRestart()的调用。利用这个方法,可以定义用于重新启动应用程序的命令行参数和当应用程序不应重新启动的情况。在这个例子中,指定应用程序不应在系统重新启动或安装补丁时重新启动,也可以指定应用程序在崩溃和挂起时不应重新启动。

```
private void RegisterForRestart()
{
    var settings = new RestartSettings(
        String.Format("/restart:{0}", tempPath),
        RestartRestrictions.NotOnReboot | RestartRestrictions.NotOnPatch);
    ApplicationRestartRecoveryManager.RegisterForApplicationRestart(
        settings);
    textStatus.Text += String.Format("{0:T}: Registered for restart\n",
        DateTime.Now);
    App.Source.TraceEvent(TraceEventType.Verbose, 0,
        "Registered for restart");
}
```

在 Windows API 代码包中, Windows API 调用定义为使用 P/Invoke 来调用:

```
[DllImport("kernel32.dll")]
[PreserveSig]
internal static extern HRESULT RegisterApplicationRestart(
    [MarshalAs(UnmanagedType.BStr)] string commandLineArgs,
    RestartRestrictions flags);
```



P/Invoke 参见第 26 章

有了这个功能,如果应用程序崩溃或挂起,就会重新启动。利用 Restart Manager,还可以定义一个处理程序方法,在应用程序因为崩溃或挂起而停止执行前调用它。为此,需要调用 ApplicationRestartRecoveryManager 类的 RegisterForApplicationRecovery()方法。这个方法包装 Windows API 函数 RegisterApplicationRecoveryCallback()。RegisterForApplicationRecovery()方法可以用 RecoverySettings 类型的对象来配置。使用 RecoverySettings 指定一个 RecoveryData 类型的对象和一个恢复 ping 时间间隔。这个 ping 时间间隔的单位是 100ns。回调方法必须在这个时间段内给恢复管理器提供进度信息。默认情况下,值为 0 的时间间隔是 5 秒。可以设置的最大值是 5 分钟。使用

RecoveryData 指定由 Restart Manager 进行恢复所调用的方法，以及传递给该方法的一个参数：

```
private void RegisterForRecovery()
{
    var settings = new RecoverySettings(new RecoveryData(
                                                DoRecovery, tempPath), 0);
    ApplicationRestartRecoveryManager.RegisterForApplicationRecovery(
        settings);
    textStatus.Text += String.Format("{0:T}: Registered for recovery\n",
                                      DateTime.Now);
    App.Source.TraceEvent(TraceEventType.Verbose, 0,
        "Registered for recovery");
}
```

DoRecovery()方法是由 RegisterForApplicationRecovery()方法调用定义的回调方法，它由 WER 调用。在处理程序中，通过调用 ApplicationRecoveryInProgress()方法，来通知 Restart Manager 恢复的进度。如果用户取消了恢复，这个方法就返回 true。在恢复过程中，可以验证用户的取消操作，以便在用户取消恢复时停止恢复。这里把 TextBox 的内容写入重新启动应用程序时使用的临时文件。在恢复过程的最后，通过调用 ApplicationRecoveryFinished()方法，通知 Restart Manager 恢复成功。

```
private int DoRecovery(object state)
{
    App.Source.TraceEvent(TraceEventType.Verbose, 0, "begin Recovery");
    this.tempPath = (string)state;
    bool canceled = ApplicationRestartRecoveryManager.
        ApplicationRecoveryInProgress();
    if (canceled)
    {
        textStatus.Text += String.Format(
            "{0:T}: Recovery canceled, shutting down\n", DateTime.Now);
        App.Source.TraceEvent(TraceEventType.Verbose, 0,
            "end Recovery with cancel");
        ApplicationRestartRecoveryManager.ApplicationRecoveryFinished(
            false);
        return 0;
    }
    SaveFile(tempPath);
    App.Source.TraceEvent(TraceEventType.Verbose, 0, "end Recovery");
    ApplicationRestartRecoveryManager.ApplicationRecoveryFinished(true);
    return 0;
}
```

这些工作完成后，就可以运行应用程序，如图 A-2 所示。底部的 TextBox 给出了状态信息，例如，重新启动和恢复的注册时间，以及 60 秒后，应用程序可以崩溃的信息。

选择 Action | Crash 命令，会调用处理程序方法 OnCrash()，该方法使应用程序快速失败。Environment.FailFast()方法用于以快速方式终止进程。这个方法会写入应用程序事件日志，并在终止进程前创建应用程序的一个转储。仅执行 CriticalFinalizerObject 对象。不执行 Try-finally 块和终结器不执行。

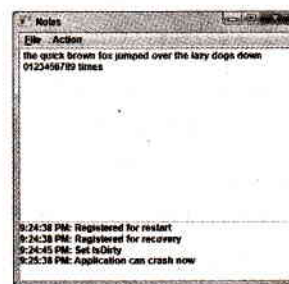


图 A-2

```
private void OnCrash(object sender, ExecutedRoutedEventArgs e)
{
    Environment.FailFast("RestartDemo stopped from Menu command");
}
```

应用程序崩溃后，会重新启动，显示的对话框如图 A-3 所示。

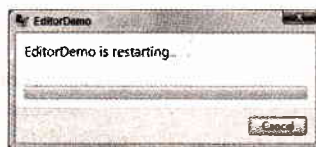


图 A-3

A.3 用户账户控制

作为开发人员，应知道用户账户控制(UAC)是 Windows Vista 中的一个功能。每次执行管理任务时，都会弹出一个请求管理权限的对话框。在 Windows 7 中，可以配置这个功能，如图 A-4 所示。默认设置是如果应用程序是操作系统的一部分，它需要改变管理权限，就不显示通知。

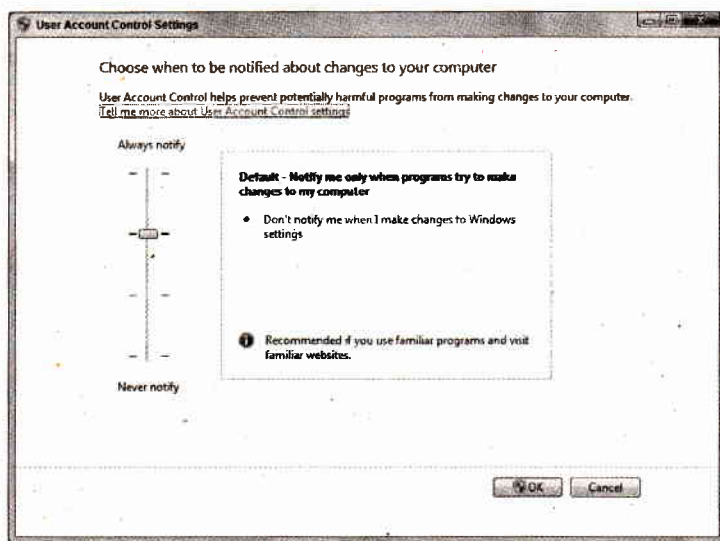


图 A-4

这个繁琐的对话框的隐含目的是什么呢？尽管 Windows 规则总是提及，如果不是确实必要应用程序就不需管理权限，但许多应用程序仍需要运行在管理员账户下。例如，一般用户不允许直接将数据写入 program files 目录，这需要管理权限。许多应用程序都不能在没有管理权限的情况下运行(但开发人员并不需要管理权限)，所以许多用户都使用管理员账户登录系统。当然，这会带来安装木马程序的高风险。

Windows Vista 及其以后版本避免了这个问题，管理员在默认情况下没有管理权限。这个过程有两个相关联的安全标记，一个用于一般用户权限；另一个用于管理权限(此时登录到管理员账户上)。对于需要管理权限的应用程序，用户可以以管理员的身份运行它，为此，可以使用上下文菜单的 Run as Administrator 选项，或者在应用程序的 Compatibility 属性中，将应用程序配置为总是需要管理权限运行它。这个设置会在注册表的 HKCU\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers 上添加应用程序兼容性标志，其值为 RUNASADMIN。

A.3.1 需要管理权限的应用程序

对于需要管理权限的应用程序，还可以添加一个应用程序清单。Visual Studio 2010 有一个项模板，用于把应用程序清单添加到应用程序中。为此，可以给已有应用程序添加一个清单文件，或者在程序集中嵌入一个 Win32 资源文件。在 Project 属性的 Application 选项卡中，可以选择作为本地资源包含的应用程序的清单文件。

应用程序清单是一个类似于应用程序配置文件的 XML 文件。应用程序配置文件的扩展名是.config，而清单文件以.manifest 结尾。该文件的名称必须设置为应用程序的名称，包括.exe 文件扩展名，其后是.manifest。Visual Studio 重命名和复制 app.manifest 文件的方式与应用程序配置文件的方式相同。清单文件包含如下所示的 XML 数据。根元素是<assembly>，它包含<trustInfo>子元素。管理员要求用<requestedExecutionLevel>元素的 level 属性来定义。



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
  xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</asmv1:assembly>
```

代码段 AdminRightsRequired/app.manifest

以这种方式启动应用程序，就会显示一个提升权限级别的提示，询问用户是否信任在管理权限下运行应用程序。

requestedExecutionLevel 设置可以指定为 requireAdministrator、highestAvailable 和 asInvoker。highestAvailable 表示，只有得到用户的允许，应用程序才能获得用户拥有的权限。requireAdministrator 需要管理员权限。如果用户没有以管理员的身份登录系统，就出现一个登录对话框，用户可以以应用程序管理员的身份登录。asInvoker 表示应用程序运行在用户的安全标记下。

uiAccess 属性指定，应用程序是否需要输入到桌面上一个权限级别较高的窗口上。例如，由于屏幕上的键盘需要将输入放在桌面上的其他窗口中，因此对于显示屏幕软键盘的应用程序，这个设置应设置为 true。不支持 UI 访问的应用程序应将这个属性设置为 false。



给应用程序授予管理权限的另一种方式是编写 Windows 服务。因为 UAC 仅用于交互式进程，所以 Windows 服务可以获得管理权限。还可以编写一个无权限的 Windows 应用程序，使用 WCF 或另一种通信技术与有权限的 Windows 服务通信。

Windows 服务参见第 25 章，WCF 参见第 43 章。

A.3.2 防火墙图标

如果应用程序或其中的任务需要管理权限，就应通过一个容易识别的防火墙图标通知用户。把防火墙图标附加到需要提升权限级别的控件上。用户在单击带防火墙的项时，会看到一个提升权限级别的提示。图 A-5 和图 A-6 显示了正在使用的防火墙图标。任务管理器需要提升权限级别，才能查看所有用户的进程。对于用户账户，需要提升权限级别，才能添加或删除用户账户，以及创建父控件。

使用本附录后面介绍的新命令链接控件，就可以在应用程序中创建防火墙图标。

当用户单击带防火墙图标的控件时，会显示一个提升权限级别的提示。提升权限级别的提示根据提升的应用程序类型的不同而不同。系统可以区分出 Windows 发布的应用程序(使用 Windows 7 的默认设置，根本不会显示这个提示)、包含证书的应用程序和不含证书的应用程序。当然，对于不含证书的应用程序，该提示会以彩色突出显示，并用粗体标记未知发布者。

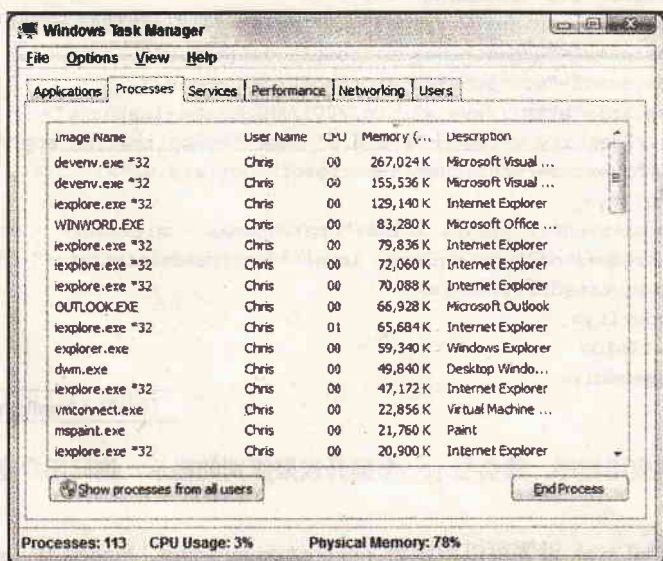


图 A-5

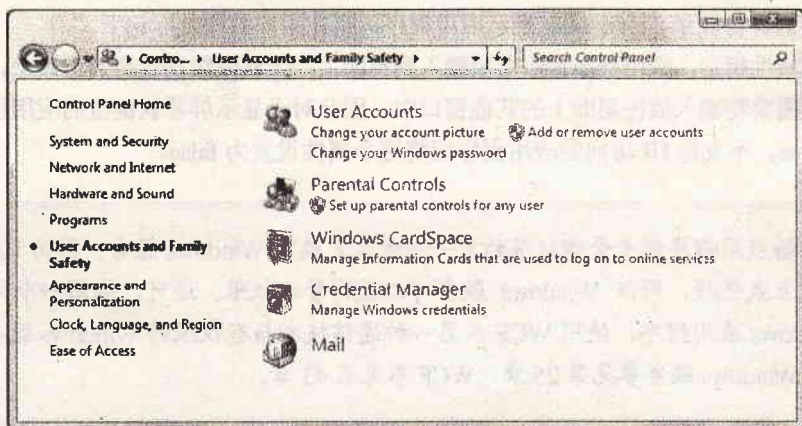


图 A-6

A.4 目录结构

Windows 的目录结构在 Windows Vista 中有变化, 在 Windows 7 中继续使用这个新结构。不再有 C:\Documents and Settings\<username> 目录, 它被新文件夹 C:\Users\<username> 替代。Windows XP 定义 My Documents 子目录, 来存储用户专用的数据。Windows 7 定义 C:\Users\<username>\Documents 来实现相同目的。

如果遵循不在程序中使用硬编码的路径值这条简单规则, 文件夹在什么地方就不重要。文件夹在不同的 Windows 语言中有所不同。对于特殊文件夹, 应使用 Environment 类和 SpecialFolder 枚举:

```
string folder = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
```

SpecialFolder 枚举定义的一些文件夹如表 A-1 所示。

表 A-1

内 容	SpecialFolder 枚举	Windows 7 的默认目录
用户的专用文档	Personal	c:\Users\<User>\Documents
漫游用户的用户专用数据	ApplicationData	c:\Users\<User>\AppData\Roaming
本地系统用户的用户专用数据	LocalApplicationData	c:\Users\<User>\AppData\Local
程序文件	ProgramFiles	c:\Program Files
在不同程序之间共享的程序文件	CommonProgramFiles	c:\Program Files\Common Files
所有用户共有的应用程序数据	CommonApplicationData	c:\ProgramData



注销时, 因为漫游目录的内容会复制到服务器上, 所以如果用户登录到另一个系统上, 就会复制相同的内容, 并且相同的内容可以用于该用户访问的所有系统。

使用特殊文件夹时必须小心, 因为一般用户不能对 Program Files 目录进行直接的写入访问。可以把应用程序中的用户专用数据写入 LocalApplicationData 中, 或者对于漫游用户则写入 ApplicationData 中。应在不同用户之间共享的数据可以写入 CommonApplicationData 中。

A.4 新控件和对话框

Windows Vista 和 Windows 7 发布了几个新控件。命令链接控件是对 Button 控件的扩展, 与其他几个控件一起使用。任务对话框是下一代的 MessageBox, 还有用于打开和保存文件的新对话框。

A.4.1 命令链接

命令链接控件是对 Windows 按钮控件的扩展。命令链接包含一个可选的图标和一个注释文本。这个控件常常用于任务对话框和向导, 它提供的信息比 OK 按钮和 Cancel 按钮控件更多。

在 .NET 应用程序中, 可以使用 Windows API 代码包创建命令链接控件。如果将 Microsoft.WindowsAPICodePack.Shell 项目添加到解决方案中, 就可以将工具箱中的 Windows 窗体控件 CommandLink 添加到 Windows 窗体应用程序中。CommandLink 类派生自 System.Windows.Forms.Button 类。命令链接是对本地 Windows Button 控件的一种扩展, 它定义了其他 Windows 消息和一个待配置按钮的新样式。包装类 CommandLink 发送 Windows 消息 BCM_SETNOTE 和 BCM_SETSHIELD, 并设置 BS_COMMANDLINK 样式。除了 Button 类的成员之外, 该类提供的公共方法和属性还有 NoteText 和 ShieldIcon。

下面的代码创建一个新的命令链接控件, 该控件设置 NoteText 和 ShieldIcon。图 A-7 显示了在运行期间配置的命令链接。



可从
wrox.com
下载源代码

```
this.commandLink1 =
new Microsoft.WindowsAPICodePack.Controls.
    WindowsForms.CommandLink();
this.commandLink1.FlatStyle =
System.Windows.Forms.FlatStyle.System;
this.commandLink1.Location = new System.Drawing.Point(44, 54);
this.commandLink1.Name = "commandLink1";
this.commandLink1.NoteText = "Clicking this command requires " +
    "admin rights";
this.commandLink1.ShieldIcon = true;
this.commandLink1.Size = new System.Drawing.Size(193, 121);
this.commandLink1.TabIndex = 0;
this.commandLink1.Text = "Give access to this computer";
this.commandLink1.UseVisualStyleBackColor = true;
this.commandLink1.Click += new System.EventHandler(
this.OnClickCommandLink);
```

代码段 WindowsFormsCommandLink/Form1.Designer.cs

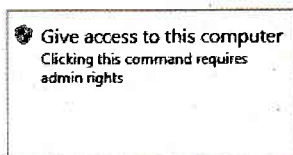


图 A-7

对于 WPF, 命令链接重新定义为 WPF 用户控件 CommandLink。这个控件位于 Microsoft.WindowsAPICodePack.Controls.WindowsPresentationFoundation 名称空间中。其 Icon 属性可以指定一个 IconSource、Link 属性(主文本的值)和 Note 属性(附加文本的值)。图 A-8 显示了这个命令链接的结果。



可从
wrox.com
下载源代码

```
<shell:CommandLink Icon="Images/Globe.ico" x:Name="commandLink1"
    Link="Browse the Web"
    Note="Find data on the World Wide Web"
    Click="OnBrowse" Margin="5" Height="57"
    VerticalAlignment="Top"
    HorizontalAlignment="Center" />
```

代码段 WPFCommandLink/MainWindow.xaml

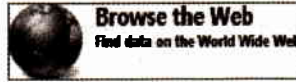


图 A-8

A.4.2 任务对话框

任务对话框是替代旧消息框的新一代对话框，它是新命令控件的一部分。Windows API 定义了 `TaskDialog()` 和 `TaskDialogIndirect()` 函数来创建任务对话框。`TaskDialog()` 函数可以创建简单的对话框，`TaskDialogIndirect()` 函数用于创建比较复杂的对话框，其中包含命令链接控件和扩展内容。

使用 Windows API 代码包，通过 `P/Invoke` 包装 `TaskDialogIndirect()` 函数的本地 API 调用：

```
[DllImport(CommonDllNames.ComCtl32, CharSet = CharSet.Auto,
    SetLastError = true)]
internal static extern HRESULT TaskDialogIndirect(
    [In] TaskDialogNativeMethods.TASKDIALOGCONFIG pTaskConfig,
    [Out] out int pnButton,
    [Out] out int pnRadioButton,
    [MarshalAs(UnmanagedType.Bool)][Out]
        out bool pVerificationFlagChecked);
```

把 `TaskDialogIndirect()` 函数的第一个参数定义为 `TASKDIALOGCONFIG` 类，它映射到本地 API 调用的相同结构上：

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Auto, Pack = 4)]
internal class TASKDIALOGCONFIG
{
    internal uint cbSize;
    internal IntPtr hwndParent;
    internal IntPtr hInstance;
    internal TASKDIALOG_FLAGS dwFlags;
    internal TASKDIALOG_COMMON_BUTTON_FLAGS dwCommonButtons;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszWindowTitle;
    internal TASKDIALOGCONFIG_ICON_UNION MainIcon;
    // NOTE: 32-bit union field, holds pszMainIcon as well
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszMainInstruction;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszContent;
    internal uint cButtons;
    internal IntPtr pButtons; // Ptr to TASKDIALOG_BUTTON structs
    internal int nDefaultButton;
    internal uint cRadioButtons;
    internal IntPtr pRadioButtons; // Ptr to TASKDIALOG_BUTTON structs
    internal int nDefaultRadioButton;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszVerificationText;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszExpandedInformation;
    [MarshalAs(UnmanagedType.LPWStr)]
    internal string pszExpandedControlText;
```

```
[MarshalAs(UnmanagedType.LPWStr)]
internal string pszCollapsedControlText;
internal TASKDIALOGCONFIG_ICON_UNION FooterIcon;
// NOTE: 32-bit union field, holds pszFooterIcon as well
[MarshalAs(UnmanagedType.LPWStr)]
internal string pszFooter;
internal PFTASKDIALOGCALLBACK pfCallback;
internal IntPtr lpCallbackData;
internal uint cxWidth;
}
```

要在 WPF 应用程序中使用常见控件的新版本，必须添加一个应用程序清单文件，以及对常见控件的第 6 个版本的引用：



可从
wrox.com
下载源代码

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1"
    xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
    xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
        processorArchitecture="*"
        publicKeyToken="6595b64144ccf1df"
        language="*" />
    </dependentAssembly>
  </dependency>
</asmv1:assembly>
```

代码段 TaskDialogDemo/app.manifest

Windows API 代码包中用于显示任务对话框的公共类是 `TaskDialog`。要显示简单的任务对话框，只需调用静态方法 `Show()`。简单的对话框如图 A-9 所示。



可从
wrox.com
下载源代码

```
TaskDialog.Show("Simple Task Dialog",
    "Additional Information", "Title");
```

代码段 TaskDialogDemo/MainWindow.xaml.cs

为了获得 `TaskDialog` 类的更多功能，应设置 `Caption`、`Text`、`StandardButtons` 和 `MainIcon` 属性。结果如图 A-10 所示。

```
var dlg1 = new TaskDialog();
dlg1.Caption = "Title";
dlg1.Text = "Some Information";
dlg1.StandardButtons =
    TaskDialogStandardButtons.Ok |
    TaskDialogStandardButtons.Cancel;
dlg1.Icon = TaskDialogStandardIcon.Information;
dlg1.Show();
```

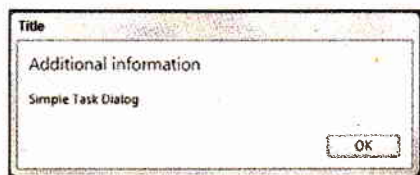


图 A-9

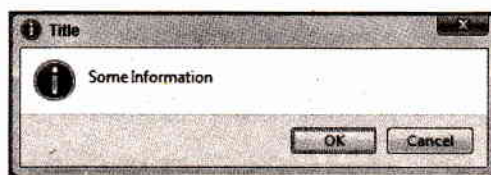


图 A-10

在任务对话框中，可以设置防火墙图标，它与命令链接一起显示。另外，还可以设置 `ExpansionMode` 属性来扩展它。使用 `TaskDialogExpandedInformationLocation` 枚举，可以指定是否应扩展内容或页脚。图 A-11 显示了折叠模式的任务对话框。

```
var dlg2 = new TaskDialog();
dlg2.Caption = "Title";
dlg2.Text = "Some Information";
dlg2.StandardButtons = TaskDialogStandardButtons.Yes |
    TaskDialogStandardButtons.No;
dlg2.Icon = TaskDialogStandardIcon.Shield;
dlg2.DetailsExpandedText = "Additional Text";
dlg2.DetailsExpandedLabel = "Less Information";
dlg2.DetailsCollapsedLabel = "More Information";
dlg2.ExpansionMode = TaskDialogExpandedDetailsLocation.ExpandContent;
dlg2.FooterText = "Footer Information";
dlg2.FooterIcon = TaskDialogStandardIcon.Information;
dlg2.Show();
```

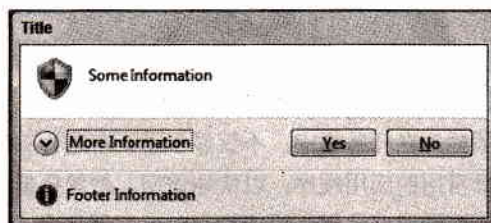


图 A-11

任务对话框也可以包含其他控件。在下面的代码段中，创建一个任务对话框，它包含两个单选按钮、一个命令链接和一个进度控件。上一节介绍了命令链接，其实，命令链接在任务对话框中使用得非常频繁。图 A-12 在内容区域中显示了带控件的任务对话框。当然，还可以将展开模式与控件组合起来。

```
var radio1 = new TaskDialogRadioButton();
radio1.Name = "radio1";
radio1.Text = "One";
var radio2 = new TaskDialogRadioButton();
radio2.Name = "radio2";
radio2.Text = "Two";

var commandLink = new TaskDialogCommandLink();
commandLink.Name = "link1";
commandLink.ShowElevationIcon = true;
```

```
commandLink.Text = "Information";
commandLink.Instruction = "Sample Command Link";

var progress = new TaskDialogProgressBar();
progress.Name = "progress";

progress.State = TaskDialogProgressBarState.Marquee;
var dlg3 = new TaskDialog();
dlg3.Caption = "Title";

dlg3.InstructionText = "Sample Task Dialog";
dlg3.Controls.Add(radiol1);
dlg3.Controls.Add(radiol2);
dlg3.Controls.Add(commandLink);
dlg3.Controls.Add(progress);
dlg3.StandardButtons = TaskDialogStandardButtons.Ok;
dlg3.Show();
```

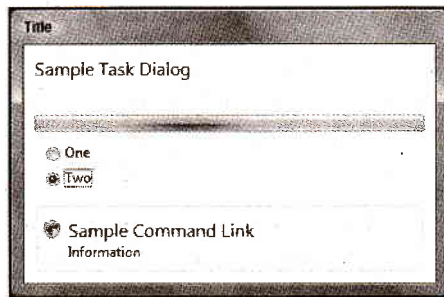


图 A-12

A.4.3 任务栏和跳转列表

Windows 7 有一个新任务栏。在任务栏中，不仅显示了正在运行的应用程序，还显示了快速访问图标。用户可以“锁定”最常用的应用程序，以快速访问。把鼠标悬停在任务栏的一项上时，就会查看当前正在运行的应用程序的预览。因为该项也可以有可见状态，所以用户从任务栏的项中接收反馈。在使用资源管理器复制某些大型文件时，会在资源管理器的项上看到进度信息。进度信息与可见状态一起显示。任务栏的另一个功能是跳转列表。在一个任务栏按钮上右击，就会打开跳转列表。可以为每个应用程序自定义这个列表。对于 Microsoft Outlook，可以直接进入收件箱、日历、联系人和任务，或者创建一封新的电子邮件。对于 Microsoft Word，可以打开最近的文档。

要在 WPF 应用程序中添加所有这些功能，可以通过 .NET Framework 4 中的 System.Windows.Shell 名称空间来完成。

本节的示例应用程序允许播放视频，在任务栏按钮上显示可见的反馈，说明视

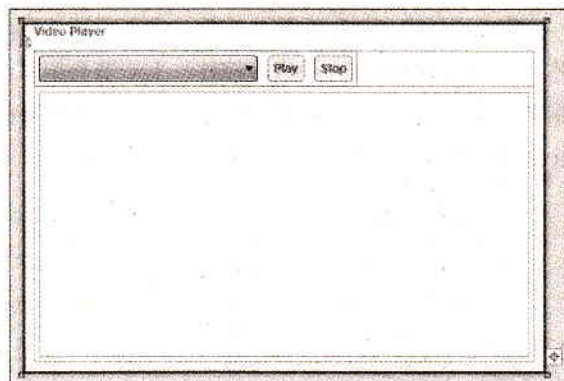


图 A-13

频是正在运行还是已停止，还允许直接从任务栏上启动和停止视频。该应用程序的主窗口包含一个两行网格，如图 A-13 所示。第一行包含一个用于显示可用的视频的 ComboBox，以及两个启动和停止视频的按钮。第二行包含一个用于播放视频的 MediaElement。

用户界面的 XAML 代码如下所示。按钮关联到 MediaCommands.Play 和 MediaCommands.Stop 命令上，它们分别映射到处理程序方法 OnPlay() 和 OnStop() 上。通过 MediaElement，把 LoadedBehavior 属性设置为 Manual，这样播放器不会在加载视频后立即播放。



可从
wrox.com
下载源代码

```
<Window.CommandBindings>
    <CommandBinding Command="MediaCommands.Play" Executed="OnPlay" />
    <CommandBinding Command="MediaCommands.Stop" Executed="OnStop" />
</Window.CommandBindings>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <StackPanel Grid.Row="0" Orientation="Horizontal"
        HorizontalAlignment="Left">
        <ComboBox x:Name="comboVideos" ItemsSource="{Binding}" Width="220"
            Margin="5" IsSynchronizedWithCurrentItem="True" />
        <Button x:Name="buttonPlay" Content="Play" Margin="5" Padding="4"
            Command="MediaCommands.Play" />
        <Button x:Name="buttonStop" Content="Stop" Margin="5" Padding="4"
            Command="MediaCommands.Stop" />
    </StackPanel>
    <MediaElement x:Name="player" Grid.Row="1" LoadedBehavior="Manual"
        Margin="5"
        Source="{Binding ElementName=comboVideos, Path=SelectedValue}" />
</Grid>
```

代码段 TaskbarDemo/MainWindow.xaml

为了配置任务栏上的项，System.Windows.Shell 名称空间给 Window 类包含了一个依赖属性，用于添加任务栏信息。TaskbarItemInfo 属性可以包含一个 TaskbarItemInfo 元素。通过 TaskbarItemInfo 属性，可以设置 Description 属性，后一个属性显示为工具提示信息。使用 ProgressState 和 ProgressValue 属性，可以给出应用程序当前状态的反馈。ProgressState 的类型是 TaskbarItemProgressState，它定义枚举值 None、Intermediate、Normal、Error 和 Paused。根据这个设置的值，会在任务栏的项上显示进度指示器。Overlay 属性允许定义一幅图像，该图像显示在任务栏的图标上。这个属性从代码隐藏中设置。

任务栏上的项可以包含一个 ThumbButtonInfoCollection，把它赋予 TaskbarItemInfo 的 ThumbButtonInfos 属性。这里可以添加 ThumbButtonInfo 类型的按钮，该按钮显示在应用程序的预览中。本示例包含两个 ThumbButtonInfo 元素，给其 Command 设置的命令与前面创建的播放和停止 Button 元素相同。有了这些按钮，就可以像使用应用程序中的其他 Button 元素那样控制应用程序。任务栏中用于按钮的图像从 StartImage 和 StopImage 键对应的资源中提取。

```
<Window.TaskbarItemInfo>
    <TaskbarItemInfo x:Name="taskBarItem" Description="Sample Application">
        <TaskbarItemInfo.ThumbButtonInfos>
```

```

        <ThumbButtonInfo IsEnabled="True" Command="MediaCommands.Play"
            CommandTarget="{Binding ElementName=buttonPlay}"
            Description="Play"
            ImageSource="{StaticResource StartImage}" />
        <ThumbButtonInfo IsEnabled="True" Command="MediaCommands.Stop"
            CommandTarget="{Binding ElementName=buttonStop}"
            Description="Stop"
            ImageSource="{StaticResource StopImage}" />
    </TaskbarItemInfo.ThumbButtonInfos>
</TaskbarItemInfo>
</Window.TaskbarItemInfo>

```



依赖属性参见第 27 章。

从 `ThumbButtonInfo` 元素中引用的图像在 `Window` 的 `Resources` 部分中定义。其中一幅图像是一个浅绿色的椭圆；另一幅图像是两条橙红色的线条。

```

<Window.Resources>
    <DrawingImage x:Key="StopImage">
        <DrawingImage.Drawing>
            <DrawingGroup>
                <DrawingGroup.Children>
                    <GeometryDrawing>
                        <GeometryDrawing.Pen>
                            <Pen Thickness="5" Brush="OrangeRed" />
                        </GeometryDrawing.Pen>
                        <GeometryDrawing.Geometry>
                            <GeometryGroup>
                                <LineGeometry StartPoint="0,0"
                                    EndPoint="20,20" />
                                <LineGeometry StartPoint="0,20"
                                    EndPoint="20,0" />
                            </GeometryGroup>
                        </GeometryDrawing.Geometry>
                    </GeometryDrawing>
                </DrawingGroup.Children>
            </DrawingGroup>
        </DrawingImage.Drawing>
    </DrawingImage>
    <DrawingImage x:Key="StartImage">
        <DrawingImage.Drawing>
            <DrawingGroup>
                <DrawingGroup.Children>
                    <GeometryDrawing Brush="LightGreen">
                        <GeometryDrawing.Geometry>
                            <EllipseGeometry RadiusX="20" RadiusY="20"
                                Center="20,20" />
                        </GeometryDrawing.Geometry>
                    </GeometryDrawing>
                </DrawingGroup.Children>
            </DrawingGroup>
        </DrawingImage.Drawing>
    </DrawingImage>

```

```

        </DrawingGroup>
    </DrawingImage.Drawing>
</DrawingImage>
</Window.Resources>

```

在代码隐藏中, 把特殊文件夹 My Videos 中的所有视频都赋予 Window 对象的 DataContext, 而且由于 ComboBox 是数据绑定的, 所以这些视频都列在 ComboBox 中。在 OnPlay() 和 OnStop() 事件处理程序中, 调用 MediaElement 的 Play() 和 Stop() 方法。为了在任务栏的项、播放和停止视频时显示可见的反馈, 需要访问图像资源, 并把它们赋予 TaskbarItemInfo 元素的 Overlay 属性。



可从
wrox.com
下载源代码

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shell;

namespace Wrox.ProCSharp.Windows7
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            string videos = Environment.GetFolderPath(
                Environment.SpecialFolder.MyVideos);
            this.DataContext = Directory.EnumerateFiles(videos);
        }

        private void OnPlay(object sender, ExecutedRoutedEventArgs e)
        {
            object image = TryFindResource("StartImage");
            if (image is ImageSource)
                taskBarItem.Overlay = image as ImageSource;
            player.Play();
        }

        private void OnStop(object sender, ExecutedRoutedEventArgs e)
        {
            object image = TryFindResource("StopImage");
            if (image is ImageSource)
                taskBarItem.Overlay = image as ImageSource;
            player.Stop();
        }
    }
}

```

代码段 TaskbarDemo/MainWindow.xaml.cs

现在就可以运行应用程序, 从任务栏上启动和停止视频, 查看可见的反馈, 如图 A-14 所示。



图 A-14

要自定义跳转列表，应给应用程序类添加一个 `JumpList`。为此，可以在代码中调用静态方法 `JumpList.SetJumpList()`，或者把 `JumpList` 元素添加为 `Application` 元素的子元素。

示例代码在代码隐藏中创建跳转列表。`JumpList.SetJumpList()` 应是应用程序对象的第一个参数，而应用程序对象从 `Application.Current` 属性中返回。第二个参数是 `JumpList` 类型的对象。`jumpList` 用 `JumpList` 构造函数创建，并给该构造函数传递 `JumpItem` 元素和两个布尔值。跳转列表默认包含常用的项和最近的项。通过设置布尔值，就可以影响这个默认行为。可以添加到 `JumpList` 中的项派生自基类 `JumpItem`。可用的类有 `JumpTask` 和 `JumpPath`。通过 `JumpTask` 类可以定义一个程序，该程序应在用户从跳转列表中选择了这一项时启动。`JumpTask` 定义的属性可用于启动应用程序和给用户提供的信息，包括 `CustomCategory`、`Title`、`Description`、`ApplicationPath`、`IconResourcePath`、`WorkingDirectory` 和 `Arguments`。通过在代码段中定义的 `JumpTask`，并用 `readme.txt` 文档启动和初始化 `notepad.exe`。利用 `JumpPath` 项，可以定义一个文件，该文件应在跳转列表中列出，以便从该文件中启动应用程序。`JumpPath` 定义一个 `Path` 属性，用于指定文件名。使用 `JumpPath` 项要求用应用程序注册文件扩展名。否则，项的注册会被拒绝。为了找出项被拒绝的原因，可以给 `JumpItemRejected` 事件注册一个处理程序。这样就可以获得一个被拒绝项(`RejectedItems`)的列表和拒绝它们的原因(`RejectedReasons`)。

```
var jumpItems = new List<JumpTask>();
var workingDirectory = Environment.CurrentDirectory;
var windowsDirectory = Environment.GetFolderPath(
    Environment.SpecialFolder.Windows);
var notepadPath = System.IO.Path.Combine(windowsDirectory,
    "Notepad.exe");

jumpItems.Add(new JumpTask
{
    CustomCategory="Read Me",
    Title="Read Me",
    Description="Open read me in Notepad.",
    ApplicationPath=notepadPath,
    IconResourcePath=notepadPath,
    WorkingDirectory=workingDirectory,
    Arguments="readme.txt"
});

var jumpList = new JumpList(jumpItems, true, true);
jumpList.JumpItemsRejected += (sender1, e1) =>
{
    var sb = new StringBuilder();
    for (int i = 0; i < e1.RejectedItems.Count; i++)
```

```

{
    if (e1.RejectedItems[i] is JumpPath)
        sb.Append((e1.RejectedItems[i] as JumpPath).Path);
    if (e1.RejectedItems[i] is JumpTask)
        sb.Append((e1.RejectedItems[i] as JumpTask).
            ApplicationPath);
    sb.Append(e1.RejectionReasons[i]);
    sb.AppendLine();
}
MessageBox.Show(sb.ToString());
};

JumpList.SetJumpList(Application.Current, jumpList);

```

运行应用程序，并单击鼠标右键，就会看到跳转列表，如图 A-15 所示。

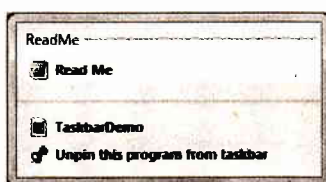


图 A-15



为了从 Windows 窗体应用程序中使用任务栏和跳转列表，可以使用 Windows API 代码包中定义的扩展。

A.6 小结

本附录介绍了可用于 Windows 7 和 Windows Server 2008 R2 中的各种功能，这些功能对应用程序的开发非常重要。

Microsoft 多年来制定了规则，指出非管理应用程序不需要管理权限。因为许多应用程序都不符合这个要求，所以操作系统现在限制了 UAC。用户必须显式地给应用程序提升管理权限。本章介绍了这如何影响应用程序。

本附录还介绍了 Windows Vista 引入的几个对话框，以获得更好的用户交互，其中包括替代消息框的新任务对话框，以及扩展为按钮控件的命令链接。Windows 7 的新对话框是任务栏上的项和跳转列表。

其他章节还介绍了只能用于 Windows Vista 和 Windows Server 2008 及以后版本的更多功能：

- 第 19 章讨论了事件日志功能和 Windows 事件跟踪(ETW)
- 第 21 章给出了下一代加密技术(CNG)的信息，这是一个 Crypto API。
- 第 23 章介绍了基于文件和基于注册表的事务。
- 第 43 章使用激活服务(WAS)驻留 WCF 服务。

第 48 章

使用 GDI+绘图

本章主要内容:

- 绘图规则
- 颜色和安全调色板
- 钢笔和笔刷
- 线条和简单图形
- BMP 图像和其他图像文件
- 绘制文本
- 字体和字体系列
- 处理打印

本书有许多章节介绍用户交互和.NET Framework, 第 39 章主要介绍了如何显示对话框或 SDI、MDI 窗口, 以及如何把各种控件放在这些窗口上, 如按钮、文本框和列表框。它还讨论了在 Windows 窗体中使用许多 Windows 窗体控件处理各种数据源中的数据。

这些标准控件的功能非常强大, 使用它们就可以获得许多应用程序的完整用户界面。但是, 有时还需要在用户界面上有更大的灵活性。例如, 要在窗口的精确位置以给定的字体绘制文本, 或者不使用图像框控件显示图像, 或者绘制简单的形状和其他图形。这些都不能使用第 39 章介绍的控件来完成。要显示这种类型的输出, 应用程序必须直接告诉操作系统需要在其窗口的什么地方显示什么内容。

在这个过程中, 还需要使用各种帮助对象, 包括钢笔(用于定义线条的特征)、画笔(用于定义区域的填充方式)和字体(用于定义文本字符的形状)。我们还将介绍设备如何识别和显示不同的颜色。

下面首先讨论 GDI+(Graphics Device Interface)技术。GDI+由.NET 基类集组成, 这些基类可用于在屏幕上完成自定义绘图, 能把合适的指令发送到图形设备的驱动程序上, 确保在屏幕上显示正确的输出(或打印到打印件中)。

48.1 理解绘图规则

本节讨论一些基本规则, 只有理解了它们, 才能开始在屏幕上绘图。首先概述 GDI, GDI+技术就建立在 GDI 的基础上, 然后说明它与 GDI+的关系。接着介绍几个简单的例子。

48.1.1 GDI 和 GDI+

一般来说，Windows 的一个优点 (实际上通常是现代操作系统的优点) 是它可以让开发人员不考虑特定设备的细节。例如，不需要理解硬盘设备驱动程序，只需在相关的.NET 类中调用合适的方法 (在.NET 推出之前，使用等价的 Windows API 函数)，就可以编程读写磁盘上的文件。这条规则也适用于绘图。计算机在屏幕上绘图时，它把指令发送给视频卡。问题是市面上有几百种不同的视频卡，大多数有不同的指令集和功能。如果把这个考虑在内，在应用程序中为每个视频卡驱动程序编写在屏幕上绘图的特定代码，这样的应用程序就根本不可能编写出来。这就是为什么在 Windows 最早期的版本中就有 Windows 图形设备界面(Graphical Device Interface, GDI)的原因。

GDI 提供了一个抽象层，隐藏了不同视频卡之间的区别，这样就可以调用 Windows API 函数完成指定的任务了，GDI 会在内部指出在客户运行特定的代码时，如何让客户端的视频卡完成要绘制的图形。GDI 还可以完成其他任务。大多数计算机都有多个显示设备——例如，显示器和打印机。GDI 成功地使应用程序所使用的打印机看起来与屏幕一样。如果要打印某些东西，而不是显示它，只需告诉系统输出设备是打印机，再用相同的方式调用相同的 Windows API 函数即可。

可以看出，DC(设备环境)是一个功能非常强大的对象，在 GDI 下，所有的绘图工作都必须通过设备环境来完成。DC 甚至可用于不涉及在屏幕或某些硬件设备上绘图的其他操作，例如，在内存中修改图像。

GDI 给开发人员提供了一个相当高级的 API，但它仍是一个基于旧的 Windows API 并且有 C 语言风格函数的 API，所以使用起来不是很方便。GDI+在很大程度上是 GDI 和应用程序之间的一层，提供了更直观且基于继承性的对象模型。尽管 GDI+基本上是 GDI 的一个包装器，但 Microsoft 已经能通过 GDI+提供新功能了，它还有一些性能方面的改进。

.NET 基类库的 GDI+部分非常庞大，本章不解释其功能。这是因为只要解释库中的一小部分内容，就会把本章变成一个仅列出类和方法的参考指南。而理解绘图的基本规则更重要；这样您就可以自己研究这些类。当然，关于 GDI+中类和方法的完整列表，可以参阅 SDK 文档。



有 VB 6 背景的开发人员会发现，自己并不熟悉绘图过程涉及的概念，因为 VB 6 的重点是处理绘图的控件。有 C++/MFC 背景的开发人员则比较熟悉这个领域，因为 MFC 要求开发人员使用 GDI 更多地控制绘图过程。但是，即使您具备很好的 GDI 背景知识，也会发现本章有许多新内容。

1. GDI+名称空间

表 48-1 列出了 GDI+基类的主要名称空间。
本章使用的几乎所有的类、结构等都包含在 System.Drawing 名称空间中。

表 48-1

名 称 空 间	说 明
System.Drawing	包含与基本绘图功能有关的大多数类、结构、枚举和委托
System.Drawing.Drawing2D	为大多数高级 2D 和矢量绘图操作提供了支持，包括消除锯齿、几何变形和图形路径

(续表)

名 称 空 间	说 明
System.Drawing.Imaging	包括有助于处理图像(位图、GIF 文件等)的各种类
System.Drawing.Printing	包含把打印机或打印预览窗口作为“输出设备”时使用的类
System.Drawing.Design	包含一些预定义的对话框、属性表和其他用户界面元素，与在设计期间扩展用户界面相关
System.Drawing.Text	包含对字体和字体系列执行更高级操作的类

2. 设备上下文和 Graphics 对象

在 GDI 中，识别输出设备的方式是使用设备上下文(DC)对象。该对象存储特定设备的信息，并能把 GDI API 函数调用转换为要发送给该设备的任何指令。还可以查询设备上下文对象，确定对应的设备有什么功能(例如，打印机是彩色的，还是黑白的)，以便据此调整输出结果。如果要求设备完成它不能完成的任务，设备上下文对象就会检测到，并采取相应的措施(这取决于具体的情形，可能抛出一个异常，或修改请求从而获得与该设备的功能最相近的匹配)。

DC 对象不仅可以处理硬件设备，还可以用作 Windows 的一个桥梁，因此能考虑到 Windows 绘图的要求或限制。例如，如果 Windows 知道只有一小部分应用程序的窗口需要重新绘制，DC 就可以捕获和取消在该区域外绘图的工作。因为 DC 与 Windows 的关系非常密切，所以通过设备上下文来工作就可以在其他方面简化代码。

例如，硬件设备需要知道在什么地方绘制对象，通常它们需要相对于屏幕(或输出设备)左上角的坐标。但应用程序可能使用自己的坐标系统，在自己的窗口的工作区(用于绘图的窗口)的特定位置上绘图。而因为窗口可以放在屏幕上的任何位置，用户可以随时移动它，所以在两个坐标系统之间转换就是一个比较困难的任务。然而，DC 总是知道窗口在什么地方，并能自动进行这种转换。

在 GDI+中，DC 包装在.NET 基类 System.Drawing.Graphics 中。大多数绘图工作都是调用关于 Graphics 实例的方法完成的。实际上，因为正是 Graphics 类负责处理大多数绘图操作，所以 GDI+中几乎没有操作不涉及 Graphics 实例。于是，理解如何处理这个对象是理解如何使用 GDI+在显示设备上绘图的关键。

48.1.2 绘制图形

下面用一个小示例 DisplayAtStartup 来说明如何在应用程序的主窗口中绘图。本章的示例都在 Visual Studio 2010 中创建为 C# Windows 应用程序。对于这种类型的项目，代码向导会提供一个类 Form1，它派生自 System.Windows.Forms，窗体表示应用程序的主窗口。还会生成一个类 Program(在 Program.cs 文件中)，它表示应用程序的主起点。除非特别声明，否则在所有的代码示例中，新代码或修改过的代码都添加到向导生成的代码中(可以从 Wrox 网站 www.wrox.com 或随书附赠光盘中找到示例代码)。



在.NET 的用法中，当说到显示各种控件的应用程序时，“窗口”大都用术语“窗体”来代替，窗体表示一个矩形对象，它代表应用程序占据了屏幕上的一块区域。本章使用术语“窗口”，因为在手工绘图时，它更有意义。当谈到用于实例化窗体/窗口的.NET 类时，使用术语“窗体”。最后，“绘图”或“绘制”可以互换使用，以描述在屏幕或其他显示设备上显示一些项的过程。

第一个示例只创建一个窗体，并在启动窗体时在构造函数中绘制它。注意，这并不是在屏幕上绘图的最佳方式或正确方式，因为这个示例并不能在启动后按照需要重新绘制窗体。但利用这个示例，我们不必做太多的工作，就可以说明绘图的许多问题。

对于这个示例，启动 Visual Studio 2010，创建一个 Windows From Application 项目。首先把窗体的背景色设置为白色。把这行代码放在 `InitializeComponent()` 方法的后面，这样 Visual Studio 2010 就会识别该行命令，并能够改变窗体的设计视图的外观。单击 `Form1.cs` 文件旁边的箭头图标(这会展开与 `Form1.cs` 文件相关的文件层次结构)，就可以看到 `Form1.Designer.cs` 文件。正是在这个文件中发现了 `InitializeComponent()` 方法。本来也可以使用设计视图设置背景色，但这会导致自动添加相同的代码：



可从
wrox.com
下载源代码

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
}
```

代码段 `DrawingShapes.sln`

接着，给 `Form1` 构造函数添加代码。使用窗体的 `CreateGraphics()` 方法创建一个 `Graphics` 对象，这个对象包含绘图时需要使用的 Windows 设备上下文。创建的设备上下文与显示设备相关，也与这个窗口相关。



可从
wrox.com
下载源代码

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

    Graphics dc = CreateGraphics();
    Show();
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0, 0, 50, 50);
    Pen redPen = new Pen(Color.Red, 2);
    dc.DrawEllipse(redPen, 0, 50, 80, 60);
}
```

代码段 `DrawingShapes.sln`

然后调用 `Show()` 方法显示窗口。之所以让该窗口立即显示，是因为在该窗口显示出来之前，我们不能做任何工作——没有可供绘图的地方。

最后，显示一个矩形，其坐标是(0,0)，宽度和高度是 50，再绘制一个椭圆，其坐标是(0, 50)，

宽度是 80，高度是 50。注意坐标(x,y)表示从窗口的工作区左上角开始向右的 x 个像素，向下的 y 个像素——这些坐标从要显示的图形的左上角开始计算。

我们使用的 `DrawRectangle()`和 `DrawEllipse()`重载方法分别带 5 个参数。第一个参数是 `System.Drawing.Pen` 类的实例。`Pen` 是许多帮助绘图的辅助对象中的一个，它包含如何绘制线条的信息。第一个 `Pen` 表示线条应是蓝色的，其宽度为 3 个像素；第二个 `Pen` 表示线条应是红色的，其宽度为两个像素。后面的 4 个参数是坐标和大小。对于矩形，它们分别表示矩形的左上角坐标(x,y)、其宽度和高度。对于椭圆，这些数值的含义相同，但它们是指椭圆假想的外接矩形，而不是椭圆本身。运行代码，会得到如图 48-1 所示的图形。当然，本书不是彩书，所以看不到颜色。

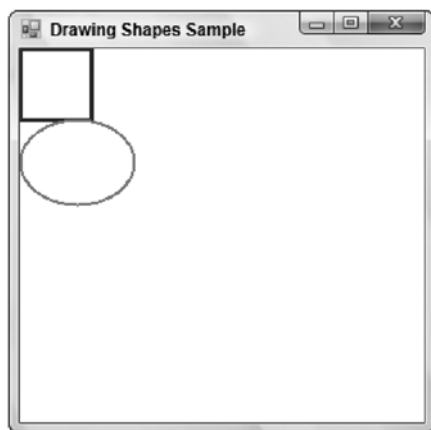


图 48-1

图 48-1 说明了两个问题。首先，用户可以很清楚地看到在窗口中工作区的位置。它是一块白色的区域——该区域受到 `BackColor` 属性设置的影响。还要注意，矩形放在该区域的一角，因为它指定了坐标(0,0)。其次，注意椭圆的顶部与矩形有轻度的重叠，这与代码中给出的坐标有点不同。这是因为 Windows 在重叠的区域上放置了矩形和椭圆的边框。在默认情况下，Windows 会试图把图形边框所在的线条放在中心位置——但这并不是总能做到的，因为线条是以像素为单位来绘制的，但每个图形的边框理论上通常位于两个像素之间。结果，1 个像素宽的线条就会正好位于图形的顶边和左边的里面，而在右边和底边的外面。这样，从严格意义上讲，相邻图形的边框就会有一个像素的重叠。我们指定的线条宽度比较大，因此重叠区域也会比较大。设置 `Pen.Alignment` 属性(详见 SDK 文档说明)，就可以改变默认的操作方式，但这里使用默认的操作方式就足够了。

但如果运行这个示例，就会注意到窗体的行为有点奇怪。如果把它放在那里，或者用鼠标在屏幕移动该窗体，它就工作正常。但如果最小化该窗体，再还原它，绘制好的图形就不见了。如果在该窗体上拖动另一个窗口，使之只遮挡一部分图形，再拖动该窗口远离这个窗体，临时被挡住的部分就消失了，只剩下一半椭圆或矩形了！

这是怎么回事？其原因是，如果窗口的一部分被隐藏了，Windows 通常会立即丢弃与其中显示的内容相关的所有信息。这是必需的，否则存储屏幕数据的内存量就会是个天文数字。一般的计算机在运行时，视频卡设置为显示 1024×768 像素、24 位彩色模式，这表示屏幕上的每个像素占据 3 个字节，于是显示整个屏幕就需要 2.25MB(本章后面会说明 24 位颜色的含义)。但是，用户常常让任务栏上有 10 个或 20 个最小化的窗口。下面考虑一种最糟糕的情况：20 个窗口，每个窗口如果没

有最小化，它就占据整个屏幕。如果 Windows 存储了这些窗口包含的可视化信息，当用户还原它们时，它们就会有 45MB。目前，比较好的图形卡有 512MB 的内存，可以应付这种情况，但在几年前图形卡有 256MB 的内存就不错了。多余的部分需要存储在计算机的主内存中。许多人仍在使用旧计算机，一些人甚至还在使用 256MB 的图形卡。很显然，Windows 不可能这样管理用户界面。

在窗口的任何某一部分消失时，隐藏的那些像素也就丢失了。因为 Windows 释放了保存这些像素的内存。但要注意，窗口的一部分被隐藏了，当它检测到窗口不再被隐藏时，就请求拥有该窗口的应用程序重新绘制其内容。这条规则有一些例外——通常窗口的一小部分被挡住的时间比较短(例如，从主菜单中选择一个菜单项，该菜单项向下拉出，临时挡住了下面的部分)。但一般情况下，如果窗口的一部分被挡住，应用程序就需要在以后重新绘制它。

这就是示例应用程序的问题的根源。我们把绘图代码放在 Form1 的构造函数中，这些代码仅在应用程序启动时调用一次，不能在以后需要时再次调用该构造函数，重新绘制图形。

在使用 Windows 窗体的服务器控件时，不需要知道如何完成上述任务，这是因为标准控件非常专业，能在 Windows 需要时重新绘制它们自己。这是编写控件时不需要担心实际绘图过程的原因之一。如果要应用程序在屏幕上绘图，还需要在 Windows 要求重新绘制窗口的全部或部分时，确保应用程序会正确响应。下一节将修改这个示例，确保应用程序的正确响应。

48.1.3 使用 OnPaint()方法绘制图形

上面的解释让您觉得绘制自己的用户界面是比较复杂的，实际上并非如此。让应用程序在需要时绘制自身是非常简单的。

Windows 会触发 Paint 事件通知应用程序完成一些重新绘制的要求。有趣的是，Form 类已经实现了这个事件的处理程序，因此不需要再添加处理程序了。Paint 事件的 Form1 处理程序处理虚方法 OnPaint()的调用，并给它传递一个参数 PaintEventArgs，这表示，我们只需重写 OnPaint()方法执行绘图操作。

我们选择重写 OnPaint()方法，也可以为 Paint 事件添加自己的事件处理程序(如 Form1_Paint()方法)来得到相同的结果，其方式与为任何其他 Windows 窗体事件添加处理程序一样。后一种方法更方便一些，因为可以通过 Visual Studio 2010 的属性窗口添加新的事件处理程序，不必输入某些代码。但是我们采用重写 OnPaint()方法的方式要略为灵活一些，因为这样可以控制何时调用基类窗口进行处理，并允许避免把控件的处理程序关联到它自己的事件上。

下面新建一个 Windows 应用程序 DrawShapes 来完成这个操作。与以前一样，使用属性窗口把背景色设置为白色，再把窗体的文本改为 DrawShapes Sample，接着在 Form1 类的自动生成代码中添加如下代码：



可从
wrox.com
下载源代码

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Pen bluePen = new Pen(Color.Blue, 3);
    dc.DrawRectangle(bluePen, 0,0,50,50);
    Pen redPen = new Pen(Color.Red, 2);
    dc.DrawEllipse(redPen, 0, 50, 80, 60);
}
```

代码段 DrawingShapes.sln

注意，OnPaint()方法声明为 protected。因为 OnPaint()方法一般在类的内部使用，所以类外部的其他代码不必知道存在 OnPaint()方法。

PaintEventArgs 是派生自 EventArgs 类的一个类，一般用于传递有关事件的信息。PaintEventArgs 类有另外两个属性，其中比较重要的是 Graphics 实例，它们主要用于优化窗口中需要绘制的部分。这样就不必调用 CreateGraphics()方法在 OnPaint()方法中获取设备上下文了——用户总是可以得到设备上下文。后面将介绍其他属性。这个属性包含窗口的哪些部分需要重新绘制的详细信息。

在 OnPaint()方法的实现代码中，首先从 PaintEventArgs 类中引用 Graphics 对象，再像以前那样绘制图形。最后调用基类的 OnPaint()方法，这一步非常重要。我们重写了 OnPaint()方法，完成了绘图工作，但 Windows 在绘图过程中可能会执行一些它自己的辅助工作。这些工作都在.NET 基类的 OnPaint()方法中完成。



对于这个示例，删除 base.OnPaint()的调用似乎并没有任何影响，但不要试图删除这个调用。这样有可能阻止 Windows 正确执行任务，结果是无法预料的。

在应用程序第一次启动和窗口第一次显示时，也调用了 OnPaint()方法，所以不需要在构造函数中复制绘图代码。

运行这段代码，得到的结果将与前面的示例的结果相同，但现在，当最小化窗口或隐藏它的一部分时，应用程序会正确执行。

48.1.4 使用剪切区域

上一节的 DrawShapes 示例说明了在窗口中绘图的主要规则，但它并不是很高效。原因是它试图绘制窗口中的所有内容，而没有考虑需要绘制多少内容。如图 48-2 所示，运行 DrawShapes 示例，当该示例在屏幕上绘图时，打开另一个窗口，把它移动到 DrawShapes 窗体上，使之遮挡一部分窗体。

但移动重叠的窗口时，DrawShapes 窗口会再次全部显示出来，Windows 通常会给窗体发送一个 Paint 事件，要求它重新绘制本身。因为矩形和椭圆都位于工作区的左上角，所以在任何时候它们都是可见的，在本例中不需要重新绘制这部分，而只需要重新绘制白色背景区域。但是，Windows 并不知道这一点，它认为应触发 Paint 事件，调用 OnPaint()方法的实现代码。OnPaint()方法不必重新绘制矩形和椭圆。

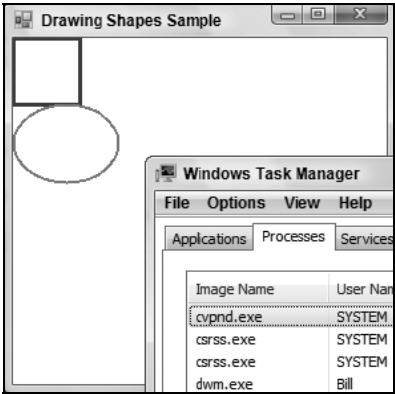


图 48-2

在本例中，没有重新绘制图形。原因是我们使用了设备上下文。Windows 将利用重新绘制哪些区域所需要的信息预先初始化设备上下文。在 GDI 中，被标记出来的重绘区域称为无效区域，但在 GDI+ 中，该术语通常改为剪切区域。设备上下文能识别这个区域。于是，它截取在这个区域外部的绘图操作，且不把相关的绘图命令传送给图形卡。这听起来不错，但仍有一个潜在的性能损失。在确定是在无效区域外部绘图前，我们不知道必须进行多少设备环境处理。在某些情况下，要处理的任务比较多，因为计算哪些像素需要改变为什么颜色，将会频繁占用处理器(好的图形卡会提供硬件加速，对此有一定的帮助)。

其底线是让 Graphics 实例完成在无效区域外部的绘图工作，这肯定会浪费处理器的时间，减慢应用程序的运行。在设计优良的应用程序中，代码将执行一些检查，以查看需要进行哪些绘图工作，然后调用相关的 Graphics 实例的方法。本节将编写一个新示例 DrawShapesWithClipping，方法是修改 DisplayShapes 示例，只完成需要的重新绘制工作。在 OnPaint() 方法的代码中，进行一个简单的测试，看看无效区域是否与需要绘制的区域重叠，如果是，就调用绘图方法。

首先，需要获得剪切区域的信息。这需要使用 PaintEventArgs 类的另一个属性——ClipRectangle，它包含要重绘区域的坐标，并封装在一个结构的实例 System.Drawing.Rectangle 中。Rectangle 是一个相当简单的结构，它包含 4 个属性：Top、Bottom、Left 和 Right。它们分别包含矩形的上下边的垂直坐标、左右边的水平坐标。

接着，需要确定进行什么测试，以决定是否进行绘制。这里进行一个简单的测试。注意，在绘图过程中，矩形和椭圆完全包含在从(0,0)点到(80,130)点的矩形工作区中，实际上，(82,132)点就已经在安全区域中了，因为线条大约偏离这个区域的外侧一个像素。所以我们要看看剪切区域的左上角是否在这个矩形区域内。如果是，就重新绘制；如果不是，就不必麻烦了。

下面是代码：



```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    if (e.ClipRectangle.Top < 132 && e.ClipRectangle.Left < 82)
    {
        Pen bluePen = new Pen(Color.Blue, 3);
        dc.DrawRectangle(bluePen, 0,0,50,50);
        Pen redPen = new Pen(Color.Red, 2);
        dc.DrawEllipse(redPen, 0, 50, 80, 60);
    }
}
```

代码段 DrawingShapes.sln

注意，显示的结果与以前显示的结果完全相同——但这次进行了早期测试，确定了哪些区域不需要绘制，提高了性能。还要注意这个示例关于是否进行绘图的测试是非常粗略的。还可以进行更精细的测试，分别确定矩形或者椭圆是否要重新绘制。这里有一个平衡。可以在 OnPaint() 方法进行更复杂的测试，以提高性能，但也可以使 OnPaint() 方法的代码更复杂一些。进行一些测试总是值得的，因为编写一些代码，可以更多地解除 Graphics 实例之外的绘制内容，Graphics 实例只是盲目地执行绘图命令。

48.2 测量坐标和区域

在上一个示例中，我们遇到了基本结构 `Rectangle`，它用于表示矩形的坐标。GDI+实际上使用几个类似的结构来表示坐标或区域。表 48-1 列出了几个结构，它们都是在 `System.Drawing` 名称空间中定义的，如表 48-2 所示。

表 48-2

结 构	主要的公共属性
Point 和 PointF	X、Y
Size 和 SizeF	Width、 Height
Rectangle 和 RectangleF	Left、 Right、 Top、 Bottom、 Width、 Height、 X、 Y、 Location、 Size

注意，其中的许多对象都有许多其他属性、方法或运算符重载，这里没有列出来。本节只讨论某些最重要的成员。

48.2.1 Point 和 PointF 结构

从概念上讲，`Point` 在这些结构中最简单的。在数学上，它等价于一个二维矢量。它包含两个公共整型属性，它表示与某个特定位置的水平和垂直距离(在屏幕上)，如图 48-3 所示。

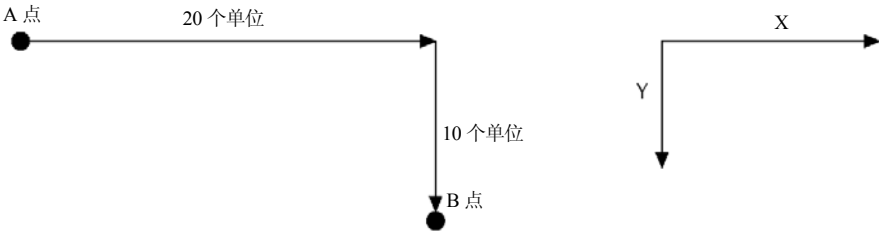


图 48-3

为了从 A 点到 B 点，需要水平移动 20 个单位，并向下垂直移动 10 个单位，在图 48-3 中标为 `x` 和 `y`，这就是它们的一般含义。下面的 `Point` 结构表示该线条：

```
Point ab = new Point(20, 10);
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```

`X` 和 `Y` 都是读写属性，因此可以在 `Point` 中设置这些值，例如：

```
Point ab = new Point();
ab.X = 20;
ab.Y = 10;
Console.WriteLine("Moved {0} across, {1} down", ab.X, ab.Y);
```

注意，按照惯例，水平坐标和垂直坐标表示为 `x` 和 `y`(小写)，但相应的 `Point` 属性是 `X` 和 `Y`(大写)，因为在 C# 中，公共属性的一般约定是名称以一个大写字母开头。

`PointF` 与 `Point` 完全相同，但 `X` 和 `Y` 属性的类型是 `float`，而不是 `int`。`PointF` 用于坐标不是整数

值的情况。已经为这些结构定义了数据类型强制转换,这样就可以把 `Point` 隐式转换为 `PointF`(注意,因为 `Point` 和 `PointF` 是结构,这种强制转换实际上涉及数据的复制)。但没有相应的逆过程,要把 `PointF` 转换为 `Point`,必须复制对应的值,或使用下面的 3 个转换方法 `Round()`、`Truncate()`和 `Ceiling()`中的一个:

```
PointF abFloat = new PointF(20.5F, 10.9F);
// converting to Point
Point ab = new Point();
ab.X = (int)abFloat.X;
ab.Y = (int)abFloat.Y;
Point ab1 = Point.Round(abFloat);
Point ab2 = Point.Truncate(abFloat);
Point ab3 = Point.Ceiling(abFloat);
// but conversion back to PointF is implicit
PointF abFloat2 = ab;
```

下面看看测量单位。在默认情况下, GDI+把单位看作是屏幕(或打印机,无论图形设备是什么,都可以这样认为)上的像素。这就是 `Graphics` 对象的方法把它们接收到的坐标看作其参数的方式。例如, `new Point(20,10)`点表示在屏幕上水平向右移动 20 个像素,垂直向下移动 10 个像素。通常这些像素从窗口客户区域的左上角开始测量,如上面的示例所示。但是,情况并不总是如此。例如,在某些情况下,需要以整个窗口的左上角(包括其边框)为原点来绘图,甚至以屏幕的左上角为原点。但在大多数情况下,除非文档特别说明,否则都可以假定像素值是相对于客户区域的左上角。

在分析了滚动后,本章将在讨论 3 种坐标系统(世界、页面和设备坐标)时介绍这个主题。

48.2.2 Size 和 SizeF 结构

与 `Point` 和 `PointF` 一样, `Size` 也有两个变体。`Size` 结构用于 `int` 类型, `SizeF` 用于 `float` 类型。除此之外, `Size` 和 `SizeF` 是完全相同的。下面主要讨论 `Size` 结构。

在许多情况下, `Size` 结构与 `Point` 结构是相同的,它也有两个整型属性,分别表示水平距离和垂直距离。主要区别是这两个属性的名称不是 `X` 和 `Y`,而是 `Width` 和 `Height`。图 48-3 可以表示为:

```
Size ab = new Size(20,10);
Console.WriteLine("Moved {0} across, {1} down", ab.Width, ab.Height);
```

严格地讲, `Size` 在数学上与 `Point` 表示的含义相同;但在概念上它的使用方式略有不同。`Point` 用于说明实体在什么地方,而 `Size` 用于说明实体有多大。但是, `Size` 和 `Point` 是紧密相关的,目前甚至支持它们之间的相互转换:

```
Point point = new Point(20, 10);
Size size = (Size) point;
Point anotherPoint = (Point) size;
```

例如,考虑前面绘制的矩形,其左上角的坐标是(0,0),大小是(50,50)。这个矩形的大小是(50,50),可以用一个 `Size` 实例来表示。其右下角的坐标也是(50,50),但它由一个 `Point` 实例来表示。要理解这个区别,假定在另一个位置绘制该矩形,其左上角的坐标是(10,10):

```
dc.DrawRectangle(bluePen, 10,10,50,50);
```

现在其右下角的坐标是(60,60)，但大小不变，仍是(50,50)。

因为 Point 和 Size 结构的加法运算符都已经重载了，所以可以把一个 Size 加到 Point 结构上，得到另一个 Point 结构：



```
static void Main(string[] args)
{
    Point topLeft = new Point(10,10);
    Size rectangleSize = new Size(50,50);
    Point bottomRight = topLeft + rectangleSize;
    Console.WriteLine("topLeft = " + topLeft);
    Console.WriteLine("bottomRight = " + bottomRight);
    Console.WriteLine("Size = " + rectangleSize);
}
```

代码段 PointsAndSizes.sln

把这段代码作为控制台应用程序 PointAndSizes 来运行，会得到如图 48-4 所示的结果。

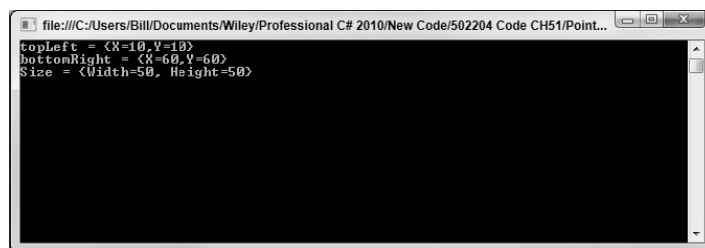


图 48-4

注意，这个结果也说明了 Point 和 Size 的 ToString()方法已被重写，并以{X,Y}格式显示该值。

也可以从一个 Point 结构中减去某个 Size 结构，得到另一个 Point 结构，还可以把两个 Size 加在一起，得到另一个 Size。但不能把一个 Point 结构加到另一个 Point 结构上。Microsoft 认为 Point 结构相加在概念上没有意义，所以不支持加法(+)运算符的任何重载版本进行这样的操作。

还可以在 Point 和 Size 结构之间进行显式的数据类型强制转换：

```
Point topLeft = new Point(10,10);
Size s1 = (Size)topLeft;
Point p1 = (Point)s1;
```

在进行这样的数据类型强制转换时，s1.Width 被赋予 topLeft.X 的值，s1.Height 被赋予 topLeft.Y 的值。因此 s1 包含(10,10)。p1 最终的值与 topLeft 的值相同。

48.2.3 Rectangle 和 RectangleF 结构

这两个结构表示一个矩形区域(通常在屏幕上)。与 Point 和 Size 一样，这里只介绍 Rectangle 结构，RectangleF 与 Rectangle 基本相同，但它的属性类型是 float，而 Rectangle 的属性类型是 int。

Rectangle 可以看作由一个 Point 和一个 Size 组成，其中 Point 表示矩形的左上角，Size 表示其大小。它的一个构造函数把 Point 和 Size 结构作为其参数。下面重写前面的 DrawShapes 示例的代码，绘制一个矩形：

```
Graphics dc = e.Graphics;
Pen bluePen = new Pen(Color.Blue, 3);
```

```
Point topLeft = new Point(0,0);
Size howBig = new Size(50,50);
Rectangle rectangleArea = new Rectangle(topLeft, howBig);
dc.DrawRectangle(bluePen, rectangleArea);
```

这段代码也使用 Graphics.DrawRectangle()方法的另一个重载版本，它的参数是 Pen 和 Rectangle 结构。

通过按顺序提供矩形的左上角水平和左上角垂直坐标，宽度和高度(它们都是数字)，可以构造一个 Rectangle:

```
Rectangle rectangleArea = new Rectangle(0, 0, 50, 50)
```

Rectangle 包含几个读写属性，如表 48-3 所示，可以用不同的属性组合来设置或提取它的维度。

表 48-3

属 性	说 明
int Left	左边界的 x 坐标
int Right	右边界的 x 坐标
int Top	顶边的 y 坐标
int Bottom	底边的 y 坐标
int X	与 Left 相同
int Y	与 Top 相同
int Width	矩形的宽度
int Height	矩形的高度
Point Location	左上角
Size Size	矩形的大小

注意，这些属性并非都是独立的。例如，设置 Width 会影响 Right 的值。

48.2.4 Region

Region 表示屏幕上一个包含复杂图形的区域。例如，图 48-5 中的阴影区域就可以用 Region 表示。

可以想象，初始化 Region 实例的过程相当复杂。从广义上看，可以指定哪些简单的图形组成这个区域，或指定绘制这个区域边界的路径。如果需要处理这样的区域，就应掌握 SDK 文档中的 Region 类。



图 48-5

48.3 调试须知

下面准备进行一些更高级的绘图工作。但首先介绍几个调试问题。如果在本章的示例中设置了断点，就会注意到调试图形例程不像调试程序的其他部分那样简单。这是因为进入和退出调试程序常常会把 `Paint` 消息传递给应用程序。结果是在 `OnPaint()` 重载方法上设置的断点会让应用程序一遍又一遍地绘制它本身，这样应用程序基本上就不能完成任何工作。

这是很典型的一种情形。要明白为什么应用程序没有正确显示，可以在 `OnPaint()` 事件中设置断点。应用程序会像期望的那样，遇到断点后，就会进入调试程序，此时在前台会显示开发环境 MDI 窗口。如果把开发环境设置为全屏显示，以便更易于观察所有的调试信息，就会完全隐藏目前正在调试的应用程序。

接着，检查某些变量的值，希望找出某些有用的信息。然后按 F5 键，告诉应用程序继续执行，完成某些处理后，看看应用程序在显示其他内容时会发生什么。但首先发生的是应用程序显示在前台中，Windows 快速检测到窗体再次可见，并提示给它发送了一个 `Paint` 事件。当然这表示程序遇到了断点。如果这就是我们想要的结果，那就很好。但更常见的是，我们希望以后在应用程序绘制了某些有趣的内容后再遇到断点，例如，在选择某些菜单项以读取一个文件或者以其他方式改变显示的内容之后，再遇到断点。这听起来就是我们需要的结果。我们或者根本没有在 `OnPaint()` 中设置断点，或者应用程序不会显示它在最初的启动窗口中显示的边界点之外的其他内容。

有一种方式可以解决这个问题。

如果有足够大的屏幕，最简单的方式就是平铺开发环境窗口，而不是把它设置为最大化，使之远离应用程序窗口，这样首先应用程序就不会被挡住了。但在大多数情况下，这并不是一个有效的解决方案，因为这样会使开发环境窗口过小也可以使用第二个监视器。另一个解决方案使用相同的规则，即在调试时把应用程序声明为最上面的应用程序。方法是在 `Form` 类中设置属性 `TopMost`，这很容易在 `InitializeComponent()` 方法中完成：

```
private void InitializeComponent()
{
    this.TopMost = true;
```

也可以在 Visual Studio 2010 的属性窗口中设置这个属性。

窗口设置为 `TopMost` 表示应用程序不会被其他窗口挡住(除了其他放在最上面的窗口)。它仍然总是放在其他窗口的上面，甚至在另一个应用程序得到焦点时，也是这样。这是任务管理器的执行方式。

即使利用这个技巧也必须小心，因为我们不能确定 Windows 何时会决定因为某种原因触发 `Paint` 事件。如果在某些特殊的情况下，在 `OnPaint()` 方法出了问题(例如，应用程序在选择某个特定菜单项后绘图，但此时出了问题)，那么最好的方式是在 `OnPaint()` 方法中添加一些虚拟代码，该方法测试某些条件，这些条件只在特殊的情况下才为 `true`。然后在 `if` 块中设置断点，如下所示：

```
protected override void OnPaint( PaintEventArgs e )
{
    // Condition() evaluates to true when we want to break
    if (Condition())
    {
```

```
int ii = 0;    // <-SET BREAKPOINT HERE!!!
}
```

这是设置条件断点的一种简捷方式。

48.4 绘制可滚动的窗口

前面的 `DrawShapes` 示例运行良好，因为需要绘制的内容正好适合最初的窗口大小。本节介绍如果绘制的内容不适合窗口的大小，需要做哪些工作。

对于本示例，下面扩展 `DrawShapes` 示例，以解释滚动的概念。为了使该示例更符合实际，首先创建一个 `BigShapes` 示例，其中将矩形和椭圆画大一些。此时将使用 `Point`、`Size` 和 `Rectangle` 结构定义绘图区域，说明如何使用它们。进行了这样的修改后，`Form1` 类的相关部分如下所示：

```
// member fields
private readonly Point rectangleTopLeft = new Point(0, 0);
private readonly Size rectangleSize = new Size(200,200);
private readonly Point ellipseTopLeft = new Point(50, 200);
private readonly Size ellipseSize = new Size(200, 150);
private readonly Pen bluePen = new Pen(Color.Blue, 3);
private readonly Pen redPen = new Pen(Color.Red, 2);
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    if (e.ClipRectangle.Top < 350 || e.ClipRectangle.Left < 250)
    {
        Rectangle rectangleArea =
            new Rectangle (rectangleTopLeft, rectangleSize);
        Rectangle ellipseArea =
            new Rectangle (ellipseTopLeft, ellipseSize);
        dc.DrawRectangle(bluePen, rectangleArea);
        dc.DrawEllipse(redPen, ellipseArea);
    }
}
```

注意，这里还把 `Pen`、`Size` 和 `Point` 对象变成成员字段——这比每次需要绘图时都新建一个 `Pen` 的效率更高。

运行这个示例，得到如图 48-6 所示的结果。

很快这里有一个问题，图形在 300×300 像素的绘图区域中放不下。

一般情况下，如果文档太大，不能完全显示，应用程序就会添加滚动条，以便用户滚动窗口，查看其中选中的部分。这是另一个区域，在该区域中如果使用标准控件构建 `Windows` 窗体，就让 .NET 运行库和基类来处理所有操作。如果在窗体中有各种控件，那么 `Form` 实例一般知道这些控件在哪里，并且如果其窗口比较小，`Form` 实例就知道需要添加滚动条。`Form` 实例还会自动添加滚动条，不仅如此，它还可以正确地绘制用户滚动到的部分屏幕。此时，用户不需要在代码中做什么工作。但在本章中，我们要在屏幕上绘制图形，所以要帮助 `Form` 实例确定何时能滚动。

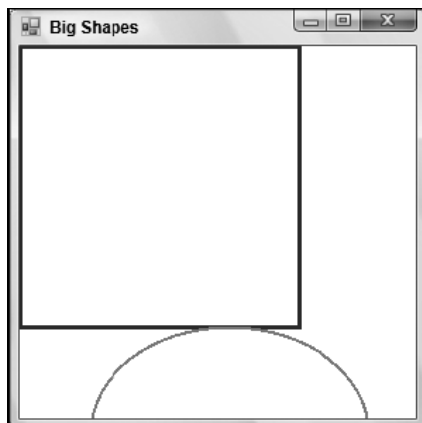


图 48-6

添加滚动条实际上很简单。Form 仍会处理所有的操作——因为它不知道绘图区域有多大。在上面的 BigShapes 示例中没有滚动条的原因是，Windows 不知道它们需要滚动条。我们需要确定的是，矩形的大小从文档的左上角(或者是在进行任何滚动前的客户区域的左上角)开始向右下角延伸，其大小应足以包含整个文档。本章把这个区域称为文档区域，如图 48-7 所示，本例的文档区域应是 250×350 个像素。

使用相关的属性 Form.AutoScrollMinSize 即可轻松确定文档的大小。因此给 InitializeComponent() 方法或 Form1 的构造函数添加下述代码：

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
    this.AutoScrollMinSize = new Size(250, 350);
}
```

另外，AutoScrollMinSize 属性还可以用 Visual Studio 2010 的属性窗口设置。注意要访问 Size 类，需要添加下面的 using 语句：

```
using System.Drawing;
```

在应用程序启动时设置最小尺寸，并保持不变，在这个特定的应用程序中是必要的，因为我们知道屏幕区域一般有多大。在运行该应用程序时，这个“文档”不会改变大小。但要记住，如果应用程序执行显示文件的内容这样的操作，或者执行改变屏幕的哪个区域的操作，就需要在其他时间设置这个属性(此时，必须手动调整代码，Visual Studio 2010 的属性窗口只能在构建窗体时设置属性的初始值)。

设置 MinScrollSize 属性只是一个开始，仅有它是不够的。图 48-8 显示了示例应用程序目前的外观——开始时，屏幕会正确地显示图形。

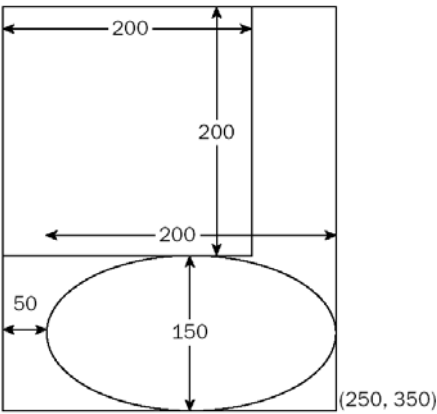


图 48-7

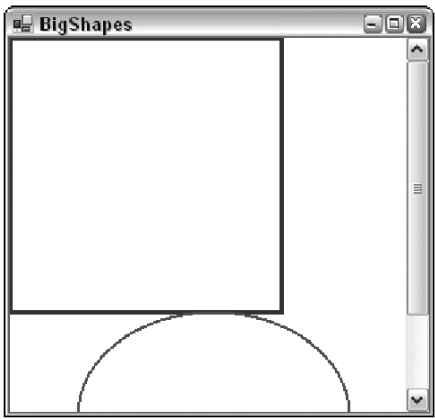


图 48-8

注意，不仅窗体正确地设置了滚动条，而且它的大小也正确设置了，以指定文档当前显示的比例。可以试着在运行示例时重新设置窗口的大小，这样就会发现滚动条会正确地响应，甚至如果使窗口变得足够大，不再需要滚动条时，滚动条就会消失。

但是，如果使用一个滚动条，并向下滚动它，会发生什么情况呢？结果如图 48-9 所示。显然，出现了错误！

出错的原因是我们没有在 `OnPaint()` 重写方法的代码中考虑滚动条的位置。如果最小化窗口，再还原它，从而重新绘制窗口自身，就可以很清楚地看出这一点。结果如图 48-10 所示。

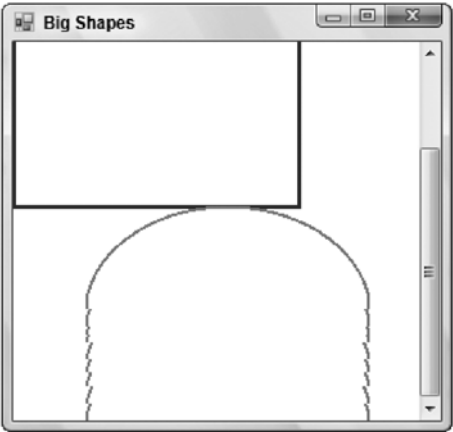


图 48-9

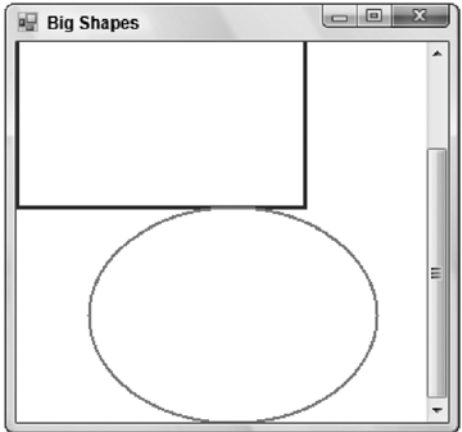


图 48-10

图形像以前一样进行了绘制，矩形的左上角嵌套在客户区域的左上角，就好像根本没有移动过滚动条一样。

在介绍如何更正这个问题前，先介绍一下在这些屏幕截图上发生了什么。

首先从 `BigShapes` 示例开始，如图 48-8 所示。在这个例子中，整个窗口刚刚重新进行了绘制。看看前面的代码，该代码的作用是使图形实例用左上角坐标(0,0)(相对于窗口的客户区域的左上角)绘制一个矩形——它是已经绘制过的。问题是，图形实例在默认情况下把坐标解释为是相对于客户

窗口的，它不能识别滚动条。代码还没有尝试为滚动条的位置调整坐标。椭圆也是这样。

下面处理图 48-9 中的屏幕截图。在向下滚动后，注意窗口上半部分显示正确，这是因为它是在应用程序第一次启动时绘制的。在滚动窗口时，Windows 没有要求应用程序重新绘制已经显示在屏幕中的内容。Windows 足够智能可以判断屏幕上目前显示的哪些内容可以平滑地移动，以匹配滚动条所处的位置。这是一个非常高效的过程，因为它也能使用某些硬件来加速完成。在这个屏幕截图中，有错误的是窗口下部的 1/3 部分。在应用程序第一次显示时，没有绘制这部分窗口，因为在滚动窗口前，这部分在客户区域的外部。这表示 Windows 要求 BigShapes 应用程序绘制这个区域。它触发 Paint 事件，把这个区域作为剪切的矩形。这也是 OnPaint()重载方法完成的任务。

该问题的另一种表达方式是我们将坐标表示为相对于文档开头的左上角——需要转换它们，用相对于客户区域的左上角的坐标表示它们。图 48-11 说明了这一点。

为了使该图更清晰，我们向下向右扩展了该文档，超出了屏幕的边界，但这不会改变我们的推理。我们还假定其上还有一个水平滚动条和一个垂直滚动条。

在图 48-11 中，细线条的矩形标记了屏幕区域的边框和整个文档的边框。粗线条标记了要绘制的矩形和椭圆。P 标记要绘制的某个任意点，这个点在后面会作为一个示例。在调用绘图方法时，提供图形实例和从 B 点到 P 点的矢量，这个矢量表示为一个 Point 实例。我们实际上需要给出从 A 点到 P 点的矢量。

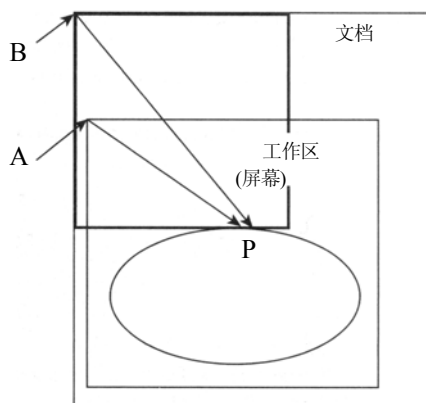


图 48-11

问题是，我们不知道从 A 点到 P 点的矢量。而知道从 B 点到 P 点的矢量，这就是 P 点相对于文档左上角的坐标——我们要在文档的 P 点绘图的位置。还知道从 B 点到 A 点的矢量就是滚动的距离，它存储在 Form 类的一个属性 AutoScrollPosition 中。但是不知道从 A 点到 P 点的矢量。

要解决这个问题，只需进行矢量相减即可。例如，要从 B 点到 P 点，可以水平向右移动 150 个像素，再垂直向下移动 200 个像素。而要从 B 点到 A 点，就需要水平向右移动 10 个像素，再垂直向下移动 57 个像素。这表示，要从 A 点到 P 点，需要水平向右移动 140 个像素(=150-10)，再垂直向下移动 143 个像素(=200-57)。为了使之更简单，Graphics 类实际上实现了一个方法来进行这些计算，这个方法是 TranslateTransform()，我们给它传递水平坐标和垂直坐标，表示工作区的左上角相对于文档的左上角(AutoScrollPosition 属性，它是图 48-11 中从 B 点到 A 点的矢量)。然后 Graphics 设备考虑客户区域相对于文档区域的位置，计算出它所有的坐标。

如果把上述解释转化为代码，通常所需做的就是下面这行代码添加到绘图代码中：

```
dc.TranslateTransform(this.AutoScrollPosition.X, this.AutoScrollPosition.Y);
```

但在本示例中，它有点复杂，因为我们还要查看剪切区域，看看是否需要进行绘制工作。这个测试需要调整，把滚动的位置也考虑在内。完成后，该示例的整个绘图代码如下所示：

```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Size scrollOffset = new Size(this.AutoScrollPosition);
    if (e.ClipRectangle.Top+scrollOffset.Width < 350 ||
        e.ClipRectangle.Left+scrollOffset.Height < 250)
    {
        Rectangle rectangleArea = new Rectangle
            (rectangleTopLeft+scrollOffset, rectangleSize);
        Rectangle ellipseArea = new Rectangle
            (ellipseTopLeft+scrollOffset, ellipseSize);
        dc.DrawRectangle(bluePen, rectangleArea);
        dc.DrawEllipse(redPen, ellipseArea);
    }
}
```

现在，滚动代码工作正常，最后得到正确地滚动的屏幕截图，如图 48-12 所示。

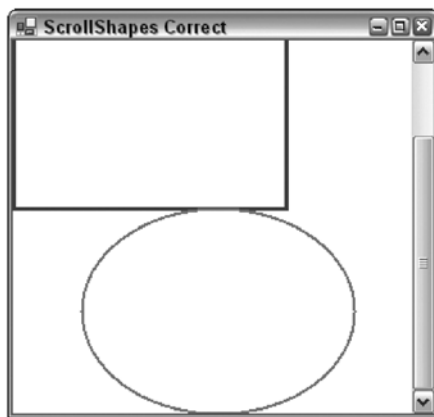


图 48-12

48.5 世界、页面和设备坐标

测量相对于文档左上角的位置和测量相对于屏幕(桌面)左上角的位置之间的区别如此重要，因此 GDI+ 对于这些坐标系统有专门的名称：

- **世界坐标(World Coordinate)**——要测量的点距离文档左上角的位置(以像素为单位)。
- **页面坐标(Page Coordinate)**——要测量的点距离客户区域左上角的位置(以像素为单位)。



熟悉 GDI 的开发人员要注意，世界坐标对应于 GDI 中的逻辑坐标。页面坐标对应于设备坐标。还要注意，编写逻辑坐标和设备坐标之间的转换代码在 GDI+中有了变化。在 GDI 中，使用 Windows API 函数 LPtoDP()和 DPtoLP()通过设备上下文进行转换，而在 GDI+中，正是 Control 类来维护转换过程中所需要的信息，Form 和各种 Windows 窗体控件设备派生自 Control 类。

GDI+还有第 3 种坐标，即设备坐标(Device Coordinate)。设备坐标类似于页面坐标，但其测量单位不是像素，而使用用户调用 Graphics.PageUnit 属性指定的某些其他单位。它可以使用的单位除了默认的像素外，还包括英寸和毫米。本章虽然没有使用 PageUnit 属性，但它可用作获取设备的不同像素密度的方式。例如，在大多数显示器上，100 个像素大约是 1 英寸。但是，激光打印机可以达到 1 200 dpi(点/英寸)——这表示 100 个像素宽的图形在该激光打印机上打印时会比较小。把单位设置为英寸，指定图形为 1 英寸宽，就可以确保图形在不同的设备上有相同的大小。这可通过如下代码来说明：

```
Graphics dc = this.CreateGraphics();
dc.PageUnit = GraphicsUnit.Inch;
```

通过 GraphicsUnit 枚举可用的值如下：

- **Display**——指定显示器的测量单位
- **Document**——把文档单位(1/300 英寸)定义为测量单位
- **Inch**——把英寸定义为测量单位
- **Millimeter**——把毫米定义为测量单位
- **Pixel**——把像素定义为测量单位
- **Point**——把打印机的点数(1/72 英寸)定义为测量单位
- **World**——把世界坐标系定义为测量单位

48.6 颜色

本节介绍如何在绘制图形时指定使用的颜色。

在 GDI+中，颜色用 System.Drawing.Color 结构的实例来表示。一般情况下，初始化这个结构后，就不能使用对应的 Color 实例对该结构进行操作了——只能把它传递给其他需要 Color 的任何其他主调方法。前面我们遇到过这种结构，在前面的每个示例中都设置了窗口的客户区域的背景色，还设置了要显示的各种图形的颜色。Form.BackColor 属性返回一个 Color 实例。本节将详细介绍这个结构，特别是要介绍构建 Color 的几种不同方式。

48.6.1 RGB 值

显示器可以显示的颜色总数非常大——超过 1600 万。其确切的数字是 2 的 24 次方，即 16 777 216。显然，我们需要对这些颜色进行索引，才能指定在任何给定的像素上要显示什么颜色。

对颜色进行索引的最常见方式是把它们分为红、绿、蓝分量，这种思想基于下述原理：人眼可

以分辨的任何颜色都由一定量的红色光、绿色光和蓝色光组成。这些光称为分量(component)。实际上,如果每种分量的光分为 256 种不同的强度,它们提供了足够平滑的过渡,可以把人眼能观察到的图像显示为高质量的照片。因此,指定颜色时,可以给出这些分量的量,其值在 0~255 之间,其中 0 表示没有这种分量,255 表示这种分量的光达到最大的强度。

这给出了向 GDI+说明颜色的第一种方式。可以调用静态函数 `Color.FromArgb()`指定该颜色的红、绿、蓝值。Microsoft 没有为此提供构造函数,原因是除了一般的 RGB 分量外,还有其他方式来表示颜色。因此,Microsoft 认为给定义的任何构造函数传递参数会引起误解:

```
Color redColor = Color.FromArgb(255,0,0);
Color funnyOrangyBrownColor = Color.FromArgb(255,155,100);
Color blackColor = Color.FromArgb(0,0,0);
Color whiteColor = Color.FromArgb(255,255,255);
```

3 个参数分别是红、绿、蓝值。这个函数有许多重载方法,其中一些也允许指定 Alpha 混合值(这是方法名 `FromArgb()`中的 A)。Alpha 混合超出了本章的范围,但把它与屏幕上已有的任何颜色混合起来,可以绘出半透明的颜色。这可以得到一些很漂亮的效果,常常用于游戏中。

48.6.2 命名颜色

使用 `FromArgb()`方法构造颜色是一种非常灵活的技巧,因为它表示我们可以指定人眼能辨识出的任何颜色。但是,如果要得到一种简单、标准、众所周知的纯色,如红色或蓝色,命名想要的颜色就比较简单。因此 Microsoft 还在 `Color` 中提供了许多静态属性,每个属性都返回一种命名颜色。在下面的示例中,把窗口的背景色设置为白色时,就使用了其中一个属性:

```
this.BackColor = Color.White;
// has the same effect as:
// this.BackColor = Color.FromArgb(255, 255, 255);
```

有几百种这样的颜色。完整的列表参见 SDK 文档。它们包括所有的纯色:红、白、蓝、绿和黑等,还包括 `MediumAquamarine`、`LightCoral` 和 `DarkOrchid` 等颜色。还有一个 `KnownColor` 枚举,它列出了命名颜色。



每种命名颜色都表示一组精确的 RGB 值,它们最初是多年前选择出来用在 Internet 上的。这种思想提供了色谱上一组有用的颜色,Web 浏览器可以辨识出这些颜色的名称——因此不必在 HTML 代码中显式写出它们的 RGB 值。几年前,这些颜色仍很重要,因为早期的浏览器不必准确地显示非常多的颜色,命名的颜色应该提供了一组在大多数浏览器中准确地显示出来的颜色。目前它们已经不那么重要了,因为现代的 Web 浏览器能准确地显示任何 RGB 值。还有一些 Web 安全的调色板可以为开发人员提供可用于大多数浏览器的颜色的完整列表。

48.6.3 图形显示模式和安全的调色板

尽管原则上显示器可以显示超过 1600 万种 RGB 颜色,但实际上这取决于如何在计算机上设置

显示属性。在 Windows 中，传统上有 3 个主要的颜色选项(尽管有些计算机还依据硬件提供其他选项)：真彩色(24 位)、增强色(16 位)和 256 色(在目前的一些图形卡上，真彩色实际上标记为 32 位，这必须对硬件进行优化，尽管此时 32 位中只有 24 位用于该颜色)。

只有真彩色模式允许同时显示所有的 RGB 颜色。这听起来是最佳选择，但它是有代价的：完整的 RGB 值需要用 3 个字节来保存，这表示要显示的每个像素都需要用图形卡内存中的 3 个字节来保存。如果图形卡内存费用较高(这种限制现在已经不像以前那样普遍了)，就可以选择一种其他模式。增强色模式用两个字节表示一个像素。每个 RGB 分量用 5 位就足够了。所以红色只有 32 种不同的强度，而不是 256 种。蓝色和绿色也是这样，总共有 65 536 种颜色。这对于需要偶尔查看照片质量的图像足够了，但比较微妙的阴影区域会被破坏。

256 色模式给出的颜色更少。但是在这种模式下，可以选择任何颜色，系统会建立一个调色板，这是一个从 1600 万种 RGB 颜色中选择出来的 256 种颜色的一个列表。在调色板中指定了颜色后，图形设备就能够只显示所指定的这些颜色。调色板在任何时候都可以改变——但图形设备每次只能在屏幕上显示 256 种不同的颜色。只有当获得高性能和视频内存费用较高时，才使用 256 色模式。大多数计算机游戏都使用这种模式——它们仍能得到相当好的图形，因为调色板经过了非常仔细的选择。

一般情况下，如果显示设备使用增强色或 256 色模式，并要显示某种 RGB 颜色，它就会从能显示的颜色池中选择一种在数学上最接近的匹配颜色。因此知道颜色模式非常重要。如果要绘制某些涉及微妙阴影或照片质量的图像，而用户没有选择 24 位颜色模式，就看不到期望的效果。如果要使用 GDI+进行绘制，就应该用不同的颜色模式测试应用程序(应用程序也可以通过编程设置给定的颜色模式，尽管本章不讨论这个问题)。

48.6.4 安全调色板

下面简要介绍安全调色板，这是一种非常常见的默认调色板。它工作的方式是为每种颜色分量设置 6 个间隔相等的值，这些值分别是 0、51、102、153、204、255。换言之，红色分量可以是这些值中的任一个。绿色分量和蓝色分量也一样。所以安全调色板中的颜色就包括(0,0,0) (黑色)、(153,0,0) (暗红色)、(0, 255, 102) (蓝绿色)等，这样就得到了 $6^3=216$ 种颜色。这是一种让调色板包含色谱中间隔相等的颜色和所有亮度的简单方式，但实际上这是不可行的，因为数学上等间隔的颜色分量并不表示这些颜色的区别在人眼看来也是相等的。

如果把 Windows 设置为 256 色模式，默认的调色板就是安全调色板，其中添加了 20 种标准的 Windows 颜色和 20 种备用颜色。

48.7 画笔和钢笔

本节介绍两个辅助类，在绘制图形时需要使用它们。前面已经用到过 Pen 类，它用于告诉 graphics 实例如何绘制线条。相关的类是 System.Drawing.Brush，它告诉图形实例如何填充线条。例如，Pen 用于绘制前面示例中矩形和椭圆的边框。如果需要把这些图形绘制为实心的，就要使用画笔指定如何填充它们。这两个类有一个共同点：很难对它们调用任何方法。用需要的颜色和其他属性构造一个 Pen 或 Brush 实例，再把它传递给需要 Pen 或 Brush 的绘图方法即可。



如果读者以前使用 GDI 编程,就可能会注意到在前两个示例中,在 GDI+中使用 Pen 的方式是不同的。在 GDI 中,一般是调用一个 Windows API 函数 SelectObject(),它实际上把钢笔关联到设备上下文上。这支钢笔用于所有需要钢笔的绘图操作中,直到再次调用 SelectObject()方法通知设备上下文停止使用它时为止。这条规则也适用于画笔和其他对象,如字体和位图。而使用 GDI+, Microsoft 选择一种无状态的模式,其中没有默认的钢笔或其他辅助对象。只需给特定方法调用指定合适的辅助对象即可。

48.7.1 画笔

GDI+有几种不同类型的画笔,本章不准备详细介绍它们,这里仅解释几种比较简单的画笔,读者掌握其要领即可。每种画笔都由一个派生自抽象类 System.Drawing.Brush 的类的实例来表示。最简单的画笔 System.Drawing.SolidBrush 仅指定了区域用纯色来填充:

```
Brush solidBeigeBrush = new SolidBrush(Color.Beige);
Brush solidFunnyOrangyBrownBrush = new SolidBrush(Color.FromArgb(255,155,100));
```

另外,如果画笔是一种 Web 安全色,就可以用另一个类 System.Drawing.Brushes 构造画笔。Brushes 是永远不能真正地实例化的一个类(它有一个私有构造函数,禁止实例化它)。它只有许多静态属性,每个属性都返回指定颜色的画笔。可以像下面这样使用画笔:

```
Brush solidAzureBrush = Brushes.Azure;
Brush solidChocolateBrush = Brushes.Chocolate;
```

比较复杂的一种画笔是影线画笔(hatch brush),它通过绘制一种图案来填充区域。因为这种类型的画笔比较高级,所以在 Drawing2D 名称空间中,用 System.Drawing.Drawing2D.HatchBrush 类表示。Brushes 类不能帮助我们使用影线画笔,而需通过提供一个影线型式和两种颜色(前景色和背景色,背景色可以忽略,此时将使用默认的黑色),来显式构造一支影线画笔。影线样式可以取自于枚举 System.Drawing.Drawing2D.HatchStyle, 其中有许多 HatchStyle 值,其完整列表参阅 SDK 文档。大体上,一般的样式包括 ForwardDiagonal、Cross、DiagonalCross、SmallConfetti 和 ZigZag。构造影线画笔的示例如下所示。

```
Brush crossBrush = new HatchBrush(HatchStyle.Cross, Color.Azure);
// background color of CrossBrush is black
Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
                                   Color.DarkGoldenrod, Color.Cyan);
```

GDI 只能使用实线画笔和影线画笔, GDI+添加了两种新画笔:

- System.Drawing.Drawing2D.LinearGradientBrush 用一种在屏幕上变化的颜色填充区域。
- System.Drawing.Drawing2D.PathGradientBrush 与上述画笔类似,但其颜色沿着要填充的区域的路径变化。

如果细心使用,这些画笔就可以显现一些惊人的效果。

48.7.2 钢笔

与画笔不同，钢笔只用一个类 `System.Drawing.Pen` 来表示。但钢笔比画笔复杂一些，因为它需要指定线条应有多宽(像素)，对于比较宽的线条，还要确定如何填充该线条中的区域。钢笔还可以指定其他许多属性，本章不讨论它们，但其中包括前面提到的 `Alignment` 属性，该属性表示相对于图形的边框，线条该如何绘制，以及在线条的末尾绘制什么图形(是否使图形光滑过渡)。

粗线条中的区域可以用纯色填充，或者使用画笔来填充。因此 `Pen` 实例可以包含 `Brush` 实例的引用。这非常强大，因为这表示我们可以绘制用影线阴影或线性阴影着色的线条。构造自己设计的 `Pen` 实例有 4 种不同的方式：可以传递一种颜色，或者传递一支画笔。这两个构造函数都会生成一个像素宽的钢笔。另外，还可以传递一种颜色或一支画笔，以及一个表示钢笔宽度的 `float` 类型的值。(该宽度必须是一个 `float` 类型的值，以允许执行绘图操作的 `Graphics` 对象使用非默认的单位，如毫米或英寸，例如可以指定宽度是小数形式的英寸)。例如，可以构造类似如下的钢笔：

```
Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
                                   Color.DarkGoldenrod, Color.Cyan);
Pen solidBluePen = new Pen(Color.FromArgb(0,0,255));
Pen solidWideBluePen = new Pen(Color.Blue, 4);
Pen brickPen = new Pen(brickBrush);
Pen brickWidePen = new Pen(brickBrush, 10);
```

另外，为了快速构造钢笔，还可以使用 `System.Drawing.Pens` 类，它与 `Brushes` 类一样，包含许多备用钢笔。这些钢笔的宽度都是一个像素，使用通常的 Web 安全色，这样就可以用下述方式构建钢笔：

```
Pen solidYellowPen = Pens.Yellow;
```

48.8 绘制图形和线条

前面介绍了在屏幕上绘制规定的图形所需要的所有基类和对象。下面复习 `Graphics` 类可以使用的一些绘图方法，并用一个小示例来介绍几种画笔和钢笔的用法。

`System.Drawing.Graphics` 类有很多方法，利用这些方法可以绘制各种线条、空心图形和实心图形。表 48-4 所示的列表并不完整，但给出了主要的方法，您应能据此掌握绘制各种图形的要领。

表 48-4

方 法	常 见 参 数	绘制的图形
<code>DrawLine()</code>	钢笔、起点和终点	一条直线
<code>DrawRectangle()</code>	钢笔、位置和大小	空心矩形
<code>DrawEllipse()</code>	钢笔、位置和大小	空心椭圆
<code>FillRectangle()</code>	画笔、位置和大小	实心矩形
<code>FillEllipse()</code>	画笔、位置和大小	实心椭圆
<code>DrawLines()</code>	钢笔、点数组	一组线条，把数组中的每个点按顺序连接起来
<code>DrawBezier()</code>	钢笔、4 个点	经过两个端点的一条光滑曲线，剩余的两个点用于控制曲线的形状

(续表)

方 法	常 见 参 数	绘制的图形
DrawCurve()	钢笔、点数组	经过这些点的一条光滑曲线
DrawArc()	钢笔、矩形、两个角	由角度定义的矩形中圆的一部分
DrawClosedCurve()	钢笔、点数组	与 DrawCurve 一样，但还要绘制一条用于闭合曲线的直线
DrawPie()	钢笔、矩形、两个角	矩形中的空心楔形
FillPie()	画笔、矩形、两个角	矩形中的实心楔形
DrawPolygon()	钢笔、点数组	与 DrawLines 一样，但还要连接第一点和最后一点，以闭合绘制的图形

在结束绘制简单对象的主题前，本节用一个简单示例来说明使用画笔可以得到的各种可视效果。该示例是 ScrollMoreShapes，它基本上是 ScrollShapes 的修正版本。除了矩形和椭圆外，我们还添加了一条粗线条，用各种自定义画笔填充图形。前面解释了绘图规则，所以这里只给出代码，而不进行过多的注释。首先，因为添加了新画笔，所以需指定使用 System.Drawing.Drawing2D 名称空间：

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Text;
using System.Windows.Forms;
```

接着是 Form1 类中的一些额外字段，其中包含了要绘制图形详细的位置信息，以及要使用的各种钢笔和画笔：



可从
wrox.com
下载源代码

```
private Rectangle rectangleBounds = new Rectangle(new Point(0,0),
                                                    new Size(200,200));
private Rectangle ellipseBounds = new Rectangle(new Point(50,200),
                                                  new Size(200,150));
private readonly Pen bluePen = new Pen(Color.Blue, 3);
private readonly Pen redPen = new Pen(Color.Red, 2);
private readonly Brush solidAzureBrush = Brushes.Azure;
private readonly Brush solidYellowBrush = new SolidBrush(Color.Yellow);
private static readonly Brush brickBrush = new
    HatchBrush(HatchStyle.DiagonalBrick, Color.DarkGoldenrod, Color.Cyan);
private readonly Pen brickWidePen = new Pen(brickBrush, 10);
```

代码段 ScrollMoreShapes.sln

把 BrickBrush 字段声明为静态，就可以使用该字段的值初始化 brickWidePen 字段。C#不允许使用一个实例字段初始化另一个实例字段，因为还没有定义要先初始化哪个实例字段。然而，如果把字段声明为静态字段就可以解决这个问题，因为只实例化了 Form1 类的一个实例，字段是静态字段还是实例字段就不重要了。

下面是 OnPaint()方法的重写版本：



```
protected override void OnPaint( PaintEventArgs e )
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    Point scrollOffset = AutoScrollPosition;
    dc.TranslateTransform(scrollOffset.X, scrollOffset.Y);
    if (e.ClipRectangle.Top+scrollOffset.X < 350 ||
        e.ClipRectangle.Left+scrollOffset.Y < 250)
    {
        dc.DrawRectangle(bluePen, rectangleBounds);
        dc.FillRectangle(solidYellowBrush, rectangleBounds);
        dc.DrawEllipse(redPen, ellipseBounds);
        dc.FillEllipse(solidAzureBrush, ellipseBounds);
        dc.DrawLine(brickWidePen, rectangleBounds.Location,
            ellipseBounds.Location+ellipseBounds.Size);
    }
}
```

代码段 ScrollMoreShapes.sln

与以前一样，也将 AutoScrollMinSize 设置为(250,350)，结果如图 48-13 所示。

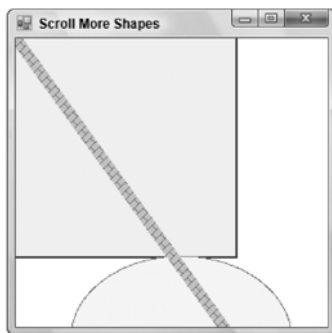


图 48-13

注意，粗对角线已在矩形和椭圆上绘制出来，它需要最后一个绘制。

48.9 显示图像

使用 GDI+ 最常见的操作是希望显示文件中已有的图像。这确实要比绘制自己的用户界面简单多了，因为图像已经预先绘制好了。实际上，我们只需要加载文件，让 GDI+ 显示它即可。图像可以只包含一个绘制的简单线条或一个图标，也可以比较复杂，如一张照片。对图像也可以执行某些操作，如拉伸或旋转图像，也可以选择只显示图像的一部分。

略有变化，本节将先给出一个示例，再讨论显示图像时需要注意的一些问题。可以这么做的原因是显示图像的代码非常简单。

我们需要 .NET 的一个基类 System.Drawing.Image。Image 类的一个实例表示一幅图像。读取一幅图像仅需使用一行代码：

```
Image myImage = Image.FromFile("FileName");
```

FromFile()方法是 Image 类的一个静态成员,是实例化图像的常用方式。文件可以是任何支持的图形文件格式,包括.bmp、.jpg、.gif 和.png。

显示图像也很简单,假定手头有一个合适的 Graphics 实例,那么调用 Graphics.DrawImage Unscaled()方法或 Graphics.DrawImage()方法就足够了。这些方法都有许多重载方法,可以根据图像的位置和要绘制的大小非常灵活地处理用户提供的信息。然后本示例使用 DrawImage()方法:

```
dc.DrawImage(myImage, points);
```

在这行代码中,假定 dc 是一个 Graphics 实例,MyImage 是要显示的图像,points 是一个 Point 结构数组,其中 points[0]、points[1]和 points[2]是图像左上角、右上角和左下角的坐标。



熟悉 GDI 的开发人员可以从图像中看出 GDI 与 GDI+的最大区别。在 GDI 中,显示图像涉及几个重要的步骤。如果图像是一个位图,加载它就很简单,但如果它是任何其他类型的文件,加载它就会涉及 OLE 对象的一系列调用。实际上,把加载的图像显示到屏幕上要获得它的一个句柄,把它放在一个内存设备上下文中,然后在设备环境之间执行一个块传输。尽管设备上下文和句柄仍在后台上,但如果要开始从代码中对图像进行复杂的编辑,就需要它们。简单的任务完美地封装在 GDI+对象模型上。

下面用一个示例 DisplayImage 来说明显示图像的过程。这个示例在应用程序的主窗口中显示一个.jpg 文件。要使操作过程简单一些,.jpg 文件的路径硬编码到应用程序中(如果运行该示例,就需要改变该路径,以反映在系统中文件的位置)。要显示的.jpg 文件是圣彼得堡的日落图片。

与其他例子一样,DisplayImage 项目是 C# Visual Studio 2010 生成的一个标准的 Windows 应用程序。在 Form1 类中添加下述字段:

```
readonly Image piccy;  
private readonly Point [] piccyBounds;
```

然后在 Form1()构造函数中加载文件:



可从
wrox.com
下载源代码

```
public Form1()  
{  
    InitializeComponent();  
    piccy =  
        Image.FromFile(@"C:\ProCSharp\GdiPlus\Images\London.jpg");  
    AutoScrollMinSize = piccy.Size;  
    piccyBounds = new Point[3];  
    piccyBounds[0] = new Point(0,0); // top left  
    piccyBounds[1] = new Point(piccy.Width,0); // top right  
    piccyBounds[2] = new Point(0,piccy.Height); // bottom left  
}
```

代码段 DisplayPicture.sln

注意,图像的大小(以像素为单位)通过其 Size 属性来获得,我们使用该属性设置文档区域。再建立一个 piccyBounds 数组,它用于标识图像在屏幕上的位置。选择 3 个角的坐标,按实际大小和图形来绘制图像,但如果要重新设置图像的大小、拉伸图像,或甚至把图像变形为非矩形的平行四

边形，则可以仅改变 `piccyBounds` 数组中 `Point` 的值来实现上述操作。

通过 `OnPaint()` 重写方法显示该图像：



可从
wrox.com
下载源代码

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    dc.ScaleTransform(1.0f, 1.0f);
    dc.TranslateTransform(AutoScrollPosition.X, AutoScrollPosition.Y);
    dc.DrawImage(piccy, piccyBounds);
}
```

代码段 DisplayPicture.sln

最后，特别注意对 IDE 生成的 `Form1.Dispose()` 方法代码进行修改：



可从
wrox.com
下载源代码

```
protected override void Dispose(bool disposing)
{
    piccy.Dispose();
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}
```

代码段 DisplayPicture.sln

只要不再需要该图像了，就应立即删除它，这一点很重要因为图像在使用时一般会占用许多内存。在调用 `Image.Dispose()` 方法后，因为 `Image` 实例不再引用任何实际图像，所以不能再显示该图像(除非加载了一幅新图像)。

运行代码，会得到如图 48-14 所示的结果。

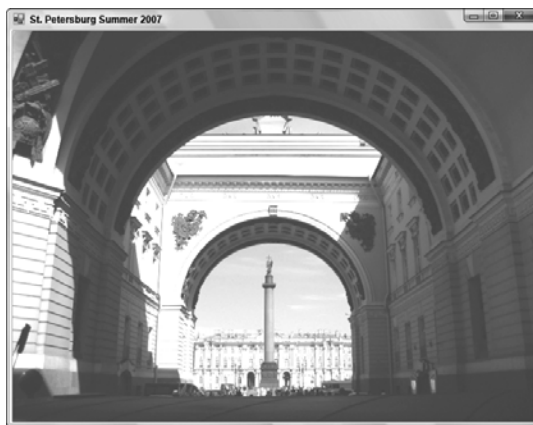


图 48-14

48.10 处理图像时的问题

尽管显示图像很简单，但需要理解一些在后台执行的什么操作。

理解图像最重要的一点是，图像的形状总是矩形。这不只是出于方便原因，其原因是底层的技术。所有现代图形卡都内置了硬件，从而可以非常高效地从内存的一个地方把像素块复制到内存的另一个地方。假定像素块表示一个矩形区域，这个硬件加速操作可以虚拟为一个操作，而且执行速度非常快。实际上，这是现代高性能图形的关键。这个操作称为位图块传输(或者 BitBlt)。Graphics.DrawImageUnscaled()方法在内部使用 BitBlt，这就是为什么能够看见一幅大图像的原因，该图像也许包含上百万个像素，但几乎是立即就显示出来。如果计算机必须把图像逐个像素地复制到屏幕上，则该图像最多在几秒钟内逐渐画出来。

因为 BitBlt 的效率非常高，所以图像的所有绘制和处理操作都使用 BitBlt 完成。甚至图像的某些编辑操作也使用表示内存区域的设备环境之间部分图像的 BitBlt 完成。在 GDI 时代中，Windows 32 API 函数 BitBlt()是最重要、使用最广泛的图像处理函数，而在 GDI+中，BitBlt 操作一般隐藏在 GDI+对象模型中。

尽管很容易模拟类似的效果，但图像的 BitBlt 区域不可能是矩形。一种方式是为了 BitBlt，把某种颜色标记为透明的。这样源图像中该颜色的区域不会覆盖目标设备中对应像素的已有颜色。在 BitBlt 过程中还可以指定，结果图像的每个像素会在进行 BitBlt 前，对源图像上和目标设备上该像素的颜色进行某些逻辑操作来形成(如按位 AND)。这样的操作由硬件加速来支持，用于产生各种微妙的效果。注意 Graphics 对象实现另一个方法 DrawImage()，该方法类似于 DrawImageUnscaled()方法，但它有许多重载方法，可以指定 BitBlt 更复杂的形式，以便在绘图过程中使用。DrawImage()方法还可以只绘制(使用 BitBlt)图像的某个特定部分，或者对它执行其他特定操作，如在绘图时缩放(缩放其大小)它。

48.11 绘制文本

到目前为止，本章还有一个非常重要的问题要讨论——显示文本。因为在屏幕上绘制文本通常比绘制简单图形更复杂。在不考虑外观的情况下，只显示一两行文本非常简单——它只需调用 Graphics 实例的一个方法 Graphics.DrawString()。但如果要显示一个文档，其中有许多文本，则事情很快变得复杂多了，这有两个原因：

- 如果只考虑外观，则需要理解字体。图形的绘制需要使用画笔和钢笔作为帮助对象，绘制文本的过程则需要把字体作为帮助对象。而且，理解字体并不容易。
- 在窗口中需要仔细布局文本。用户通常期望文字一个跟一个地排列——排成一行，其间有一定的间隔。这是一个比想象中更困难的任务。对于初学者，一般事先不知道屏幕上文字之间的间隔有多大。这需要计算(使用 Graphics.MeasureString()方法)。另外，屏幕上文字占用的间隔会影响文档中后续的文字在屏幕上放置的位置。如果应用程序自动换行，就需要在确定应在何处放置换行符前仔细斟酌文字的大小。下次运行 Microsoft 的 Word 时，仔细看看 Word 是如何重新定位用户输入的文本的。这里有许多复杂的处理操作。有可能任何 GDI+ 应用程序都不像 Word 那样复杂，但如果需要显示任何文本，仍需要考虑同样的问题。

总之，好的文本处理需要一定的技巧。假定知道字体和放置它的位置，那么把一行文本显示在屏幕上实际上非常简单。因此，下一节介绍一个小示例，说明如何显示一些文本。此后，探讨字体和字体系列的一些规则，并介绍一个更真实(和相关)的文本处理示例 CapsEditor。

48.12 简单的文本示例

这个示例——DisplayText 是常见的 Windows 窗体效果。这次重写了 OnPaint()方法，添加了成员字段，如下所示：



```
private readonly Brush blackBrush = Brushes.Black;
private readonly Brush blueBrush = Brushes.Blue;
private readonly Font haettenschweilerFont = new Font("Haettenschweiler", 12);
private readonly Font boldTimesFont = new Font("Times New Roman", 10,
    FontStyle.Bold);
private readonly Font italicCourierFont = new Font("Courier", 11,
    FontStyle.Italic | FontStyle.Underline);

protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    dc.DrawString("This is a groovy string", haettenschweilerFont, blackBrush,
        10, 10);
    dc.DrawString("This is a groovy string " +
        "with some very long text that will never fit in the box",
        boldTimesFont, blueBrush,
        new Rectangle(new Point(10, 40), new Size(100, 40)));
    dc.DrawString("This is a groovy string", italicCourierFont, blackBrush,
        new Point(10, 100));
}
```

代码段 DisplayText.sln

运行这个示例，会得到如图 48-15 所示的结果。

这个示例说明了如何使用 Graphics.DrawString()方法绘制文本项，DrawString()方法有许多重载版本，这里介绍其中的 3 个。但这些不同的重载方法都需要用参数指定要显示的文本、绘制字符串所使用的字体，以及用于构造各种线条和曲线以组成每个文本字符的画笔。其余的参数有另外两种指定方式。但一般情况下，可以指定一个 Point(或两个等价的数字)或一个 Rectangle。

如果指定 Point，文本就从该 Point 的左上角开始，并仅向右延伸。如果指定 Rectangle，Graphics 实例就把字符串放在该矩形的内部。如果文本在矩形内部容纳不下，它就会被剪切，如图 48-15 中的第 4 行文本所示。把矩形传递给 DrawString()方法，表示绘图过程将持续较长时间，因为 DrawString()方法需要指定在什么地方放置换行符，但其结果应看起来好一些(如果字符串可以放在矩形中)。



图 48-15

这个示例还说明了构造字体的两种方法，一般需要包含字体的名称及其大小(高度)。也可以选择性地传递不同的样式以修改文本的绘制方式(黑体、下划线等)。

48.13 字体和字体系列

字体准确地描述每个字母的显示方式。在一个文档中选择合适的字体和提供适当数量的不同字体，对于提高该文档的可读性非常重要。

大多数人在提到字体时，会指定 Arial、Times New Roman(Windows 用户)或 Times、Helvetica(Mac OS 用户)等。实际上，这些都不是字体，它们是字体系列(font families)，字体系列以通俗的术语来说，是文本的可视化风格。字体系列是应用程序整体外观中的一个关键因素，大多数人都能识别常用字体系列的风格，即使我们意识不到这一点。

而实际字体应是像 Arial 9-point italic 这样的东西。换言之，字体添加了更多的信息，如指定文本的大小，以及是否对该文本进行其他修改等。这些修改包括文本是否为黑体、斜体、带下划线，或者显示为小型大写字母或下标，这种修改在技术上称为样式，尽管在某些情况下该术语有误导作用，因为可视化外观主要取决于字体系列。

通过指定文本的高度，就可以测量其大小。高度以点为单位来测量，这是一个传统单位，表示 1/72 英寸(0.351 毫米)。例如，10 点的字体高度大约为 1/7 英寸或 3.5 毫米。但字体大小为 10 点的 7 行文本，在屏幕或纸上的高度不等于 1 英寸，因为还需要考虑行与行之间的间隔。



严格来讲，测量高度并不像这样简单，因为还需要考虑几个不同的高度。例如，较高字母，如 A 或 F 的高度(这是指我们讨论高度时字母的真实高度)，重音字母如 Å 或 Ñ 中的额外高度(内部前导)，以及字母的尾部超出底线的附加高度如 y 和 g(下行高度)。但是本章不讨论这些高度，一旦指定了字体系列和主要高度，这些辅助高度就会自动确定。

在处理字体时，还会遇到其他常用于描述某种字体系列的术语：

- **Serif** 字体系列在组成字符的许多线条尾部有一个小标记(这些标记称为衬线)。Times New Roman 就是这种字体的一个典型例子。
- 相反，**Sans Serif** 字体系列没有这些小标记。例如，Arial 和 Verdana。没有小标记常会使文本看起来比较生硬，所以 Sans Serif 字体常用于重要的文本。
- **TrueType** 字体系列以一种精确的数学方式定义组成字符的曲线形状。这表示可以使用相同的定义来计算如何在系列内部绘制任意大小的字体。目前，我们实际上使用的所有字体都是 TrueType 字体。Windows 3.1 时代的一些旧字体系列通过指定每种字体大小的每个字符的位图来定义，但现在最好不要使用这些字体。

Microsoft 提供了两个主要类来处理何时选择或处理字体：

- `System.Drawing.Font`
- `System.Drawing.FontFamily`

前面已经介绍了 `Font` 类的主要用法。在绘制文本时，实例化 `Font` 的一个实例，并把它传递给 `DrawString()` 方法，以确定应该如何绘制文本。`FontFamily` 实例用于表示一个字体系列。

FontFamily 类的一个用法是：如果知道需要某种类型的字体(Serif、Sans Serif 或 TrueType)，但不介意使用哪种字体，就可以使用该类。静态属性 GenericSerif、GenericSansSerif 和 GenericMonospace 返回满足这些条件的默认字体：

```
FontFamily sansSerifFont = FontFamily.GenericSansSerif;
```

然而，如果编写一个高级应用程序，就应以更先进的方法选择字体。可以实现绘图代码，从而使它检查哪些字体系列可用，并选择合适的字体，例如从偏爱的字体列表中选择第一种可用的字体。进一步地，如果要让应用程序的用户友好性更高，则列表中的第一个选项可能是用户上次运行软件时选择的字体。通常情况下，安全起见，最好使用最常用的字体系列，如 Arial 和 Times New Roman。但如果要使用某种不存在的字体显示文本，则结果通常总是不可预料的。很容易发现，Windows 仅是替代了标准系统字体，从而系统很容易绘制文本，但这看起来并不是非常友好，如果文档出现这种情况，就会使人觉得软件的质量非常糟糕。

使用 `InstalledFontCollection` 类可以查看系统上的可用字体，该类在 `System.Drawing.Text` 名称空间中。这个类实现一个属性 `Families`，该属性是一个包含系统上所有可用字体的数组：

```
InstalledFontCollection insFont = new InstalledFontCollection();
FontFamily [] families = insFont.Families;

foreach (FontFamily family in families)
{
    // do processing with this font family
}
```

48.14 示例：枚举字体系列

本节介绍一个小示例 `EnumFontFamilies`，该示例列出系统上所有可用的字体系列，使用合适的字体(该字体系列的 12 点常规版本)显示它们的名称。运行这个示例，会得到如图 48-16 所示的结果。

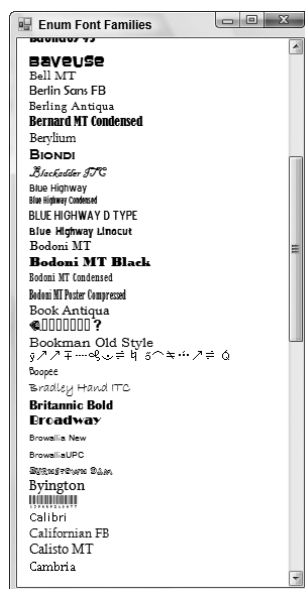


图 48-16

当然，根据计算机上安装的字体，用户在运行这个示例时，可能会得到不同的结果。

对于这个示例，创建一个标准的 C# Windows 应用程序 ListFonts，首先添加要搜索的另一个名称空间。我们要使用 InstalledFontCollection 类，它在 System.Drawing.Text 名称空间中定义：

```
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;
```

然后在 Form1 类中添加如下常量：

```
private const int margin = 10;
```

margin 是文本与文档边缘之间的左边距和上边距的大小——它防止文本显示在工作区边缘的右边。

由于该示例以快捷方式显示字体系列，因此，代码比较粗糙，在许多情况下并不是以实际应用程序中的方式执行任务。例如，我们把文档的大小设置为一个估计值(200,1500)，使用 Visual Studio 2010 的 Properties 窗口把 AutoScrollMinSize 属性设置为这个值。一般情况下，必须查看要显示的文本，计算出文档的大小。下一节介绍这个过程。

下面是 OnPaint()方法：



```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    int verticalCoordinate = margin;
    InstalledFontCollection insFont = new InstalledFontCollection();
    FontFamily [] families = insFont.Families;
    e.Graphics.TranslateTransform(AutoScrollPosition.X,
                                AutoScrollPosition.Y);
    foreach (FontFamily family in families)
    {
        if (family.IsStyleAvailable(FontStyle.Regular))
        {
            Font f = new Font(family.Name, 10);
            Point topLeftCorner = new Point(margin, verticalCoordinate);
            verticalCoordinate += f.Height;
            e.Graphics.DrawString (family.Name, f,
                                Brushes.Black, topLeftCorner);
            f.Dispose();
        }
    }
}
```

代码段 ListFonts.sln

在这段代码中，首先使用 InstalledFontCollection 对象获得一个数组，其中包含所有可用的字体系列的详细信息。对于每个系列，我们实例化大小为 10 点的一个 Font。为 Font 使用了一个简单的构造函数——有许多构造函数可以指定更多的选项。使用的构造函数带有两个参数：系列的名称和字体的大小：

```
Font f = new Font(family.Name, 10);
```

这个构造函数构造了一个一般风格的字体。但是为了安全起见，在使用该字体显示任何文本前，

先检查一下这种风格是否可用于每个字体系列。这利用方法 `FontFamily.IsStyleAvailable()` 实现。这种检查非常重要，因为并不是所有的字体都可以使用所有的风格：

```
if (family.IsStyleAvailable(FontStyle.Regular))
```

`FontFamily.IsStyleAvailable()` 方法带一个参数，即 `FontStyle` 枚举。该枚举包含许多标记，它们可以用按位 OR 运算符来组合。可能的标记有 `Bold`、`Italic`、`Regular`、`Strikeout` 和 `Underline`。

最后，使用 `Font` 类的一个属性 `Height`，该属性返回显示该字体对应的文本需要的高度，以便算出行间距：

```
Font f = new Font(family.Name, 10);
Point topLeftCorner = new Point(margin, verticalCoordinate);
verticalCoordinate += f.Height;
```

为了使事情简单化，`OnPaint()` 方法的这个版本暴露出一些不太好的编程问题。首先，没有检查哪些文档区域需要绘制——而是显示所有内容。另外，如前所述，因为实例化 `Font` 是一个计算量较大的过程，所以应保存字体，而不是每次调用 `OnPaint()` 方法时都实例化新副本。因为这样设计出来的代码对于本示例将需要花费相当长的时间来绘图。为了节省内存，帮助垃圾回收器，在完成操作后对每个 `Font` 实例调用 `Dispose()` 方法。如果不这样，在 10 或 20 次绘图操作后，就会浪费许多内存存储不再需要的字体。

48.15 编辑文本文档：CapsEditor 示例

下面是本章中经过扩展的一个示例。`CapsEditor` 示例用于说明如何把前面介绍的绘图规则应用到实际环境中。这个示例除了响应用户通过鼠标输入的信息外，不需要任何新资料，但它将说明如何管理文本的绘制，让应用程序仍具有高性能，并确保主窗口的工作区的内容总是最新的。

`CapsEditor` 程序允许用户读取文本文件，该文本逐行显示在工作区中。如果用户双击任何一行文本，该行就会全部变为大写。这就是该示例的功能。即使只有这组有限的功能，也涉及许多复杂的工作：确保把所有的信息都显示在正确的位置上，并考虑性能问题。特别是这里有一个新元素，文档的内容可以修改——用户选择菜单项，以读取一个新文件时；或用户双击一行，使之全部变为大写字母时，都要修改文档的内容。在第一种情况下，需要更新文档的大小，使滚动条仍能正确工作，并重新显示所有内容。在第二种情况下，需要仔细检查文档的大小是否发生了变化，哪些文本需要重新显示。

本节首先看看 `CapsEditor` 的外观。第一次运行该应用程序时，它没有已加载的文档，显示结果如图 48-17 所示。

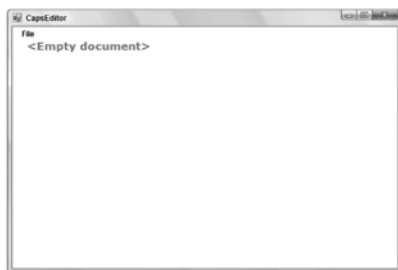


图 48-17

File 菜单有两个菜单项: Open 和 Exit。Open 调用 OpenFileDialog()方法, 读取用户选择的任何文件 Exit 退出应用程序。图 48-18 是 CapsEditor 显示其源文件 Form1.cs 的情形(我们已经在该图片中双击几行, 把它们全部转换为大写)。

水平和垂直滚动条的大小也是正确的。滚动工作区, 使之正好显示整个文档。CapsEditor 不会给文本换行——即使没有这个功能, 该示例也比较复杂了。它只显示文件被读取的各行文本。对文件的大小没有限制, 但这里假定这是一个文本文件, 且不包含任何非打印字符。

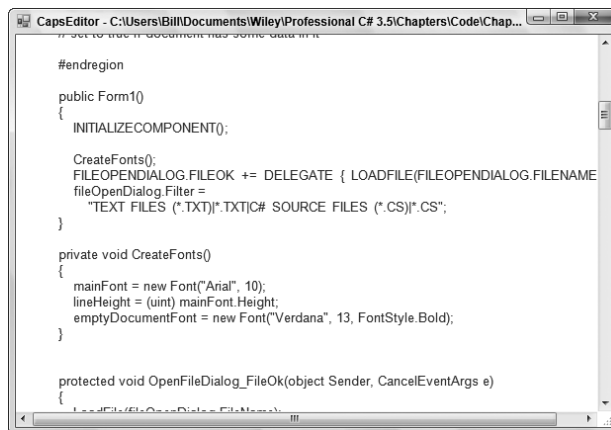


图 48-18

首先添加一些 using 命令:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.IO;
using System.Windows.Forms;
```

这是因为我们使用了 StreamReader 类, 它在 System.IO 名称空间中。接着, 在 Form1 类中添加一些字段:



可从
wrox.com
下载源代码

```
#region Constant fields
private const string standardTitle = "CapsEditor";
// default text in titlebar
private const uint margin = 10;
// horizontal and vertical margin in client area
#endregion
#region Member fields
// The 'document'
private readonly List<TextLineInformation> documentLines =
    new List<TextLineInformation>();
private uint lineHeight; // height in pixels of one line
private Size documentSize; // how big a client area is needed to
// display document
private uint nLines; // number of lines in document
private Font mainFont; // font used to display all lines
private Font emptyDocumentFont; // font used to display empty message
```

```

private readonly Brush mainBrush = Brushes.Blue;
                                // brush used to display document text
private readonly Brush emptyDocumentBrush = Brushes.Red;
                                // brush used to display empty document message
private Point mouseDoubleClickPosition;
    // location mouse is pointing to when double-clicked
private readonly OpenFileDialog fileOpenDialog = new OpenFileDialog();
    // standard open file dialog
private bool documentHasData = false;
    // set to true if document has some data in it
#endregion

```

代码段 CapsEditor.sln

这些字段中的大多数都很容易理解。`documentLines` 字段是一个 `List<TextLineInformation>`，它包含文件中已经读入的实际文本。实际上，这个字段包含了文档中的数据。`documentLines` 的每个元素都包含一行已读入的文本的信息。因为它是一个 `List<TextLineInformation>`，而不是常见的数组，所以在读取文件时，可以给它动态地添加元素。

如前所述，每个 `documentLines` 元素都包含一行文本的相关信息。这些信息实际上是另一个类 `TextLineInformation` 的一个实例：

```

class TextLineInformation
{
    public string Text;
    public uint Width;
}

```

`TextLineInformation` 类看起来像一个典型的示例，其中宁愿使用结构，而不是类，因为它只是把几个字段组合在一起。但它的实例总是作为 `List<TextLineInformation>` 的元素来访问，这样其元素就会存储为引用类型。

每个 `TextLineInformation` 实例都存储了一行文本——它可以看作是显示为一项的最小项，一般情况下，在 GDI+应用程序中，对于每个这样的项，都要存储该项的文本，以及显示它的世界坐标和其大小(只要用户进行了滚动操作，页面坐标通常就会改变，而世界坐标通常只有在文档的其他部分以某种方式进行了修改时才会改变)。本例只需存储该项的 `Width` 即可。其原因是此时的高度是选中字体的高度，它对于所有的文本行都一样，所以不必为每行文本单独地存储高度。它只需要在字段 `Form1.lineHeight` 字段中存储一次即可，位置也是这样。在这里，`x` 坐标等于页边距，`y` 坐标很容易计算出来：

```
margin + lineHeight*(本行上面显示的行数)
```

如果要显示和操作单个单词，而不是整行文本，则每个单词的 `x` 坐标必须使用该文本行上在该单词前面的所有单词的宽度来计算，但为了使这个计算简单一些，应把每行文本当作单一的项来看待。

下面处理主菜单。这部分应用程序涉及更多的是 Windows 窗体的内容(参见第 39 章)，而不是 GDI+ 的内容。在 Visual Studio 2010 中使用设计视图添加菜单项，把它们重命名为 `menuFile`、`menuFileOpen` 和 `menuFileExit`。接着使用 Visual Studio 2010 的 Properties 窗口添加 File Open 和 File Exit 菜单项的事件处理程序。这些事件处理程序的名称是 Visual Studio 2010 生成的，即 `menuFileOpen_Click()` 和 `menuFileExit_Click()`。

还需要在 Form1()构造函数中添加一些初始化代码:



```
public Form1()
{
    InitializeComponent();
    CreateFonts();
    fileOpenDialog.FileOk += delegate { LoadFile(fileOpenDialog.FileName); };
    fileOpenDialog.Filter =
        "Text files ( * .txt)| * .txt|C# source files ( * .cs)| * .cs";
}
```

代码段 CapsEditor.sln

这里为实例添加的事件处理程序是在用户单击 File Open 对话框中的 OK 按钮时执行的。我们还给 Open File 对话框设置了筛选器,只能加载文本文件。因为本示例选择了.txt文件和C#源文件,所以可以使用应用程序查看示例的源代码。

CreateFonts()是一个辅助方法,它挑选出要使用的字体:

```
private void CreateFonts()
{
    mainFont = new Font("Arial", 10);
    lineHeight = (uint)mainFont.Height;
    emptyDocumentFont = new Font("Verdana", 13, FontStyle.Bold);
}
```

处理程序的实际定义非常标准:

```
protected void menuFileOpen_Click(object sender, EventArgs e)
{
    fileOpenDialog.ShowDialog();
}
protected void menuFileExit_Click(object sender, EventArgs e)
{
    Close();
}
```

下面介绍 LoadFile()方法。这个方法处理文件的打开和读取操作,并确保引发 Paint 事件,以使用新文件强制重新绘图:



```
private void LoadFile(string FileName)
{
    StreamReader sr = new StreamReader(FileName);
    string nextLine;
    documentLines.Clear();
    nLines = 0;
    TextLineInformation nextLineInfo;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nextLineInfo = new TextLineInformation();
        nextLineInfo.Text = nextLine;
        documentLines.Add(nextLineInfo);
        ++nLines;
    }
    sr.Close();
}
```

```

        documentHasData = (nLines > 0) ? true: false;
        CalculateLineWidths();
        CalculateDocumentSize();
        Text = standardTitle + "-" + FileName;
        Invalidate();
    }

```

代码段 CapsEditor.sln

这个函数的大部分代码都是标准的文件读取操作(参见第 29 章)。注意在读取文件时,逐渐地给 documentLines ArrayList 添加文本行,使这个数组最终按顺序包含每行文本的信息。在读取文件后,设置 documentHasData 标记,它指定是否有要显示的信息。下一个任务是确定要显示内容的位置,之后确定显示文件的工作区有多大,以及用于设置滚动条的文档大小。最后,设置标题栏文本,并调用 Invalidate()方法。Invalidate()是 Microsoft 提供的一个非常重要的方法,所以下一节首先介绍这个方法的应用,之后介绍 CalculateLineWidths()和 CalculateDocument Size()方法的代码。

48.15.1 Invalidate()方法

Invalidate()方法是 System.Windows.Forms.Form 的一个成员,它把窗口的工作区标记为无效,因此在需要重新绘制时,它可以确保引发 Paint 事件。Invalidate()方法有两个重载版本:可以给它传递一个矩形,该矩形指定(使用页面坐标)需要重新绘制窗口的哪个区域。如果不提供任何参数,它就把整个工作区标记为无效。

如果知道需要绘制某些内容,为什么不调用 OnPaint()方法或直接完成绘制任务的其他方法?原因是,一般情况下,最好不要直接调用绘图例程——如果代码要完成某些绘图任务,一般就应调用 Invalidate()方法。其原因如下所示:

- 因为绘图总是 GDI+应用程序执行的一种处理器最密集型的任务,所以在其他工作的中间进行绘图会妨碍其他工作的进行。在前面的示例中,如果从 LoadFile()方法中直接调用一个方法来完成绘图,LoadFile()方法就将在绘图工作完成后才能返回。在这段时间里,应用程序不会响应任何其他事件。另一方面,通过调用 Invalidate()方法,在从 LoadFile()方法快速返回之前,仅可以让 Windows 引发一个 Paint 事件。接着 Windows 就可以检查等待处理的事件。其内部的工作方式是事件被当作消息队列中的一条消息。Windows 会定期检查该队列,如果其中有事件,Windows 就选择它,并调用相应的事件处理程序。现在 Paint 事件是队列中的唯一事件,所以 OnPaint()方法会被立即调用。但是,在一个比较复杂的应用程序中,可能会有其他事件,其中一些优先级比 Paint 事件高。特别是如果用户已决定退出应用程序,该事件就会用消息 WM_QUIT 来标记。
- 如果有一个比较复杂的多线程应用程序,就会希望用一个线程处理所有的绘图操作。使用 Invalidate()方法可以把所有的绘图操作传递到消息队列中,这有助于确保无论其他线程请求什么绘图操作,都由同一个线程完成所有的绘图操作(无论什么线程负责消息队列,都是由线程 Application.Run()方法处理绘图操作)。
- 还有一个与性能有关的原因。假定在同一时刻有几个不同的屏幕绘制请求,也许代码刚刚修改文档,以确保显示更新后的文档,而同时用户刚刚移开另一个覆盖部分工作区的窗口。调用 Invalidate()方法,可以让 Windows 有机会注意到发生的事件。Windows 就会在需要时合并 Paint 事件,合并无效的区域,这样绘图操作就只执行一次。

- 最后，执行绘图的代码可能是应用程序中最复杂的代码部分，特别是当有一个比较复杂的用户界面时，就更是如此。需要长时间维护该代码的人员希望我们把所有的绘图代码都放在一个地方，且尽可能简单——如果程序的其他部分没有过多的路径进入该部分代码，维护就更容易。

其底线是最好把所有的绘图代码都放在 `OnPaint()` 例程中，或者从该方法中调用的其他方法中。但是要维持一个平衡。如果要在屏幕上替换一个字符，最好不要影响到已经绘制好的其他内容，那么此时可能不需要使用 `Invalidate()` 方法(因为系统开销过大)，而只需编写一个独立的绘图例程。



在非常复杂的应用程序中，甚至可以编写一个完整的类，它专门负责在屏幕上绘图。几年前当 MFC 仍是 GDI 密集型应用程序的标准技术时，MFC 就遵循这种模型，使用一个 C++ 类 `C<ApplicationName>View` 完成绘图操作。但即使是这样，这个类也有一个成员函数 `OnDraw()`，它用作大多数绘图请求的入口点。

48.15.2 计算项的大小和文档的大小

下面返回 `CapsEditor` 示例，介绍从 `LoadFile()` 方法中调用的 `CalculateLineWidths()` 和 `CalculateDocumentSize()` 方法：



可从
wrox.com
下载源代码

```
private void CalculateLineWidths()
{
    Graphics dc = this.CreateGraphics();
    foreach (TextLineInformation nextLine in documentLines)
    {
        nextLine.Width = (uint)dc.MeasureString(nextLine.Text,
            mainFont).Width;
    }
}
```

代码段 `CapsEditor.sln`

这个方法仅遍历已经读取的每行文本，并使用 `Graphics.MeasureString()` 方法计算和存储字符串在屏幕上需要的水平间距。存储这个值是因为 `MeasureString()` 方法要进行大量的计算。如果 `CapsEditor` 示例不够简单，不能很容易地计算出每一行的高度和位置，那么这个方法也肯定需要按计算所有量的方式来实现。

知道屏幕上每一项的大小后，就可以计算每一项的位置，并计算出文档的实际大小。高度是每行文本的高度乘以行数。宽度则需要计算，方法是遍历每行文本，确定哪一行最长。对于高度和宽度，也可以在显示的文档周围设置一个较小的页边距，从而使应用程序看起来更吸引人。

下面是计算文档大小的方法：



可从
wrox.com
下载源代码

```
private void CalculateDocumentSize()
{
    if (!documentHasData)
    {
        documentSize = new Size(100, 200);
    }
    else
```

```

{
    documentSize.Height = (int)(nLines*lineHeight) + 2*(int)margin;
    uint maxLineLength = 0;
    foreach (TextLineInformation nextWord in documentLines)
    {
        uint tempLineLength = nextWord.Width;
        if (tempLineLength > maxLineLength)
        {
            maxLineLength = tempLineLength;
        }
    }
    maxLineLength += 2*margin;
    documentSize.Width = (int)maxLineLength;
}
AutoScrollMinSize = documentSize;
}

```

代码段 CapsEditor.sln

这个方法首先检查是否有数据要显示。如果没有，就使用硬编码的文档大小，该大小足以显示很大的红色<Empty Document>警告。如果要正确显示数据，就应使用 MeasureString()方法确定警告信息到底有多大。

计算出文档大小后，就设置 Form.AutoScrollMinSize 属性，以告诉 Form 实例文档有多大。完成后，后台就会发生一些有趣的事。在设置这个属性的过程中，工作区会标记为无效，并引发 Paint 事件。出于非常合理的原因，改变文档的大小就意味着需要添加或修改滚动条，需要重新绘制整个工作区。为什么说这很有趣？如果回过头来看看 LoadFile()方法的代码，就会发现在该方法中调用 Invalidate()方法实际上是多余的。在设置文档大小时，肯定要使工作区无效。在 LoadFile()方法中显式调用 Invalidate()方法，说明了在一般情况下应如何完成任务。实际上，在这个示例中，再次调用 Invalidate()方法将重复请求 Paint 事件，这是不必要的。但是，这依次说明了 Invalidate()方法如何给 Windows 一个优化性能的机会。第二个 Paint 事件实际上并不会被引发：Windows 发现队列中已经有一个 Paint 事件，就会比较请求的无效区域，看看是否需要合并它们。在本例中，因为两个 Paint 事件都指定了整个工作区，所以不需要合并，Windows 会撤销第二个 Paint 请求。当然，这个过程会占用处理器一定的时间，但与某些绘图操作所占用的时间相比，它可以忽略不计。

48.15.3 OnPaint()方法

前面介绍了 CapsEditor 如何加载文件，下面看看如何绘图：



可从
wrox.com
下载源代码

```

protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics dc = e.Graphics;
    int scrollPositionX = AutoScrollPosition.X;
    int scrollPositionY = AutoScrollPosition.Y;
    dc.TranslateTransform(scrollPositionX, scrollPositionY);
    if (!documentHasData)
    {
        dc.DrawString("<Empty document>", emptyDocumentFont,
            emptyDocumentBrush, new Point(20, 20));
        base.OnPaint(e);
    }
}

```

```

        return;
    }
    // work out which lines are in clipping rectangle
    int minLineInClipRegion =
        WorldYCoordinateToLineIndex(e.ClipRectangle.Top -
                                    scrollPositionY);

    if (minLineInClipRegion == -1)
    {
        minLineInClipRegion = 0;
    }
    int maxLineInClipRegion =
        WorldYCoordinateToLineIndex(e.ClipRectangle.Bottom -
                                    scrollPositionY);

    if (maxLineInClipRegion >= documentLines.Count ||
        maxLineInClipRegion == -1)
    {
        maxLineInClipRegion = documentLines.Count-1;
    }
    TextLineInformation nextLine;
    for (int i=minLineInClipRegion; i<=maxLineInClipRegion; i++)
    {
        nextLine = (TextLineInformation)documentLines[i];
        dc.DrawString(nextLine.Text, mainFont, mainBrush,
                      LineIndexToWorldCoordinates(i));
    }
}

```

代码段 CapsEditor.sln

这个OnPaint()重载方法的核心是一个循环,它迭代文档的每行文本,同时调用Graphics.DrawString()方法绘制每行文本。其余的代码主要是关于优化绘图操作——通常是精确地确定哪些部分需要绘制,而不是告诉图形实例绘制所有内容。

首先检查一下文档中是否有数据。如果没有,就显示一条消息,说明调用基类的 OnPaint()方法并退出。如果有数据,就开始查看剪切的矩形,方法是调用另一个方法 WorldYCoordinateToLineIndex()。后面再介绍这个方法,但本质上该方法带一个相对于文档顶部的 y 坐标,计算此时应显示文档中的哪些文本。

第一次调用 WorldYCoordinateToLineIndex()方法时,给它传递坐标值 e.ClipRectangle.Top - scrollPositionY,这是剪切区域的顶部(把它转换为世界坐标后)。如果返回值为-1,就是安全的,并假定需要从文档的开头开始(如果剪切区域位于顶部边距中,就会出现这种情况)。

完成后,必须对剪切矩形的底部重复相同的过程,找出文档在剪切区域中的最后一行文本。第一行和最后一行的索引分别存储在 minLineInClipRegion 和 maxLineInClipRegion 中。这样就可以在这些值之间运行 for 循环,以执行绘图操作。在绘图循环中,实际上需要通过 WorldYCoordinateToLineIndex()方法粗略地进行逆转换:给出一行文本的索引,要确定该行文本应绘制在什么位置。这个计算实际上相当简单,我们把它封装到另一个方法 LineIndexToWorldCoordinates()中,它返回该行文本左上角的坐标。尽管返回的坐标是世界坐标,但这很好,因为我们已在 Graphics 对象上调用了 TranslateTransform()方法,所以在要求显示文本时,需要给它传递世界坐标,而不是页面坐标。

48.15.4 坐标转换

本节介绍在 CapsEditor 示例中编写的几个辅助方法，以帮助进行坐标转换。这些方法是上一节提到的 WorldYCoordinateToLineIndex()和 LineIndexToWorldCoordinates()方法，还有几个其他的方法。

首先，LineIndexToWorldCoordinates()方法的参数是一个给定的行索引，它使用已知的页边距和行高度，计算出该行左上角的世界坐标：



```
private Point LineIndexToWorldCoordinates(int index)
{
    Point TopLeftCorner = new Point(
        (int)margin, (int)(lineHeight*index + margin));
    return TopLeftCorner;
}
```

代码段 CapsEditor.sln

我们还使用一个方法在 OnPaint()方法中粗略地进行逆转换。WorldYCoordinateToLineIndex()方法可计算出行索引，但它只考虑了一个垂直的世界坐标。这是因为它相对于剪切区域的顶部和底部来计算行索引。

```
private int WorldYCoordinateToLineIndex(int y)
{
    if (y < margin)
    {
        return -1;
    }
    return (int)((y-margin)/lineHeight);
}
```

还有 3 个方法，它们从处理程序例程中调用，这些例程响应用户的双击鼠标操作。首先，用一个方法计算出要在给定世界坐标处显示的文本行的索引。与 WorldYCoordinateToLineIndex()方法不同，这个方法考虑了 x 坐标和 y 坐标，如果在该坐标上没有文本行，它就返回-1：

```
private int WorldCoordinatesToLineIndex(Point position)
{
    if (!documentHasData)
    {
        return -1;
    }
    if (position.Y < margin || position.X < margin)
    {
        return -1;
    }
    int index = (int)(position.Y-margin)/(int)this.lineHeight;
    // check position is not below document
    if (index >= documentLines.Count)
    {
        return -1;
    }
    // now check that horizontal position is within this line
    TextLineInformation theLine =
        (TextLineInformation)documentLines[index];
    if (position.X > margin + theLine.Width)
```

```
    {
        return -1;
    }
    // all is OK. We can return answer
    return index;
}
```

最后，需要在行索引和页面坐标(而不是世界坐标)之间转换，下面的方法完成这个任务：

```
private Point LineIndexToPageCoordinates(int index)
{
    return LineIndexToWorldCoordinates(index) +
        new Size(AutoScrollPosition);
}
private int PageCoordinatesToLineIndex(Point position)
{
    return WorldCoordinatesToLineIndex(position-new
        Size(AutoScrollPosition));
}
```

注意在转换到页面坐标时，添加了 `AutoScrollPosition`，这是一个负值。

虽然这些方法看起来并不是很有趣，但它们说明了需要经常使用的一个技巧。在 `GDI+`中，系统常常会给出一些特定坐标(如用户在此处单击鼠标的坐标)，而我们需要确定在该坐标处要显示什么内容。或者相反——给出要显示的内容，确定它在什么地方显示。因此，如果编写一个 `GDI+`应用程序，就会发现编写完成等价的坐标转换的方法很有效。

48.15.5 响应用户输入

到目前为止，除了 `CapsEditor` 示例中的 `File` 菜单外，本章介绍的所有内容都是单向的：应用程序告诉用户一些信息，方法是在屏幕上显示它们。当然，几乎所有的软件都是双向的：用户也可以与应用程序进行交互。下面就把这个功能添加到 `CapsEditor` 示例中。

让 `GDI+`应用程序响应用户输入，实际上比编写代码在屏幕上绘图更简单。实际上第 39 章已经介绍了如何处理用户输入。即重写 `Form` 类的方法，这些方法从相关的事件处理程序中调用。在引发 `Paint` 事件时，就是以这种方式调用 `OnPaint()`方法。

当用户单击或移动鼠标时，可以重写的方法如表 48-5 所示。

表 48-5	
方 法	何 时 调 用
<code>OnClick(EventArgs e)</code>	单击鼠标
<code>OnDoubleClick(EventArgs e)</code>	双击鼠标
<code>OnMouseDown(MouseEventArgs e)</code>	按鼠标左键
<code>OnMouseHover(MouseEventArgs e)</code>	鼠标在移动后仍停留在某处
<code>OnMouseMove(MouseEventArgs e)</code>	移动鼠标
<code>OnMouseUp(MouseEventArgs e)</code>	释放鼠标左键

如果要检测用户什么时候输入文本，就可以重写表 48-6 中的方法。

表 48-6

方 法	何 时 调 用
OnKeyDown(KeyEventArgs e)	按下一个键
OnKeyPress(KeyPressEventArgs e)	按下并释放一个键
OnKeyUp(KeyEventArgs e)	释放被按下的键

注意，其中的一些事件是重叠的。例如，如果用户按下鼠标按钮，就会引发 `MouseDown` 事件。如果立即释放按钮，就会引发 `MouseUp` 事件和 `Click` 事件。另外，一些方法接受一个派生自 `EventArgs` 的参数，而不是 `EventArgs` 本身的实例。派生类的这些实例可用于给出特定事件的更多信息。`MouseEventArgs` 有两个属性(X 和 Y)，它们给出鼠标按下时设备的坐标。`KeyEventArgs` 和 `KeyPressEventArgs` 的属性则指定与事件相关的键。

这就是用户响应方法。用户应考虑自己要完成的工作的逻辑。唯一要注意的是，编写 GDI+应用程序可能要比编写 `Windows.Forms` 应用程序做更多的逻辑工作，这是因为在 `Windows.Forms` 应用程序中，一般响应的是比较高级的事件(例如，文本框的 `TextChanged` 事件)。GDI+则相反，其中的事件都比较基本，用户单击鼠标，或按 H 键。应用程序采取的操作取决于一系列事件，而不是单个事件。例如，假定应用程序的工作方式类似于 `Microsoft Word for Windows`：为了选择一些文本，用户通常首先要单击鼠标左键，再移动鼠标，最后释放鼠标左键。虽然应用程序接收到 `MouseDown` 事件，但仅有这个事件是不能完成更多任务，它只是记录下该鼠标单击时光标的位置。那么，当接收到 `MouseDown` 事件时，就需要检查刚才的记录，确定鼠标左键是否被按下，如果按下，突出显示的文本就是用户选择的文本。当用户释放鼠标左键时，对应操作(在 `OnMouseUp()`方法中)就需要检查：按下的鼠标按钮是否有拖动操作，在方法中是否有相应的操作。只有这样，这个序列才算完成。

另外，因为某些事件有重叠，常常要选择希望代码响应哪个事件。

重要规则是仔细考虑用户可能启动的鼠标移动或单击和键盘事件的每个组合的逻辑，确保应用程序以直观的方式响应，使应用程序在各种情况下都按照期望的方式执行。我们的工作大多数是思考，而不是编码，尽管通过编码工作很复杂，因为我们需要考虑用户输入的许多组合。例如，如果用户开始输入文本，而鼠标按钮处于按下状态，应用程序该如何响应？这听起来是不可能的，但最终用户会尝试这么做的！

在 `CapsEditor` 示例中，为了使工作尽可能简单，并没有真正考虑用户输入的任何组合。在本例中只响应用户的双击，此时把光标悬停在那行文本变为大写字母。

这应是一个相当简单的任务，但有一个障碍，需要捕获 `DoubleClick` 事件。但表 48-6 说明，这个事件接受一个 `EventArgs` 参数，不是一个 `MouseEventArgs` 参数。问题是如果要把一行文本正确地标识为大写，我们需要知道用户双击时光标在什么地方，为此又需要一个 `MouseEventArgs` 参数。有两个变通方法，一个方法是使用由 `Form1` 对象 `Control.MousePosition` 实现的一个静态方法，来确定光标的位置：



可从
wrox.com
下载源代码

```
protected override void OnDoubleClick(EventArgs e)
{
    Point MouseLocation = Control.MousePosition;
    // handle double click
}
```

代码段 `CapsEditor.sln`

在大多数情况下,这个方法可以正常工作。但如果应用程序(甚至是其他一些有较高优先级的应用程序)在用户双击时进行某些计算密集型的工作,该方法就会有問題。此时要在用户双击之后大约半秒钟内,才调用 `OnDoubleClick()` 事件处理程序。我们不期望有这样的延迟,因为这会让用户感到很厌烦,但有时因为应用程序不能控制的原因(如计算机较慢),会偶尔发生这种情况。问题是半秒的时间足够光标在屏幕上移动到其他位置了——此时调用 `Control.MousePosition` 会返回完全错误的位置!

比较好的方法是依赖于鼠标事件之间的其中一个重叠。双击鼠标的第一部分涉及按下鼠标左键。这表示如果调用 `OnDoubleClick()` 方法,我们就知道刚调用 `OnMouseDown()` 方法,此时鼠标的位置不变。可以使用 `OnMouseDown()` 方法的重写版本记录光标的位置,为 `OnDoubleClick()` 方法做准备。这就是在 `CapsEditor` 中采用的方法:

```
protected override void OnMouseDown(MouseEventArgs e)
{
    base.OnMouseDown(e);
    mouseDoubleClickPosition = new Point(e.X, e.Y);
}
```

下面查看 `OnDoubleClick()` 方法的重写版本,这里该方法要完成许多工作:



```
protected override void OnDoubleClick(EventArgs e)
{
    int i = PageCoordinatesToLineIndex(mouseDoubleClickPosition);
    if (i >= 0)
    {
        TextLineInformation lineToBeChanged =
            (TextLineInformation)documentLines[i];
        lineToBeChanged.Text = lineToBeChanged.Text.ToUpper();
        Graphics dc = this.CreateGraphics();
        uint newWidth = (uint)dc.MeasureString(lineToBeChanged.Text,
            mainFont).Width;

        if (newWidth > lineToBeChanged.Width)
            lineToBeChanged.Width = newWidth;
        if (newWidth + 2 * margin > this.documentSize.Width)
        {
            documentSize.Width = (int)newWidth;
            AutoScrollMinSize = this.documentSize;
        }
        Rectangle changedRectangle = new Rectangle(
            LineIndexToPageCoordinates(i),
            new Size((int)newWidth,
                (int)this.lineHeight));

        Invalidate(changedRectangle);
    }
    base.OnDoubleClick(e);
}
```

代码下载 CapsEditor.sln

首先调用 `PageCoordinatesToLineIndex()` 方法计算在用户双击时光标悬停在哪行文本上。如果该调用返回-1,则光标没有悬停在任何文本行上,因此什么也不做。当然,调用 `OnDoubleClick()` 方法的基类版本,可以让 Windows 执行一些默认操作。

假定已经标识一行文本，那么可以使用 `string.ToUpper()` 方法把它转换为大写。这很容易实现。比较难的部分是确定要在什么地方重绘什么内容。幸运的是，因为这个示例非常简单，所以没有太多的组合。可以假定把文本转换为大写通常要么保持这行文本的宽度不变，要么增大其宽度。因为大写字母比小写字母大，所以宽度不会减小。因为本例没有换行功能，所以文本行不会溢出到下一行上，并使其他文本向下移动。把文本行转换为大写的操作不会改变正在显示的任何其他项的位置，这是一个非常大的简化！

接着代码使用 `Graphics.MeasureString()` 方法计算文本的新宽度。这有两种可能性：

- 新宽度会使该行最长，导致整个文档的宽度增加。如果情况是这样，就需要把 `AutoScrollMinSize` 设置为新值，以便正确地放置滚动条。
- 文档的大小没有变化。

在这两种情况下，都需要调用 `Invalidate()` 方法重绘屏幕。因为只有一行文本改变了，所以不希望重绘整个文档。而需要计算出仅包含被修改的文本行的矩形边界，并把这个矩形传递给 `Invalidate()` 方法，确保只重绘该行文本。这就是以前代码的作用。`Invalidate()` 方法会在鼠标事件处理程序最终返回时调用 `OnPaint()` 方法。本章前面曾提到在 `OnPaint()` 方法中设置断点的困难，如果运行该示例，在 `OnPaint()` 方法中设置断点，捕获绘图操作，就会发现传递给 `OnPaint()` 方法的 `PaintEventArgs` 参数包含一个与指定矩形匹配的剪切区域。因为重载了 `OnPaint()` 方法来仔细考虑剪切区域，所以只重绘要求的文本行。

48.16 打印

本章前面主要介绍如何在屏幕上绘图。但有时还需要应用程序生成数据的打印件。这就是本节的主题。我们将扩展 `CapsEditor` 示例，以便它能进行打印预览，并打印正在编辑的文档。

但是，因为本书没有详细讲述这个过程，所以这里实现的打印功能非常基本。通常，如果要实现应用程序的打印数据功能，就要在主 `File` 菜单中为应用程序添加 3 个菜单项：

- **Page Setup**，允许用户选择各种选项，例如，要打印哪一页，要使用什么打印机等。
- **Print Preview**，打开一个新窗体，新窗体显示打印副本的预览外观。
- **Print**，实际打印文档。

在本例中，为了简单起见，不实现 `Page Setup` 菜单项。打印也只使用默认的设置。但要注意，如果要实现 `Page Setup`，Microsoft 已经编写了一个页面设置对话框类，以供使用。这个类是 `System.Windows.Forms.PrintDialog`。一般应编写一个事件处理程序，该事件处理程序显示这个窗体，并保存用户选择的设置。

在许多情况下，打印与在屏幕上显示完全相同：提供一个设备环境(`Graphics` 实例)，对该实例调用所有常见的显示命令。Microsoft 编写了许多类以帮助完成这个工作，我们需要的两个主要的类是 `System.Drawing.Printing.PrintDocument` 和 `System.Drawing.Printing.PrintPreviewDialog`。这两个类可以确保传递给设备环境的绘图命令能得到正确地处理，我们不必担心什么内容应打印到何处的问题。

打印/打印预览和显示在屏幕上有一些重要的区别。打印机不能滚动，但它们输出页面。所以需要确保找到一种合适的方式给文档分页，按要求绘制每一页。其他工作还有计算文档有多少内容可以放在一个页面上，因此计算出需要多少页，文档的每个部分应写到哪个页面上。

尽管上面描述得相当复杂，但打印过程相当简单。从编程的角度来看，需要采取的步骤大致如下：

- **打印**——实例化一个 `PrintDocument` 对象，并调用其 `Print()` 方法。这个方法向 `PrintPage` 事件发出打印第一个页面的信号。`PrintPage` 事件接受一个 `PrintPageEventArgs` 参数，该参数提供页面大小和页面设置的信息，以及用于绘图命令的 `Graphics` 对象。因此应为这个事件编写一个事件处理程序，并实现该处理程序，以打印页面。这个事件处理程序还应把 `PrintPageEventArgs` 参数的布尔属性 `HasMorePages` 设置为 `true` 或 `false`，表示是否还有要打印的页面。`PrintDocument.Print()` 方法将重复引发 `PrintPage` 事件，直到把 `HasMorePages` 属性设置为 `false` 为止。
- **打印预览**——在这种情况下，实例化 `PrintDocument` 对象和 `PrintPreviewDialog` 对象。使用 `PrintPreviewDialog.Document` 属性把 `PrintDocument` 关联到 `PrintPreviewDialog` 上，然后调用对话框的 `ShowDialog()` 方法。这个方法会模拟地显示对话框，使之呈现为 Windows 标准打印预览窗体，打印预览窗体显示文档的页面。在内部，则重复引发 `PrintPage` 事件，多次显示页面，直到把 `HasMorePages` 属性为 `false` 为止。不需为此编写一个事件处理程序；而可以使用打印每个页面时使用的同一个事件处理程序，因为绘图代码在这两种情况下应完全相同(毕竟，打印预览的内容应与打印出来的版本看起来完全相同)。

实现打印和打印预览

既然简要概述了该过程，本节介绍如何在代码中完成这些步骤。可以从 www.wrox.com 或随书附赠光盘上找到 `PrintingCapsEdit` 项目，它包含 `CapsEditor` 项目，以及下述修改。

首先使用 Visual Studio 2010 设计视图在 `File` 菜单中添加两个新菜单项：`Print` 和 `Print Preview`。再使用 `Properties` 窗口把这两个项命名为 `menuFilePrint` 和 `menuFilePrintPreview`，把它们设置为应用程序启动时是禁用的(只有打开文档，才能进行打印)。在主窗体的 `LoadFile()` 方法中添加如下代码，启用这两个菜单项，`LoadFile()` 方法负责把文件加载到 `CapsEditor` 应用程序中：



```
private void LoadFile(string FileName)
{
    StreamReader sr = new StreamReader(FileName);
    string nextLine;
    documentLines.Clear();
    nLines = 0;
    TextLineInformation nextLineInfo;
    while ( (nextLine = sr.ReadLine()) != null)
    {
        nextLineInfo = new TextLineInformation();
        nextLineInfo.Text = nextLine;
        documentLines.Add(nextLineInfo);
        ++nLines;
    }
    sr.Close();
    if (nLines > 0)
    {
        documentHasData = true;
        menuFilePrint.Enabled = true;
        menuFilePrintPreview.Enabled = true;
    }
    else
    {
        documentHasData = false;
        menuFilePrint.Enabled = false;
        menuFilePrintPreview.Enabled = false;
    }
}
```

```

    }
    CalculateLineWidths();
    CalculateDocumentSize();
    Text = standardTitle + "-" + FileName;
    Invalidate();
}

```

代码下载 [Printing.sln](#)

上面突出显示的代码是添加到这个方法中的新代码。接着，给 Form1 类添加一个成员字段：

```

public partial class Form1: Form
{
    private int pagesPrinted = 0;
}

```

这个字段用于指定目前正在打印的页面。使它成为一个成员字段，因为需要在 PrintPage 事件处理程序的调用之间记忆上述信息。

接着是用户选择 Print 或 Print Preview 菜单项时执行的事件处理程序：



```

private void menuFilePrintPreview_Click(object sender, System.EventArgs e)
{
    this.pagesPrinted = 0;
    PrintPreviewDialog ppd = new PrintPreviewDialog();
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += this.pd_PrintPage;
    ppd.Document = pd;
    ppd.ShowDialog();
}
private void menuFilePrint_Click(object sender, System.EventArgs e)
{
    this.pagesPrinted = 0;
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += new PrintPageEventHandler
        (this.pd_PrintPage);
    pd.Print();
}

```

代码下载 [Printing.sln](#)

前面解释了涉及打印的主要过程，可以看出，这些事件处理程序仅实现了这个过程。在打印和打印预览中，都实例化一个 PrintDocument 对象，并把一个事件处理程序附加到该对象的 PrintPage 事件中。对于打印，应调用 PrintDocument.Print()方法；而对于打印预览，则把 PrintDocument 对象附加到 PrintPreviewDialog 中，并调用打印预览对话框对象的 ShowDialog()方法。实际工作将在 PrintPage 事件的处理程序中完成。下面是该处理程序的代码：



```

private void pd_PrintPage(object sender, PrintPageEventArgs e)
{
    float yPos = 0;
    float leftMargin = e.MarginBounds.Left;
    float topMargin = e.MarginBounds.Top;
    string line = null;
    // Calculate the number of lines per page.
    int linesPerPage = (int)(e.MarginBounds.Height /
        mainFont.GetHeight(e.Graphics));
}

```

```

int lineNo = pagesPrinted * linesPerPage;
// Print each line of the file.
int count = 0;
while(count < linesPerPage && lineNo < this.nLines)
{
    line = ((TextLineInformation)this.documentLines[lineNo]).Text;
    yPos = topMargin + (count * mainFont.GetHeight(e.Graphics));
    e.Graphics.DrawString(line, mainFont, Brushes.Blue,
        leftMargin, yPos, new StringFormat());
    lineNo++;
    count++;
}
// If more lines exist, print another page.
if(this.nLines > lineNo)
    e.HasMorePages = true;
else
    e.HasMorePages = false;
pagesPrinted++;
}

```

代码下载 [Printing.sln](#)

在声明了两个局部变量后，应先计算出一个页面上可以显示多少行文本——即页面的高度除以一文本行的高度，并向下舍入。页面的高度可以从 `PrintPageEventArgs.MarginBounds` 属性获得，这个属性是一个 `RectangleF` 结构，初始化该结构来给出页面的边界。一行文本的高度可以从 `Form1.mainFont` 字段中获得，该字段是显示文本所使用的字体。这里也必须使用相同的字体进行打印。注意，对于 `PrintingCapsEditor` 示例，因为每页的行数总相同，所以可以在第一次计算出该行数后，把它缓存起来。但是，该计算并不困难，在比较复杂的应用程序中，因为这个值可能会有变化，所以每次打印页面时重新计算它也是可以的。

我们还初始化一个 `lineNo` 变量。这给出页面第 1 行对应的文档的文本行索引(该索引从 0 开始)。这条信息非常重要，因为在原则上，本可以调用 `pd_PrintPage()` 方法打印任何页面，而不仅仅是打印第一页。`lineNo` 的值是每个页面的行数和已打印的页数的乘积。

接着运行一个循环，打印每一行文本。当打印完文档中的所有文本行，或者打印完本页面上的所有文本行时，这个循环就终止(只要满足这两个条件之一即可)。最后，检查是否还有要打印的文档，并据此设置 `PrintPageEventArgs` 参数的 `HasMorePages` 属性。同时递增 `pagesPrinted` 字段，这样下一次调用 `PrintPage` 事件处理程序时，就会打印正确的页面。

这个事件处理程序要注意的一点是不必担心绘图命令的发送目的地。我们使用所提供的 `Graphics` 对象和 `PrintPageEventArgs` 参数。Microsoft 编写的 `PrintDocument` 类将在内部确保，如果正在打印，`Graphics` 对象就与打印机关联起来；如果正在打印预览，`Graphics` 对象就与屏幕上的打印预览窗体关联起来。

最后，需要确保为类型定义搜索 `System.Drawing.Printing` 名称空间：

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Printing;
using System.Text;

```

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

剩下的内容全部是编译项目，并检查代码的工作情况。如果运行 CapsEdit，加载一个文本文档(与以前一样，为该项目选择 C#源文件)，并选择 Print Preview，就可以显示出该文档的打印预览版本，如图 48-19 所示。

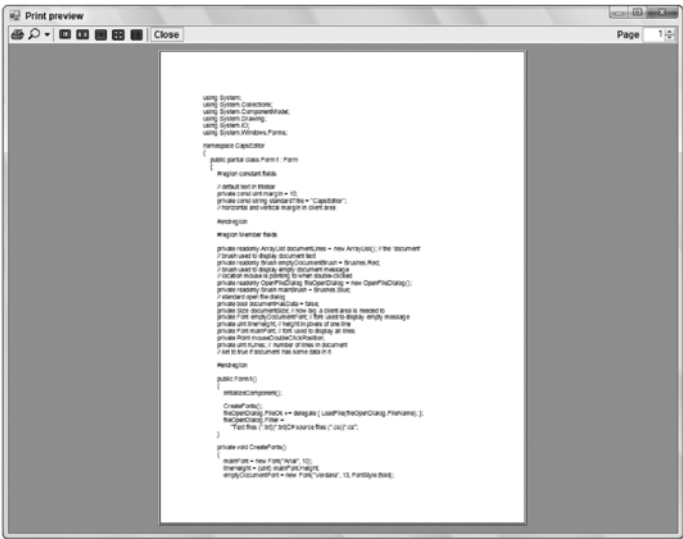


图 48-19

在图 48-19 上，滚动到文档的第 5 页上，把预览设置为显示正常的尺寸。PrintPreviewDialog 提供了许多功能，这可以从窗体顶部的工具栏中看出来。可用的选项包括打印文档、缩小和放大文档、一次显示 2、3、4 或 6 页。这些选项的功能都很完整，不需要我们做任何工作。例如，如果把缩放改为自动，单击显示 4 页的选项(右数第 3 个工具栏)，就会得到如图 48-20 所示的结果。

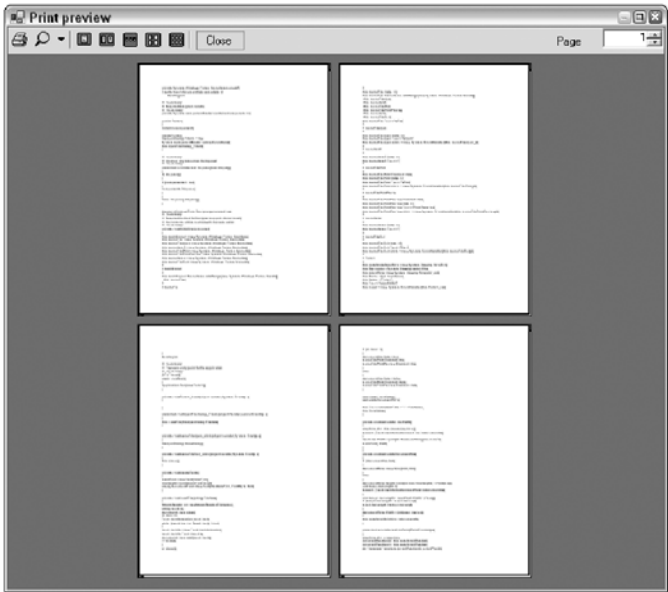


图 48-20

48.17 小结

本章介绍了如何在显示设备上绘图操作，其中绘图操作通过代码实现，而不是由一些预定义的控件或对话框实现，这就是 GDI+ 的实质。GDI+ 是一个功能强大的工具，有许多 .NET 基类都可用于在设备上绘图。绘图过程实际上相对简单。在大多数情况下，只使用几条 C# 语句，就可以绘制文本和专业的图形或显示图像。但是，管理绘图操作——后台的工作涉及计算出要绘制的内容、确定在什么地方绘制它，以及在任何给定情况下，哪些内容需要重绘，哪些内容不需要重绘。这些工作都非常复杂，需要经过仔细的算法设计。因此，很好地理解 GDI+ 的工作方式，以及 Windows 采取什么操作来完成工作也非常重要。特别是，由于 Windows 的体系结构，在绘图过程中，应使窗口的区域失效，并依赖 Windows 通过引发 Paint 事件来响应。

本章并没有介绍绘图操作所涉及的其他 .NET 类。然而，如果您理解了绘图操作的规则，就可以通过查看 SDK 文档中这些类的方法列表，实例化它们的实例，看看它们能做什么，以掌握它们。最后，如果要超越标准控件的功能，那么绘图与编程的任何其他方面一样，也需要逻辑、仔细的思考和清晰的算法。如果软件设计得好，它就有助于改善用户友好性和视觉外观。许多应用程序完全依赖于控件来设计其用户界面。这很有效，但这种应用程序看起来非常类似。通过添加一些 GDI+ 代码完成某些自定义绘图操作，可以使软件与众不同，显得比较新颖——这只会有助于软件的销售！

第 49 章

VSTO

本章内容:

- 可以用 VSTO 创建的项目类型, 在这些项目中可以包含的功能
- 应用于所有 VSTO 解决方案类型的基础技术
- 使用宿主项和宿主控件
- 构建带自定义 UI 的 VSTO 解决方案

Visual Studio Tools for Office(VSTO)技术可以使用 .NET Framework 定制和扩展 Microsoft Office 应用程序和文档。它包含的工具还可以使在 Visual Studio 中简化这个自定义过程, 例如, 用于 Office Ribbon 控件的可视化设计器。

VSTO 是 Microsoft 发布的一系列产品中的最新产品, 可以定制 Office 应用程序。用于访问 Office 应用程序的对象模型随时间逐步演化。如果读者过去曾使用过它, 就会熟悉它的某些部分。如果读者以前为 Office 应用程序编写过 VBA 插件, 就为本章讨论的技术做好了准备。但 VSTO 提供的、便于与 Office 交互的类已经扩展到 Office 对象模型之外。例如, VSTO 类包括 .NET 数据绑定功能。

在 Visual Studio 2008 推出之前, VSTO 一直是一个独立下载的软件包, 如果要开发 Office 解决方案, 就可以得到它。从 Visual Studio 2008 开始, VSTO 集成到 Visual Studio IDE 中。VSTO 的这个版本也称为 VSTO 3, 包含对 Office 2007 的全部支持, 还包括许多新功能。例如, 可以与 Word 内容控件交互, 前面提及的 ribbon 可视化设计器等。

在 Visual Studio 2010 中, VSTO 4 扩展并改进了以前的版本, 它更便于部署, 且不再需要在客户端 PC 上安装主互操作程序集(Primary Interop Assembly, PIA), 这通过 CLR 4 类型嵌入功能实现。还包含对 Office 2010 的支持。

本章不需要 VSTO 或其以前版本的任何预备知识。

49.1 VSTO 概述

VSTO 包含如下组件:

- 一组项目模板, 可用于创建各种类型的 Office 解决方案
- 设计器, 支持 ribbons、动作面板和自定义任务面板的可视化布局
- 建立在 Office 对象模型基础之上的类, 它们还提供扩展功能

VSTO 支持 Office 2003、2007 和 2010。VSTO 类库有多种形式，分别用于这几种 Office 版本，它们分别使用不同集的程序集。为了简单起见，本章主要介绍 2007 版。
VSTO 解决方案的一般体系结构如图 49-1 所示。

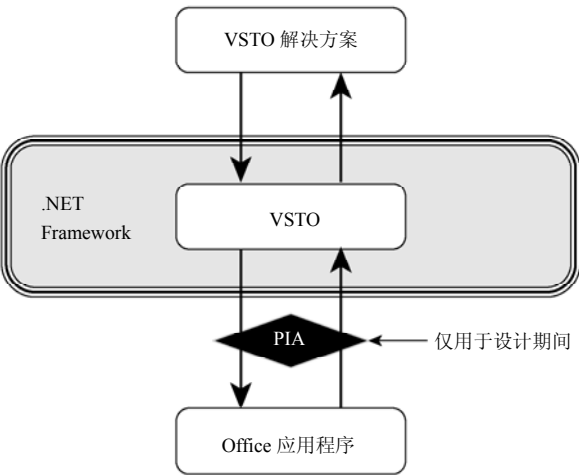


图 49-1

注意，当面向 .NET 4 时，在设计期间仅需要如图 49-1 所示的 PIA，当部署到客户端时这用作内嵌的类型。这对于面向 .NET 3.5 的 VSTO 解决方案不成立，此时在开发计算机和目标计算机上都需要 PIA。

49.1.1 项目类型

图 49-2 显示了 VS for Office 2007 中的项目模板(Office 2010 中可用的列表与此类似)。本章主要讨论 Office 2007。

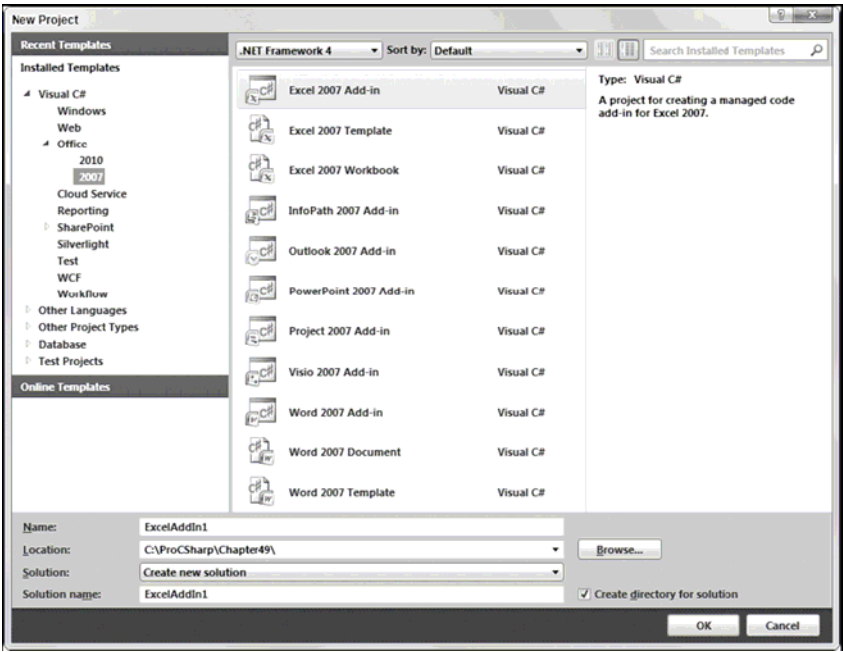


图 49-2

VSTO 项目模板可以分为如下类别：

- 文档级自定义
- 应用程序级的插件
- SharePoint 工作流模板(在图 49-2 中未显示，因为它们位于有关单独的模板类别中)



注意，VSTO 的以前版本包含第 4 类：InfoPath 窗体模板，它在 VSTO 4 中不再可用。

本章主要讨论最常用的项目类型，即文档级自定义和应用程序级的插件。

1. 文档级自定义

创建这种类型的项目时，会生成一个链接到单个文档上的程序集，如 Word 文档、Word 模板或 Excel 工作簿。加载该文档时，关联的 Office 应用程序会检测到该自定义，加载程序集，使 VSTO 定制可用。

这类项目可以给某个特定业务范围的文档提供附加功能，或者在文档模板中添加自定义功能，为整类文档添加附加功能。所包含的代码可以操作文档和文档的内容，包括嵌入对象。还可以提供自定义菜单，包括可以用 Visual Studio Ribbon 设计器创建的功能区菜单。

创建文档级的项目时，可以选择创建新文档，或者复制已有的文档，作为开发的起点。也可以选择要创建的文档类型。例如，对于 Word 文档，就可以选择创建.docx(默认)、.doc 或.docm 文档(.docm 是启用宏的文档)。其对话框如图 49-3 所示。

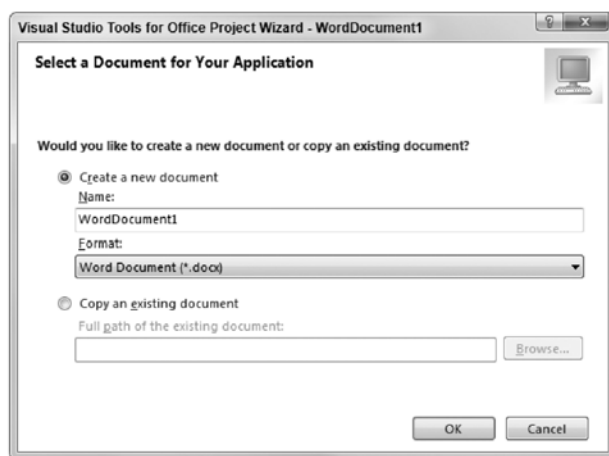


图 49-3

2. 应用程序级的插件

应用程序级的插件不同于文档级的自定义，因为前者可用于整个目标 Office 应用程序。我们可以访问插件代码，而无论加载什么文档，其中都可能包含菜单、文档操作等。

启动某个 Office 应用程序(如 Word)时，它会寻找已在注册表中有注册项的关联插件，并加载它需要的任何程序集。

3. SharePoint 工作流模板

这些项目提供了创建 SharePoint 工作流应用程序的模板。它们用于管理 SharePoint 进程中的文档流。创建这类项目后，就可以在文档的生命周期中，在重要的时刻执行自定义代码。

49.1.2 项目功能

在各种 VSTO 项目类型中有几个可以使用的功能，如交互面板和控件。我们使用的项目类型决定了可用的功能。表 49-1~49-3 根据项目类型列出了这些功能。

表 49-1

功 能	说 明
动作面板	动作面板是驻留在 Word 或 Excel 的动作面板中的对话框。可以在这里显示任意控件，这是扩展文档和应用程序的一种通用方式
数据缓存	数据的缓存可以在文档外部的缓存数据岛上存储在文档中使用的数据。这些数据岛可以从数据源中更新或手工更新，在数据源脱机或不可用时，允许 Office 文档访问数据
VBA 代码的端点	如前所述，VSTO 支持与 VBA 的交互操作。在文档级的自定义中，可以提供从 VBA 代码中调用的端点方法
宿主控件	宿主控件是 Office 对象模型中已有控件的扩展包装器。可以操作这些对象，并把数据绑定到这些对象上
智能标记	智能标记是嵌入在 Office 文档中、有类型化内容的对象。它们在 Office 文档的内容中自动检测，例如，应用程序检测到相应的文本时，就会自动添加股票报价智能标记。可以创建自己的智能标记类型，定义可以在该类标记上执行的操作
可视化文档设计器	处理文档自定义项目时，要使用 Office 对象模型创建一个可视化设计界面，用于交互式地排放控件。设计器中显示的工具栏和菜单(如本章后面所述)功能齐全

表 49-2 列出了应用程序级的插件功能

表 49-2

功 能	说 明
自定义任务面板	任务面板一般停靠在 Office 应用程序的一个边界上，并提供各种功能。例如，Word 的一个任务面板用于操作样式。与动作面板一样，它们也提供了很大的灵活性
跨应用程序的通信	一旦为某个 Office 应用程序创建了插件，就可以把这个功能提供给其他插件。例如，可以在 Excel 中创建一个财务计算服务，再从 Word 中使用该服务——无需创建一个单独的插件
Outlook 窗体区域	可以创建在 Outlook 中使用的窗体区域

表 49-3 列出了所有项目类型可用的功能

表 49-3

功 能	说 明
ClickOnce 部署	可以通过 ClickOnce 部署方法把自己创建的任意 VSTO 项目发布给最终用户，让用户检测对应用程序集清的变化，拥有文档级和应用程序级解决方案的最新版本
功能区菜单	功能区菜单在所有的 Office 应用程序中使用。VSTO 提供了创建自定义功能区菜单的两种方式。可以使用 XML 定义功能区，也可以使用功能区设计器，后者更容易使用，尽管采用 XML 版本可以保证向后兼容性

49.2 VSTO 项目基础

既然知道了 VSTO 包含的内容, 就该查看 VSTO 更实用的一面了, 并学习如何构建 VSTO 项目。本节介绍的技术可以应用于所有 VSTO 项目类型。

本节介绍如下内容:

- Office 对象模型
- VSTO 名称空间
- 宿主项和宿主控件
- 基本 VSTO 项目结构
- Globals 类
- 事件处理

49.2.1 Office 对象模型

Office 应用程序的 2007 和 2010 套件通过一个 COM 对象模型提供其功能。可以从 VBA 中直接使用这个对象模型, 来控制 Office 功能的任意方面。Office 对象模型在 Office 97 中引入, 之后有了许多演变, Office 中的功能也有许多改变。

Office 对象模型有大量类, 其中一些类在 Office 应用程序的套件中使用, 一些类专门用于个别应用程序。例如, Word 2007 对象模型包含一个 Documents 集合, 它表示当前加载的对象, 每个对象都用一个 Document 对象表示。在 VBA 代码中, 可以根据名称或索引访问文档, 调用方法对它们执行操作。例如, 下面的 VBA 代码关闭名称为 My Document 的文档, 且不保存修改的内容:

```
Documents("My Document").Close SaveChanges:= wdDoNotSaveChanges
```

Office 对象模型包含命名常量(如上述代码中的 wdDoNotSaveChanges)和枚举, 更便于使用。

49.2.2 VSTO 名称空间

VSTO 包含一个名称空间集合, 该集合包含的类型可用于给 Office 对象模型编写程序。这些名称空间中的许多类型直接映射到 Office 对象模型中的类型上。可以通过 Office PIA 在设计期间访问它们, 在部署解决方案时则通过嵌入的类型信息来访问。由于嵌入的类型, 因此主要使用用于访问 Office 对象模型的接口。VSTO 还包含不能直接映射的类型, 或者与 Office 对象模型无关的类型。例如, 有许多类用于 Visual Studio 中支持的设计器。

包装 Office 对象模型中的对象或与它们通信的类型分别放在不同的名称空间中。用于 Office 开发的名称空间如表 49-4 所示。

表 49-2

名 称 空 间	说 明
Microsoft.Office.Core Microsoft.Office.Interop.*	因为这些名称空间包含 Office 对象模型的接口和瘦包装器, 所以提供了处理 Office 类的基本功能。在 Microsoft.Office.Interop 名称空间中有几个嵌套的名称空间, 用于每个 Office 产品
Microsoft.Office.Tools	这个名称空间包含的通用类型提供了 VSTO 功能和用于嵌套名称空间中的许多类的基类。例如, 这个名称空间包含实现文档级自定义中的动作面板所需的类, 以及应用程序级插件的基类

(续表)

名 称 空 间	说 明
Microsoft.Office.Tools.Excel Microsoft.Office.Tools.Excel.*	这些名称空间包含的类型用于与 Excel 应用程序和 Excel 文档交互
Microsoft.Office.Tools.Outlook	这个名称空间包含的类型用于与 Outlook 应用程序交互
Microsoft.Office.Tools.Ribbon	这个名称空间包含的类型用于处理和创建功能区菜单
Microsoft.Office.Tools.Word Microsoft.Office.Tools.Word.*	这些名称空间包含的类型用于与 Word 应用程序和 Word 文档交互
Microsoft.VisualStudio.Tools.*	这些名称空间提供的 VSTO 基础结构可以在 Visual Studio 中开发 VSTO 解决方案时使用

49.2.3 宿主项和宿主控件

宿主项和宿主控件是经过扩展的接口，使文档级的自定义更容易与 Office 文档交互。这些接口简化了代码，因为它们提供了 .NET 样式的事件，且进行了全面地管理。宿主项和宿主控件中的“宿主”表示，这些接口封装和扩展了本地 Office 对象。

在使用宿主项和宿主控件时，也需要使用底层的交互操作类型。例如，如果创建了一个新的 Word 文档，就会接收到对交互操作 Word 文档类型的引用，而不是 Word 文档宿主项。必须注意这一点，并据此编写代码。

Word 和 Excel 文档级自定义都有宿主项和宿主控件。

1. Word

Word 只有一个宿主项 Microsoft.Office.Tools.Word.Document。这表示一个 Word 文档。果然，这个接口有许多方法和属性，可用于与 Word 文档交互。

Word 有 12 个宿主控件，如表 49-5 所示，所有宿主控件都在 Microsoft.Office.Tools.Word 名称空间中。

表 49-5

控 件	说 明
Bookmark	这个控件表示 Word 文档中的一个位置，它可以是单个位置，或一个字符范围
XMLNode、XmlNodes	当文档有一个附加的 XML 架构时使用这两个控件，它们允许通过文档内容的 XML 节点位置来引用文档内容。也可以用这两个控件操作文档的 XML 结构
ContentControl	这个接口与本表中剩余 8 个控件有相同的基接口(ContentControlBase)，允许处理 Word 内容控件。内容控件把内容表示为控件，或者启用文档中纯文本没有提供的功能
BuildingBlockGallery-ContentControl	这个控件允许添加和处理文档构建模块，如格式化的表、封面等
ComboBoxContentControl	这个控件表示格式化为组合框的内容
DatePickerContentControl	这个控件表示格式化为日期拾取器的内容
DropDownListContentControl	这个控件表示格式化为下拉列表的内容

(续表)

控 件	说 明
GroupContentControl	这个控件表示的内容是其他内容项的分组集合，包括文本和其他内容控件
PictureContentControl	这个控件表示一幅图像
RickTextContentControl	这个控件表示一块格式文本内容
PlainTextContentControl	这个控件表示一块纯文本内容

2. Excel

Excel 有 3 个宿主项和 4 个宿主控件，它们都包含在 Microsoft.Office.Tools.Excel 名称空间中。Excel 宿主项如表 49-6 所示。

表 49-6

主 机 项	说 明
Workbook	这个宿主项表示整个 Excel 工作簿，它可以包含多个工作表和图表
Worksheet	这个宿主项用于工作簿中的单个工作表
Chartsheet	这个宿主项用于工作簿中的单个图表

Excel 宿主控件如表 49-7 所示。

表 49-7

控 件	说 明
Chart	这个控件表示嵌入到工作表中的图表
ListObject	这个控件表示工作表中的一个列表
NamedRange	这个控件表示工作表中的一个命名区域
XmlMappedRange	当 Excel 电子表格有附加的架构时使用这个控件，它用于处理映射到 XML 架构元素上的范围

49.2.4 基本的 VSTO 项目结构

第一次创建 VSTO 项目时，系统创建的文件随项目类型的不同而不同，但有一些共同的功能。本节介绍 VSTO 项目的组成。

1. 文档级自定义项目结构

创建文档级自定义项目时，在 Solution Explorer 窗口中有一项表示文档类型。它可以是：

- 表示 Word 文档的.docx 文件
- 表示 Word 模板的.dotx 文件
- 表示 Excel 工作簿的.xlsx 文件
- 表示 Excel 模板的.xltx 文件

每个文档类型都有一个设计器视图和一个代码文件，如果在 Solution Explorer 窗口中展开该项，就会看到它们。Excel 模板还包含子项，它们表示整个工作簿和工作簿中的每个工作表。这个结构可以在每个工作表或工作簿的基础上提供自定义功能。

如果查看上述项目类型的隐藏文件，就会看到几个设计器文件，查看这些设计器文件，还会看到模板生成的代码。每个 Office 文档项都在 VSTO 名称空间中有关联的类，代码文件中的类派生自这些类。这些类定义为部分类，这样自定义代码会与可视化设计器生成的代码分隔开，类似于 Windows 窗体应用程序的结构。

例如，Word 文档模板提供了一个派生自 `Microsoft.Office.Tools.Word.Document` 宿主项的类。这个类通过 `Base` 属性提供 `Document` 宿主项，这个类包含在 `ThisDocument.cs` 中，如下所示：

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Xml.Linq;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Office = Microsoft.Office.Core;
using Word = Microsoft.Office.Interop.Word;

namespace WordDocument1
{
    public partial class ThisDocument
    {
        private void ThisDocument_Startup(object sender, System.EventArgs e)
        {
        }

        private void ThisDocument_Shutdown(object sender, System.EventArgs e)
        {
        }
        #region VSTO Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InternalStartup()
        {
            this.Startup += new System.EventHandler(ThisDocument_Startup);
            this.Shutdown += new System.EventHandler(ThisDocument_Shutdown);
        }
        #endregion
    }
}
```

这些模板生成的代码包含两个主要名称空间的别名，在为 Word 创建文档级自定义时，需要使用这两个名称空间。`Microsoft.Office.Core` 用于主要的 VSTO Office 类，`Microsoft.Office.Interop.Word` 用于 Word 专用的类。注意如果要使用 Word 宿主控件，那么还要为 `Microsoft.Office.Tools.Word` 名称空间添加一条 `using` 语句。模板生成的代码还定义两个事件处理程序挂钩——`ThisDocument_Startup()` 和 `ThisDocument_Shutdown()`，用于在加载或卸载文档时执行代码。

每个文档级自定义项目类型的代码文件(或者，对于 Excel 文件或代码文件)都有类似的结构，还定义了名称空间别名以及 VSTO 类中各个 `Startup` 和 `Shutdown` 事件的处理程序。以此为起点，可以添加对话框、动作面板、ribbon 控件、事件处理程序和自定义代码，来定义自定义行为。

在文档级自定义中，还可以通过文档设计器定制文档。根据所创建的解决方案类型，这可能需要给模板添加样板文件，给文档添加交互式内容或其他内容。设计器实际上是 Office 应用程序的宿主版本，使用它们可以像在应用程序中那样输入内容。然而，还可以在文档中添加控件，如宿主控件和 Windows 窗体控件，以及这些控件的代码。

2. 应用程序级插件的项目结构

当创建应用程序级插件时，在 Solution Explorer 窗口中没有文档。而有一项表示创建插件所使用的应用程序，如果展开该项，会看到一个 ThisAddIn.cs 文件。这个文件包含 ThisAddIn 类的一个部分类定义，该类是插件的入口点。这个类派生自 Microsoft.Office.Tools.AddIn Base，它提供了实现插件功能的代码。

例如，Word 插件模板生成的代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
using Microsoft.Office.Tools.Word;
namespace WordAddIn1
{
    public partial class ThisAddIn
    {
        private void ThisAddIn_Startup(object sender, System.EventArgs e)
        {
        }
        private void ThisAddIn_Shutdown(object sender, System.EventArgs e)
        {
        }
        #region VSTO generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InternalStartup()
        {
            this.Startup += new System.EventHandler(ThisAddIn_Startup);
            this.Shutdown += new System.EventHandler(ThisAddIn_Shutdown);
        }
        #endregion
    }
}
```

可以看出，这个结构非常类似于文档级自定义使用的结构。它包含相同 Microsoft.Office.Core 和 Microsoft.Office.Interop.Word 名称空间的别名，并提供 Startup 和 Shutdown 事件的事件处理程序 (ThisAddIn_Startup() 和 ThisAddIn_Shutdown())。这些事件与文档级自定义略有不同，因为它们在加载或卸载插件时引发，而不是在打开或关闭个别文档时引发。

定制应用程序级插件与文档级自定义相同：添加 ribbon 控件、任务面板和其他代码。

49.2.5 Globals 类

所有 VSTO 项目类型都定义一个 Globals 类，它提供了如下内容的全局访问权限：

- 对于文档级自定义，可以访问解决方案中的所有文档。这通过其名称匹配文档类名的成员来提供，如 Globals.ThisWorkbook 和 Globals.Sheet1。
- 对于应用程序级的插件，可以访问插件对象。这通过 Globals.ThisAddIn 提供。
- 对于 Outlook 插件项目，可以访问所有 Outlook 窗体区域。
- 通过 Globals.Ribbons 属性，可以访问解决方案中的所有 ribbon 控件。
- 派生自 Microsoft.Office.Tools.Factory 的接口提供项目类型专用的实用程序方法，通过 Factory 属性可访问这些方法。

在后台，在解决方案中由设计器维护的各种代码文件中通过一系列部分定义来创建 Globals 类。例如，在 Excel 工作簿项目中，默认的 Sheet1 工作表包含设计器生成的如下代码：

```
internal sealed partial class Globals
{
    private static Sheet1 _Sheet1;
    internal static Sheet1 Sheet1
    {
        get
        {
            return _Sheet1;
        }
        set
        {
            if ((_Sheet1 == null))
            {
                _Sheet1 = value;
            }
            else
            {
                throw new System.NotSupportedException();
            }
        }
    }
}
```

这段代码把 Sheet1 成员添加到 Globals 类中。

Globals 类还允许使用 Globals.Factory.GetVstoObject() 方法把交互操作类型转换为 VSTO 类型。例如，这会从 Microsoft.Office.Interop.Excel.Workbook 接口中得到 VSTO Workbook 接口。另外，Globals.Factory.HasVstoObject() 方法可用于确定这种转换是否可行。

49.2.6 事件处理

本章前面介绍了宿主项和宿主控件类如何提供我们可以处理的事件。但交互操作类不是这样。我们只能使用几个事件，大多数事件都很难用于创建事件驱动的解决方案。为了响应事件，我们常常要关注宿主项和宿主控件提供的事件。

此处一个明显的问题是应用程序级的插件项目没有宿主项和宿主控件。在使用 VSTO 时，必须面对这个问题。但是，我们在插件中侦听的大多数常用事件都关联到功能区菜单和任务面板的交互操作

中。我们用集成的功能区设计器设计功能区控件，响应功能区控件生成的事件，使控件可以交互操作。任务面板常常作为 Windows 窗体用户控件实现(尽管也可以使用 WPF)，这里可以使用 Windows 窗体事件。这表示，我们不会常常遇到如下情形：需要的功能没有可用的事件。

当需要使用 Office 交互操作对象提供的事件时，通过这些对象上的接口提供这些事件。考虑一个 Word 插件项目。这个项目中的 ThisAddIn 类有一个 Application 属性，利用该属性可以获得对 Office 应用程序的引用。这个属性的类型是 Microsoft.Office.Interop.Word.Application，它通过 Microsoft.Office.Interop.Word.ApplicationEvents4_Event 接口提供事件。这个接口共提供 29 个事件(对于像 Word 这样复杂的应用程序 29 个事件并不多)。例如，我们可以处理 DocumentBeforeClose 事件，来响应 Word 文档的关闭请求。

49.3 构建 VSTO 解决方案

前面几节解释了 VSTO 项目的概念、结构和可以在各种项目类型中使用的功能。本节讨论如何实现 VSTO 解决方案。

图 49-4 列出了文档级自定义解决方案的结构。

对于文档级自定义，至少要与一个宿主项交互操作，该宿主项一般包含多个宿主控件。可以直接使用 Office 对象包装器，但在大多数情况下，应通过宿主项和宿主控件访问 Office 对象模型及其功能。

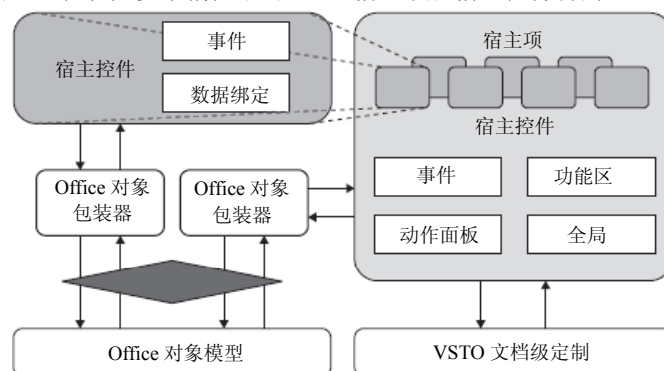


图 49-4

可以在代码中使用宿主项和宿主控件事件、数据绑定、功能区菜单、动作面板和全局对象。

图 49-5 列出了应用程序级插件解决方案的结构。

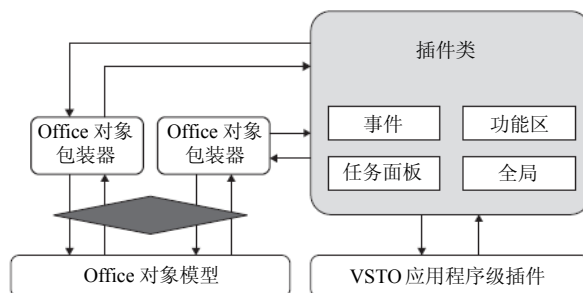


图 49-5

在这个略微简单的模型中，很可能要直接使用 Office 对象的瘦包装器，或者至少通过封装解决方案的插件类来使用。还要在代码中使用插件类提供的事件、功能区菜单、动作面板和全局对象。

本节介绍这两种应用程序类型以及如下主题：

- 管理应用程序级插件
- 与应用程序和文档交互操作
- UI 自定义

49.3.1 管理应用程序级插件

在创建应用程序级插件时，会发现 Visual Studio 执行了通过 Office 应用程序注册插件所需的所有步骤。这表示，添加注册表项，这样在 Office 应用程序启动时，它会自动定位和加载程序集。如果以后要添加或删除插件，就必须导航 Office 应用程序设置，或者手工操作注册表。

例如，在 Word 中，必须打开 Office Button 菜单，单击 Word Options，选择 Add-Ins 选项卡，如图 49-6 所示。

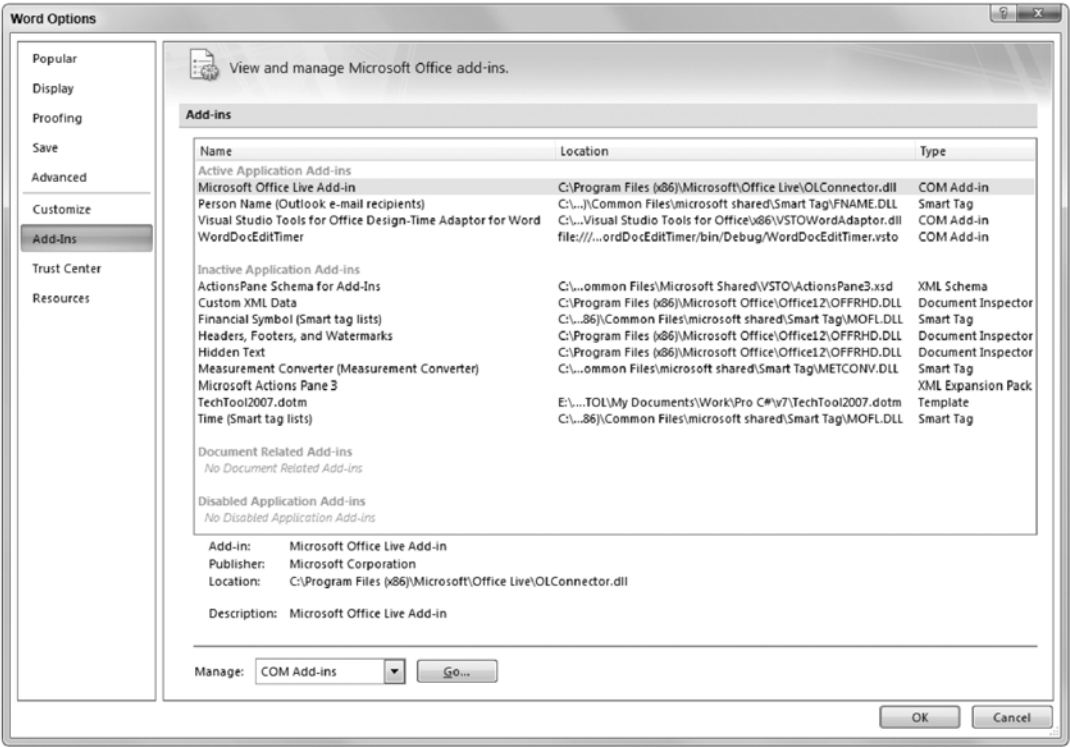


图 49-6

图 49-6 显示了用 VSTO 创建的一个插件：WordDocEditTimer。要添加或删除插件，必须在 Manage 下拉列表中选择 COM Add-Ins(默认选项)，单击 Go 按钮，打开如图 49-7 所示的对话框。

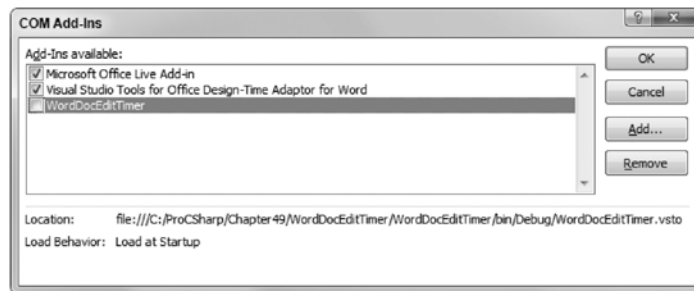


图 49-7

在 COM Add-Ins 对话框中取消选择插件，就会卸载插件，如图 49-7 所示。还可以使用 Add 和 Remove 按钮添加新插件或删除旧插件。

49.3.2 与应用程序和文档交互操作

无论创建什么类型的应用程序，都要与宿主应用程序和/或宿主操作中的文档进行交互。这包括使用下一节介绍的 UI 自定义功能。还可能需监控应用程序中的文档，这表示必须处理一些 Office 对象模型事件。例如，要监控 Word 中的文档，需要 Microsoft.Office.Interop.Word.Application Events4_Event 接口中如下事件的事件处理程序：

- DocumentOpen——打开文档时引发
- NewDocument——创建新文档时引发
- DocumentBeforeClose——保存文档时引发

另外，Word 第一次启动时，它会加载一个文档，它可以是空白的新文档，也可以是已加载的旧文档。



本章的可下载代码包含一个 WordDocEditTimer 示例，它维护 Word 文档的一个编辑次数表。这个应用程序的部分功能是监控已加载的文档，其原因在后面解释。因为这个示例也使用自定义任务面板和 ribbon 菜单，所以在介绍完这些主题后，再介绍这个示例。

在 Word 中，通过 ThisAddIn.Application.ActivateDocument 属性可以访问当前活动的文档，通过 ThisAddIn.Application.Documents 属性访问已打开的文档集合。由于有了多文档界面(Multiple Document Interface, MDI)，类似的属性也存在于其他 Office 应用程序中。例如，可以通过 Microsoft.Office.Interop.Word.Document 类的属性操作文档的各个属性。

这里要注意，在开发 VSTO 解决方案时，必须处理的类和类成员的数量相当大。除非已经习惯了，否则很难找到需要的功能。例如，在 Word 中，当前活动选区不是通过活动文档获得的，而是通过应用程序获得的(利用 ThisAddIn.Application.Selection 属性)，其原因并不是很明显。

通过 Range 属性可以把选区应用于插入、读取或替换文本的操作。例如：

```
ThisAddIn.Application.Selection.Range.Text = "Inserted text";
```

但是，本章没有足够的篇幅来详细介绍对象库，而读者可以在本章讨论相关的内容时学习对象库。

49.3.3 UI 的自定义

在 VSTO 的最新版本中，最重要的方面是可用于定制用户的自定义 UI 和插件的灵活性。可以给已有的功能区菜单中添加内容，添加全新的功能区菜单，通过添加动作面板定制动作面板，添加全新的动作面板，集成 Windows 窗体、WPF 窗体和控件。

本节介绍这些主题。

1. 功能区菜单

可以在本章介绍的所有 VSTO 项目中添加功能区菜单。添加功能区菜单时，会看到如图 49-8 所示的设计器窗口。



图 49-8

设计器可以给 Office 按钮菜单和功能区菜单上的组添加控件(显示在图 49-8 的左上部)，来定制这个功能区菜单。也可以添加其他组。

功能区中使用的类在 `Microsoft.Office.Tools.Ribbon` 名称空间中。这包括用于创建功能区的派生 ribbon 类 `OfficeRibbon`。这个类可以包含 `RibbonTab` 对象，每个 `RibbonTab` 对象都包含单个选项卡的内容。选项卡则包含 `RibbonGroup` 对象，类似图 49-8 中的 `group1` 组。这些选项卡都可以包含各种控件。

选项卡上的组可以位于目标 Office 应用程序的一个全新选项卡上，或者位于其中一个已有的选项卡上。组在何处显示取决于 `RibbonTab.ControlId` 属性。这个属性有一个 `ControlIdType` 属性，它可以设置为 `RibbonControlIdType.Custom` 或 `RibbonControlIdType.Office`。如果使用 `Custom`，那么还必须把 `RibbonTab.ControlId.CustomId` 设置为 `String` 值，这是选项卡的标识符。这里可以使用任意标识符。但如果给 `ControlIdType` 属性使用 `Office`，就必须把 `RibbonTab.ControlId.OfficeId` 设置为一个 `String` 值，该值匹配在当前 Office 产品中使用的其中一个标识符。例如，在 Excel 中，把这个属性设置为 `TabHome`，可以把组添加到 Home 选项卡中；设置为 `TabInsert`，可以把组添加到 Insert 选项卡中等。插件的默认属性是 `TabAddIns`，它由所有插件共享。

许多选项卡可用，尤其在 Outlook 中；可以从 www.microsoft.com/downloads/details.aspx?FamilyID=4329D9E9-4D11-46A5-898D-23E4F331E9AE&displaylang=en 中下载包含完整列表的一系列电子表格。

一旦决定在何处放置 ribbon 组后，就可以添加如表 49-9 所示的控件。

表 49-8

控 件	说 明
RibbonBox	这是一个容器控件，它可用于排放组中的其他控件。可以把 BoxStyle 属性改为 RibbonBoxStyle.Horizontal 或 RibbonBoxStyle.Vertical，在 RibbonBox 中水平或垂直排放控件
RibbonButton	这个控件可用于在组中添加大按钮或小按钮，在按钮的旁边可以有或没有文本标签。把 ControlSize 属性设置为 RibbonControlSize.RibbonControlSizeLarge 或 RibbonControlSize.RibbonControlSizeRegular，就可以控制大小。按钮的 Click 事件处理程序可用于响应交互操作。还可以设置自定义图像或存储在 Office 系统中的其中一幅图像(详见本表后面的内容)
RibbonButtonGroup	这是一个容器控件，它表示一组按钮。它可以包含 RibbonButton、RibbonGallery、RibbonMenu、RibbonSplitButton 和 RibbonToggleButton 控件
RibbonCheckBox	复选框控件，有 Click 事件和 Checked 属性
RibbonComboBox	组合框(合并了文本项和下拉列表项)。列表项使用 Items 属性，输入的文本使用 Text 属性，TextChanged 事件用于响应文本更改
RibbonDropDown	这个容器可以包含 RibbonDropDownItem 和 RibbonButton 列表项，它们分别在 Items 和 Buttons 属性中指定。按钮和列表项格式化到下拉列表中。使用 SelectionChanged 事件响应交互操作
RibbonEditBox	文本框，用户可用于输入或编辑 Text 属性中的文本。这个控件有 TextChanged 事件
RibbonGallery	与 RibbonDropDown 控件相同，这个控件也可以包含 RibbonDropDownItem 和 RibbonButton 项，它们分别在 Items 和 Buttons 属性中指定。这个控件使用 Click 和 ButtonClick 事件，来替代 RibbonDropDown 控件的 SelectionChanged 事件
RibbonLabel	显示简单的文本，用 Label 属性设置
RibbonMenu	弹出菜单，在设计视图中打开它时，可以用其他控件填充该菜单，如 RibbonButton 和嵌套的 RibbonMenu 控件。处理菜单上的菜单项的事件
RibbonSeparator	一个简单的分隔符，用于定制组中的控件布局
RibbonSplitButton	合并了 RibbonButton 或 RibbonToggleButton 和 RibbonMenu 的控件。用 ButtonType 设置按钮的样式，该属性可以是 RibbonButtonType.Button 或 RibbonButtonType.ToggleButton。使用主按钮的 Click 事件或菜单中各按钮的 Click 事件来响应交互操作
RibbonToggleButton	一个按钮，可以处于选中或未选中状态，用 Checked 属性指定。这个控件也有 Click 事件

也可以设置组的 DialogBoxLauncher 属性，以便把一个图标显示在组的右下部。顾名思义，使用这个属性可以显示一个对话框，或者打开一个任务面板，或者执行任何其他操作。通过 GroupView Tasks 菜单可以添加或删除这个图标，如图 49-9 所示，该图还显示了表 49-8 中的其他一些控件，因为它们在设计视图中显示在功能区上。

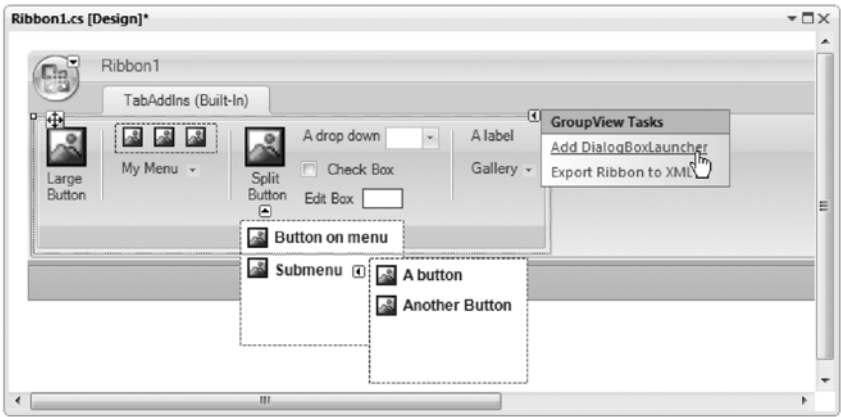


图 49-9

要给控件设置图像, 如给 `RibbonButton` 控件设置图像, 就可以把 `Image` 属性设置为自定义图像, 把 `ImageName` 设置为图像名(以便在 `OfficeRibbon.LoadImage` 事件处理程序中优化图像的加载), 也可以使用内置的 Office 图像。为此, 应把 `OfficeImageId` 属性设置为图像的 ID。

可以使用许多图像; 还可以从 www.microsoft.com/downloads/details.aspx?familyid=12b99325-93e8-4ed4-8385-74d0f7661318&displaylang=en 上下载列出这些图像的电子表格。图 49-10 显示了一个示例。

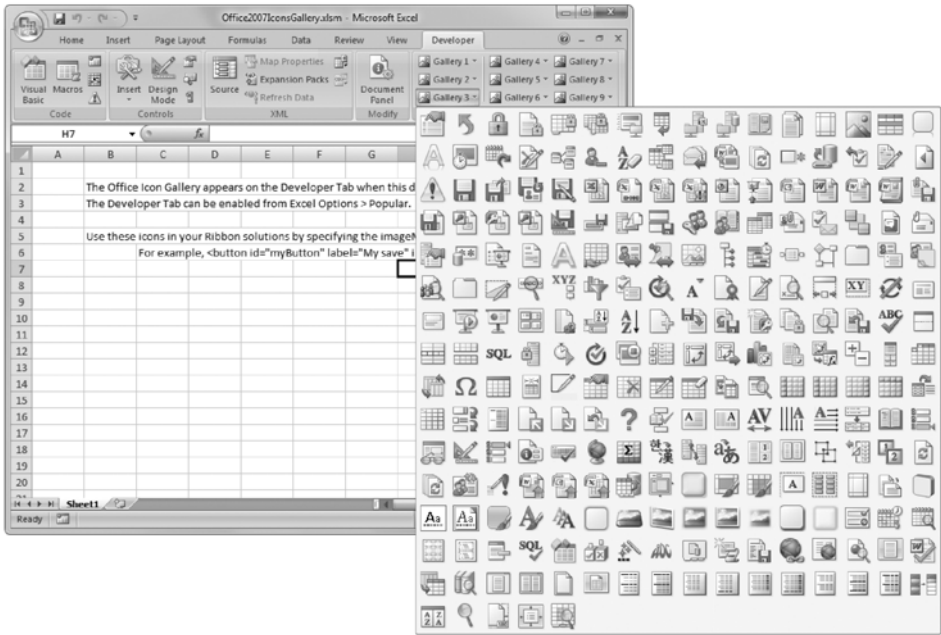


图 49-10

图 49-10 显示了 Developer 功能区选项卡, 通过 Popular 选项卡上的 Excel Options 对话框中的 Office 按钮可以启用它。

单击一幅图像，就会打开一个对话框，指出该图像的 ID 是什么，如图 49-11 所示。

功能区设计器非常灵活，还可以提供希望出现在 Office 功能区上的许多额外功能。但是，如果要进一步定制 UI，就要使用动作面板和任务面板，因为可以通过它们创建任意 UI 和功能。

2. 动作面板和自定义任务面板

使用动作面板和任务面板可以显示停靠在 Office 应用程序界面的任务面板区域中的内容。任务面板在应用程序级的插件中使用，动作面板在文档级自定义中使用。任务面板和动作面板都必须继承自 UserControl 对象，这表示应使用 Windows 窗体创建一个 UI。如果把 WPF 窗体驻留在 UserControl 的 ElementHost 控件上，那么还可以使用 WPF UI。

要把动作面板添加到文档级自定义中的一个文档中，应把动作面板类的一个实例添加到文档的 ActionsPane 属性的 Controls 集合中。例如：

```
public partial class ThisWorkbook
{
    private UserControl1 actionsPane;
    private void ThisWorkbook_Startup(object sender, System.EventArgs e)
    {
        Workbook wb = Globals.Factory.GetVstoObject(this.Application.ActiveWorkbook);
        wb.AcceptAllChanges();
        actionsPane = new UserControl1();
        this.ActionsPane.Controls.Add(actionsPane);
    }
    ...
}
```

这段代码在加载文档(这里是 Excel 工作簿)时添加了动作面板。也可以在 ribbon 按钮事件处理程序中添加动作面板。

在应用程序级的插件项目中，自定义任务面板通过 ThisAddIns.CustomTaskPanes.Add()方法属性添加。这个方法也允许命名任务窗口，例如：

```
public partial class ThisAddIn
{
    Microsoft.Office.Tools.CustomTaskPane taskPane;
    private void ThisAddIn_Startup(object sender, System.EventArgs e)
    {
        taskPane = this.CustomTaskPanes.Add(new UserControl1(), "My Task Pane");
        taskPane.Visible = true;
    }
    ...
}
```

注意 Add()方法返回一个 Microsoft.Office.Tools.CustomTaskPane 类型的对象。可以通过这个对象的 Control 属性访问用户控件本身。还可以使用这个类型的其他属性，例如，上面代码中的 Visible 属性，来控制任务面板。



图 49-11

此时,应注意 Office 应用程序的一个不太寻常的功能,尤其是 Word 和 Excel 之间的区别。由于历史原因,尽管 Word 和 Excel 都是 MDI 应用程序,但这两个应用程序驻留文档的方式不同。在 Word 中,每个文档都有一个唯一的父窗口。而在 Excel 中,每个文档都共享同一个父窗口。

在调用 `CustomTaskPanes.Add()` 方法时,默认行为是把任务面板添加到当前的活动窗口中。在 Excel 中,这表示每个文档都显示该任务面板,因为它们都使用同一个父窗口。而在 Word 中,情况有所不同。如果希望任务面板显示给每个文档,就必须把它添加到包含文档的每个窗口中。

要把任务面板添加到特定的文档中,应给 `Add()` 方法传递 `Microsoft.Office.Interop.Word.Windows` 类的一个实例,作为第 3 个参数。通过 `Microsoft.Office.Interop.Word.Document.ActiveWindow` 属性可以获得与文档关联的窗口。

下一节介绍如何完成这个操作。

49.4 示例应用程序

如前所述,本章的示例代码包含一个 `WordDocEditTimer` 应用程序,它维护 Word 文档的一个编辑次数列表。本节将详细解释这个应用程序的代码,因为该应用程序揭示了前面介绍的所有内容,还包含一些有用的提示。

这个应用程序的一般操作是只要创建或加载文档,就启动一个链接到文档名上的计时器。如果关闭文档,该文档的计时器就暂停。如果打开以前计时的文档,计时器就恢复。另外,如果使用 `Save As` 把文档另存为另一个文件名,计时器就更新为使用新文件名。

这个应用程序是一个 Word 应用程序级的插件,使用一个自定义任务面板和一个功能区菜单。功能区菜单包含一个按钮和一个复选框,按钮用于打开和关闭任务面板,复选框用于暂停当前的活动文档的计时器。包含这些控件的组追加到 `Home` 功能区选项卡的最后。任务面板显示一个活动计时器列表。

这个用户界面如图 49-12 所示。

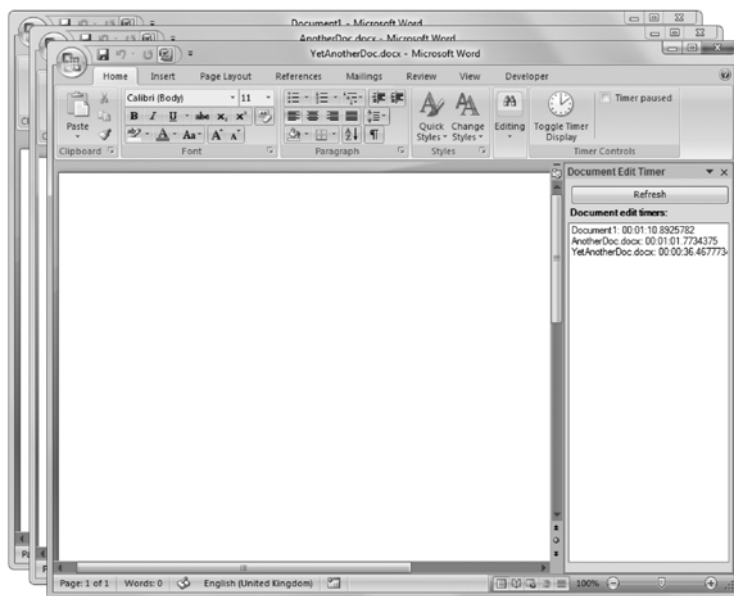


图 49-12

计时器通过 DocumentTimer 类来维护:



可从
wrox.com
下载源代码

```
public class DocumentTimer
{
    public Word.Document Document { get; set; }
    public DateTime LastActive { get; set; }
    public bool IsActive { get; set; }
    public TimeSpan EditTime { get; set; }
}
```

代码段 DocumentTimer.cs

这段代码保存了对 Microsoft.Office.Interop.Word.Document 接口的一个引用、总编辑时间、计时器是否激活, 以及它上一次激活的时间。ThisAddIn 类维护这些对象的一个集合, 这些对象与文档名关联起来:



可从
wrox.com
下载源代码

```
public partial class ThisAddIn
{
    private Dictionary<string, DocumentTimer> documentEditTimes;
```

代码段 DocumentTimer.cs

因此, 每个计时器都可以通过文档引用或文档名来定位。这是必要的, 因为文档引用可以跟踪文档名的变化(这里没有可用于监控文档名变化的事件), 文档名允许跟踪已关闭和再次打开的文档。

ThisAddIn 类还维护一个 CustomTaskPane 对象列表(如前所述, Word 中的每个窗口都需要一个 CustomTaskPane 对象):

```
private List<Tools.CustomTaskPane> timerDisplayPanels;
```

插件启动时, ThisAddIn_Startup() 方法执行几个任务。首先它初始化两个集合:

```
private void ThisAddIn_Startup(object sender, System.EventArgs e)
{
    // Initialize timers and display panels
    documentEditTimes = new Dictionary<string, DocumentTimer>();
    timerDisplayPanels = new List<Microsoft.Office.Tools.CustomTaskPane>();
}
```

接着通过 ApplicationEvents4_Event 接口添加几个事件处理程序:

```
// Add event handlers
Word.ApplicationEvents4_Event eventInterface = this.Application;
eventInterface.DocumentOpen += new Microsoft.Office.Interop.Word
    .ApplicationEvents4_DocumentOpenEventHandler(
        eventInterface_DocumentOpen);
eventInterface.NewDocument += new Microsoft.Office.Interop.Word
    .ApplicationEvents4_NewDocumentEventHandler(
        eventInterface_NewDocument);
eventInterface.DocumentBeforeClose += new Microsoft.Office.Interop.Word
    .ApplicationEvents4_DocumentBeforeCloseEventHandler(
        eventInterface_DocumentBeforeClose);
eventInterface.WindowActivate += new Microsoft.Office.Interop.Word
    .ApplicationEvents4_WindowActivateEventHandler(
```

```
eventInterface_WindowActivate);
```

这些事件处理程序用于监控文档的打开、创建和关闭，并确保功能区上的 **Pause** 复选框保持最新状态。后一个功能使用 **WindowsActivate** 事件跟踪窗口的激活状态来实现。

在这个事件处理程序中，最后一个任务是开始监控当前文档，把自定义任务面板添加到包含文档的窗口中：

```
// Start monitoring active document
MonitorDocument(this.Application.ActiveDocument);
AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);
}
```

MonitorDocument()实用程序方法为文档添加一个计时器：

```
internal void MonitorDocument(Word.Document Doc)
{
    // Monitor doc
    documentEditTimes.Add(Doc.Name, new DocumentTimer
    {
        Document = Doc,
        EditTime = new TimeSpan(0),
        IsActive = true,
        LastActive = DateTime.Now
    });
}
```

这个方法仅为文档创建了一个新的 **DocumentTimer** 对象。该 **DocumentTimer** 引用文档，其编辑次数是 0，处于激活状态，且在当前时间激活。接着把这个计时器添加到 **documentEditTimes** 集合中，并把它关联到文档名上。

AddTaskPaneToWindow()方法把自定义任务面板添加到窗口中。这个方法首先检查已有的任务面板，确保窗口中还没有一个任务面板。此外，**Word** 中的另一个奇怪的功能是如果在加载应用程序后，立即打开一个旧文档，默认的 **Document1** 文档就会消失，且不引发关闭事件。因为在文档中访问包含任务面板的文档窗口时，这可能导致引发异常，所以该方法还检查表示该异常的 **ArgumentNullException**：

```
private void AddTaskPaneToWindow(Word.Window Wn)
{
    // Check for task pane in window
    Tools.CustomTaskPane docPane = null;
    Tools.CustomTaskPane paneToRemove = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        try
        {
            if (pane.Window == Wn)
            {
                docPane = pane;
                break;
            }
        }
        catch (ArgumentNullException)
        {
            // pane.Window is null, so document1 has been unloaded.
        }
    }
}
```

```

        paneToRemove = pane;
    }
}

```

如果抛出一个异常，就从集合中删除冲突的任务面板：

```

// Remove pane if necessary
timerDisplayPanes.Remove(paneToRemove);

```

如果窗口中没有任务面板，这个方法最终就添加一个任务面板：

```

// Add task pane to doc
if (docPane == null)
{
    Tools.CustomTaskPane pane = this.CustomTaskPanes.Add(
        new TimerDisplayPane(documentEditTimes),
        "Document Edit Timer",
        Wn);
    timerDisplayPanes.Add(pane);
    pane.VisibleChanged +=
        new EventHandler(timerDisplayPane_VisibleChanged);
}
}

```

添加的任务面板是 `TimerDisplayPane` 类的一个实例。稍后介绍这个类。添加它时使用的名称是 `Document Edit Timer`。另外，在调用 `CustomTaskPanes.Add()` 方法后，还为得到的 `CustomTaskPane` 的 `VisibleChanged` 事件添加了一个事件处理程序。这样在第一次显示任务面板时，可以刷新显示内容：

```

private void timerDisplayPane_VisibleChanged(object sender, EventArgs e)
{
    // Get task pane and toggle visibility
    Tools.CustomTaskPane taskPane = (Tools.CustomTaskPane)sender;
    if (taskPane.Visible)
    {
        TimerDisplayPane timerControl = (TimerDisplayPane)taskPane.Control;
        timerControl.RefreshDisplay();
    }
}

```

`TimerDisplayPane` 类提供一个 `RefreshDisplay()` 方法，它在上面的代码中调用。顾名思义，这个方法刷新 `timerControl` 对象的显示内容。

接着，代码确保监控所有文档。首先，创建新文档时，调用 `eventInterface_NewDocument()` 事件处理程序，调用 `MonitorDocument()` 和前面介绍过的 `AddTaskPaneToWindow()` 方法监控该文档。

```

private void eventInterface_NewDocument(Word.Document Doc)
{
    // Monitor new doc
    MonitorDocument(Doc);
    AddTaskPaneToWindow(Doc.ActiveWindow);
}

```

新文档在计时器运行时启动，此时这个方法还清除了功能区菜单中的 `Pause` 复选框。这通过一个实用程序方法 `SetPauseStatus()` 实现，该方法在功能区中定义：

```
// Set checkbox
Globals.Ribbon.TimerRibbon.SetPauseStatus(false);
}
```

在关闭文档之前,调用 `eventInterface_DocumentBeforeClose()` 事件处理程序。这个方法冻结文档的计时器,更新总编辑时间,清除 `Document` 引用,并删除文档窗口中的任务面板(使用稍后介绍的 `RemoveTaskPaneFromWindow()` 方法),之后关闭窗口。

```
private void eventInterface_DocumentBeforeClose(Word.Document Doc,
    ref bool Cancel)
{
    // Freeze timer
    documentEditTimes[Doc.Name].EditTime += DateTime.Now
        - documentEditTimes[Doc.Name].LastActive;
    documentEditTimes[Doc.Name].IsActive = false;
    documentEditTimes[Doc.Name].Document = null;
    // Remove task pane
    RemoveTaskPaneFromWindow(Doc.ActiveWindow);
}
```

打开文档时,调用 `eventInterface_DocumentOpen()` 方法。此处该方法要完成更多工作,因为在监控文档之前,这个方法必须查看计时器的名称,确定文档是否已有计时器:

```
private void eventInterface_DocumentOpen(Word.Document Doc)
{
    if (documentEditTimes.ContainsKey(Doc.Name))
    {
        // Monitor old doc
        documentEditTimes[Doc.Name].LastActive = DateTime.Now;
        documentEditTimes[Doc.Name].IsActive = true;
        documentEditTimes[Doc.Name].Document = Doc;
        AddTaskPaneToWindow(Doc.ActiveWindow);
    }
}
```

如果目前还没有监控文档,就为新文档配置一个新监控器:

```
else
{
    // Monitor new doc
    MonitorDocument(Doc);
    AddTaskPaneToWindow(Doc.ActiveWindow);
}
}
```

`RemoveTaskPaneFromWindow()` 方法用于从窗口中删除任务面板。其代码首先检查特定的窗口中是否有任务面板:

```
private void RemoveTaskPaneFromWindow(Word.Window Wn)
{
    // Check for task pane in window
    Tools.CustomTaskPane docPane = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        if (pane.Window == Wn)
```

```

        {
            docPane = pane;
            break;
        }
    }

```

如果找到一个任务面板，就调用 `CustomTaskPanes.Remove()` 方法删除它。还要从任务面板引用的本地集合中删除它。

```

// Remove document task pane
if (docPane != null)
{
    this.CustomTaskPanes.Remove(docPane);
    timerDisplayPanes.Remove(docPane);
}
}

```

这个类中的最后一个事件处理程序是 `eventInterface_WindowActivate()`，在激活窗口时调用它。这个方法获得活动文档的计时器(先检查文档是否有计时器，因为在调用这个方法时，新文档可能还没有添加计时器)，选中功能区菜单上的复选框，以便更新文档的复选框：

```

private void eventInterface_WindowActivate(Word.Document Doc,
    Word.Window Wn)
{
    if (documentEditTimes.ContainsKey(this.Application.ActiveDocument.Name))
    {
        // Ensure pause checkbox in ribbon is accurate, start by getting timer
        DocumentTimer documentTimer =
            documentEditTimes[this.Application.ActiveDocument.Name];
        // Set checkbox
        Globals.Ribbons.TimerRibbon.SetPauseStatus(!documentTimer.IsActive);
    }
}

```

`ThisAddIn` 类的代码还包含两个实用程序方法。第一个方法——`ToggleTaskPaneDisplay()`用于设置 `CustomTaskPanes.Visible` 属性，为当前的活动文档显示或隐藏任务面板。

```

internal void ToggleTaskPaneDisplay()
{
    // Ensure window has task window
    AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);
    // toggle document task pane
    Tools.CustomTaskPane docPane = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        if (pane.Window == this.Application.ActiveDocument.ActiveWindow)
        {
            docPane = pane;
            break;
        }
    }
    docPane.Visible = !docPane.Visible;
}

```

上述代码中的 `ToggleTaskPaneDisplay()` 方法由功能区控件上的事件处理程序调用，如后面所述。最后，该类有另一个从功能区菜单中调用的方法，它允许功能区控件暂停或恢复文档的计时器：

```
internal void PauseOrResumeTimer(bool pause)
{
    // Get timer
    DocumentTimer documentTimer =
        documentEditTimes[this.Application.ActiveDocument.Name];
    if (pause && documentTimer.IsActive)
    {
        // Freeze timer
        documentTimer.EditTime += DateTime.Now - documentTimer.LastActive;
        documentTimer.IsActive = false;
    }
    else if (!pause && !documentTimer.IsActive)
    {
        // Resume timer
        documentTimer.IsActive = true;
        documentTimer.LastActive = DateTime.Now;
    }
}
```

这个类定义中的其他代码是 `Shutdown` 事件的空事件处理程序，以及 `VSTO` 为关联 `Startup` 和 `Shutdown` 事件处理程序而生成的代码。

接着布置项目中的功能区，即 `TimerRibbon`，如图 49-13 所示。

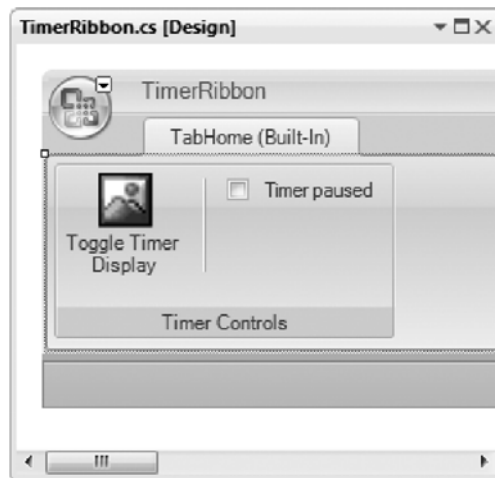


图 49-13

这个功能区包含一个 `RibbonButton`、一个 `RibbonSeparator`、一个 `RibbonCheckBox` 和一个 `DialogBoxLauncher`。按钮使用大显示样式，其 `OfficeImageId` 设置为 `StartAfterPrevious`，显示如图 49-13 所示的钟面(这些图像在设计期间不可见)。功能区使用 `TabHome` 选项卡类型，从而其内容追加到 `Home` 选项卡中。

功能区有 3 个事件处理程序，每个处理程序都调用前面介绍的 `ThisAddIn` 类的一个实用程序方法：



```
private void group1_DialogLauncherClick(object sender,
    RibbonControlEventArgs e)
{
    // Show or hide task pane
    Globals.ThisAddIn.ToggleTaskPaneDisplay();
}
private void pauseCheckBox_Click(object sender, RibbonControlEventArgs e)
{
    // Pause timer
    Globals.ThisAddIn.PauseOrResumeTimer(pauseCheckBox.Checked);
}
private void toggleDisplayButton_Click(object sender,
    RibbonControlEventArgs e)
{
    // Show or hide task pane
    Globals.ThisAddIn.ToggleTaskPaneDisplay();
}
```

代码段 TimerRibbon.cs

ribbon 还包含自己的实用程序方法 `SetPauseStatus()`，如前所述，该方法由 `ThisAddIn` 类中的代码调用，以选中复选框或清除复选框。

```
internal void SetPauseStatus(bool isPaused)
{
    // Ensure checkbox is accurate
    pauseCheckBox.Checked = isPaused;
}
```

这个解决方案中的另一个组件是在任务面板中使用的 `TimerDisplayPane` 用户控件。这个控件的布局如图 49-14 所示。

这个控件包含一个按钮、一个标签和一个列表框——这些都是很普通的显示控件，也可以用更漂亮的 WPF 控件替代它们也很简单。

该控件的代码保存对文档计时器的一个本地引用，该引用在构造函数中设置：

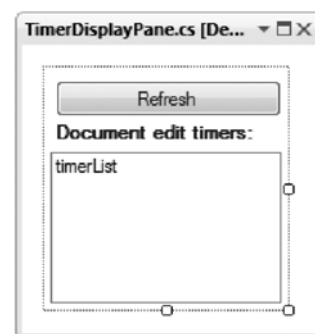


图 49-14



```
public partial class TimerDisplayPane : UserControl
{
    private Dictionary<string, DocumentTimer> documentEditTimes;
    public TimerDisplayPane()
    {
        InitializeComponent();
    }
    public TimerDisplayPane(Dictionary<string, DocumentTimer>
        documentEditTimes): this()
    {
        // Store reference to edit times
        this.documentEditTimes = documentEditTimes;
    }
}
```

代码段 TimerDisplayPane.cs

按钮事件处理程序调用 `RefreshDisplay()` 方法刷新计时器的显示内容：

```
private void refreshButton_Click(object sender, EventArgs e)
{
    RefreshDisplay();
}
```

RefreshDisplay()方法也从 ThisAddIn 类中调用,如前所述。考虑到该方法的任务,这是一个相当复杂的方法。它还参照已加载文档列表,检查被监控的文档列表,并修正任何问题。这种代码在 VSTO 应用程序中常常是必不可少的,因为 COM Office 对象模型的接口偶尔不能像期望的那样工作。这里的经验规则是防御式编码。

该方法首先清除 timerList 列表框中的当前计时器列表:

```
internal void RefreshDisplay()
{
    // Clear existing list
    this.timerList.Items.Clear();
}
```

接着,检查监控器。这个方法迭代 Globals.ThisAddIn.Application.Documents 集合中的每个文档,确定文档是被监控、未被监控、或被监控了但从上次刷新以后改变了文件名。

要找出被监控的文档,只需根据键的 documentEditTimes 集合中的文档名,检查文档名:

```
// Ensure all docs are monitored
foreach (Word.Document doc in Globals.ThisAddIn.Application.Documents)
{
    bool isMonitored = false;
    bool requiresNameChange = false;
    DocumentTimer oldNameTimer = null;
    string oldName = null;
    foreach (string documentName in documentEditTimes.Keys)
    {
        if (doc.Name == documentName)
        {
            isMonitored = true;
            break;
        }
    }
}
```

如果文档名不匹配,就比较文档引用,这样可以检测对文档名的更改,如下面的代码所示:

```
else
{
    if (documentEditTimes[documentName].Document == doc)
    {
        // Monitored, but name changed!
        oldName = documentName;
        oldNameTimer = documentEditTimes[documentName];
        isMonitored = true;
        requiresNameChange = true;
        break;
    }
}
```

对于未监控的文档,需要创建一个新的监控器:

```
// Add monitor if not monitored
if (!isMonitored)
{
    Globals.ThisAddIn.MonitorDocument(doc);
}
```

而名称改变的文档需要通过用于旧命名文档的监控器重新关联起来：

```
// Rename if necessary
if (requiresNameChange)
{
    documentEditTimes.Remove(oldName);
    documentEditTimes.Add(doc.Name, oldNameTimer);
}
}
```

调整文档编辑计时器后，生成一个列表。代码还会检测引用的文档是否依然加载，对于没有依然加载的文档，把 `IsActive` 属性设置为 `false`，暂停该文档的计时器。这也是防御性编程方式：

```
// Create new list
foreach (string documentName in documentEditTimes.Keys)
{
    // Check to see if doc is still loaded
    bool isLoading = false;
    foreach (Word.Document doc in
        Globals.ThisAddIn.Application.Documents)
    {
        if (doc.Name == documentName)
        {
            isLoading = true;
            break;
        }
    }
    if (!isLoading)
    {
        documentEditTimes[documentName].IsActive = false;
        documentEditTimes[documentName].Document = null;
    }
}
```

对于每个监控器，把一个列表项添加到列表框中，其列表框包含文档名和总编辑时间：

```
// Add item
this.timerList.Items.Add(string.Format("{0}: {1}", documentName,
    documentEditTimes[documentName].EditTime +
    (documentEditTimes[documentName].IsActive ?
    (DateTime.Now - documentEditTimes[documentName].LastActive):
    new TimeSpan(0))));
}
}
```

这就完成了这个例子中的代码。这个例子说明了如何使用功能区和任务面板控件，如何维护多个 Word 文档中的任务面板。它还揭示了本章前面介绍的许多技术。

49.5 小结

本章学习了如何使用 VSTO 为 Office 产品创建托管解决方案。

本章的第一部分介绍了 VSTO 项目的一般结构和可以创建的项目类型，还探讨了可用于简化 VSTO 编程的功能。

接下来详细论述了 VSTO 解决方案中可用的一些功能，讨论了如何实现与 Office 对象模型通信，介绍了 VSTO 中可用的名称空间和类型，学习了如何使用这些类型实现各种功能。之后，探讨了 VSTO 项目的一些编码功能，以及如何使用这些功能获得希望的结果。

然后，进行了一些实践。我们学习了如何在 Office 应用程序中管理插件，如何与 Office 对象模型交互操作，如何用功能区菜单、任务面板和动作面板定制应用程序的 UI。

最后，探讨了一个示例应用程序，它揭示了前面学习的 UI 和交互操作技术。这个示例不仅包含许多代码，还包含有用的技巧，包括如何在多个 Word 文档窗口中管理任务面板。

第50章

MAF

本章内容如下：

- MAF 的体系结构
- 定义协定
- 实现管道
- 创建插件
- 保存插件

本章详细介绍 Managed Add-In Framework(MAF)的体系结构,以及如何使用 MAF 创建和驻留插件。我们将学习 MAF 的体系结构,它通过驻留插件解决什么问题,如版本问题、发现、激活和隔离。接着讨论创建 MAF 管道所需的所有步骤,MAF 管道可用于连接插件和宿主;如何创建插件;如何创建使用插件的宿主应用程序。

50.1 MAF 体系结构

.NET 4 包含两个用于创建和使用插件的技术。第 28 章介绍了如何使用 Managed Extensibility Framework(MEF)创建插件。使用 MEF 的应用程序在运行期间在目录或程序集中查找插件,并使用属性连接它们。MAF 与 MEF 的区别是,MEF 没有使用应用程序域或不同的进程把插件与宿主应用程序分隔开。要把插件和宿主应用程序分隔开,应使用 MAF。但为了达到这个目的,MAF 的复杂性提高了不少。如果希望利用 MEF 和 MAF 的优点,就可以合并这两个技术。当然,这也会增加复杂性。

创建允许在运行期间添加插件的应用程序时,需要处理一些问题。例如,如何找到插件,如何解决版本问题,以便宿主应用程序和插件可以独立地升级。本节讨论插件的问题和 MAF 的体系结构如何解决这些问题。

要创建一个宿主应用程序,动态加载以后添加的程序集,必须解决几个问题,如表 50-1 所示。

表 50-1

插 件 问 题	说 明
发现	如何为宿主应用程序查找新插件? 这有几个选项。一个选项是在配置文件中添加插件的信息。其缺点是安装新插件时,需要修改已有的配置文件。另一个选项是把包含插件的程序集复制到预定义的目录中,通过反射读取程序集的信息。 反射的更多内容可参见第 14 章

(续表)

插 件 问 题	说 明
激活	程序集动态加载后，还不能使用 new 运算符创建实例。但可以用 Activator 类创建这类程序集。另外，如果插件加载到另一个应用程序域中或一个新进程中，那么还需要使用不同的激活选项。程序集和应用程序域的更多内容可参见第 18 章
隔离	插件可能会使宿主应用程序崩溃，读者可能见过 IE 因各种插件而崩溃的情况。根据宿主应用程序的类型和插件的集成方式，插件可以加载到另一个应用程序域或另一个进程中
生命周期	清理对象是垃圾回收器的工作。但是，垃圾回收器在这里没有任何帮助，因为插件可能在另一个应用程序域中或另一个进程中激活。把对象保存在内存中的其他方式有引用计数、租借和主办机制
版本	版本问题是插件的一个大问题。通常宿主的一个新版本仍可以加载旧插件，而旧宿主应有加载新插件的选项

下面探讨 MAF 的体系结构，说明这个架构如何解决这些问题。影响 MAF 的设计目标如下：

- 应易于开发插件
- 在运行期间查找插件应很高效
- 开发宿主应用程序应是一个很简单过程，但不像开发插件那么容易
- 插件和宿主应用程序应独立地升级

MAF 使用管道解决了这些问题，管道在其核心使用协定。我们将讨论 MAF 如何使用发现功能来查找插件，如何激活插件，如何使插件保持激活状态，如何处理版本问题。

50.1.1 管道

MAF 体系结构基于一个包含 7 个程序集的管道。这个管道解决了插件的版本问题。因为管道中的程序集之间的依赖性很弱，所以协定、宿主程序和插件应用程序升级到新版本可以完全互不干扰。

图 50-1 显示了 MAF 体系结构的管道。其中心是协定程序集。这个程序集包含一个协定接口，其中列出了插件必须实现并且可以由宿主调用的方法和属性。协定的左边是宿主端，右边是插件端。图 50-1 还显示了程序集之间的依赖关系。最左端的宿主程序集与协定程序集没有依赖关系，插件程序集与协定程序集也没有依赖关系，这两个程序集实际上都没有实现协定定义的接口，只是有一个对视图程序集的引用。宿主应用程序引用宿主视图；插件引用插件视图。视图包含抽象的视图类，该类定义的方法和属性与协定相同。

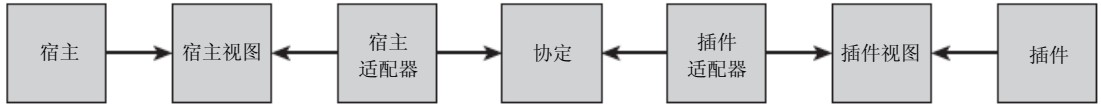


图 50-1

图 50-2 显示了管道中类的关系。宿主类与抽象的宿主视图类有一个关联，并调用其方法。抽象的宿主视图类由宿主适配器实现。适配器在视图和协定之间建立连接。插件适配器实现协定的方法和属性。这个适配器包含对插件视图的引用，把来自宿主端的调用转发给插件视图。宿主适配器类定义了一个具体的类，它派生自宿主视图的抽象基类，用于实现方法和属性。这个适配器包含对协定的一个引用，用于把来自视图的调用转发给协定。

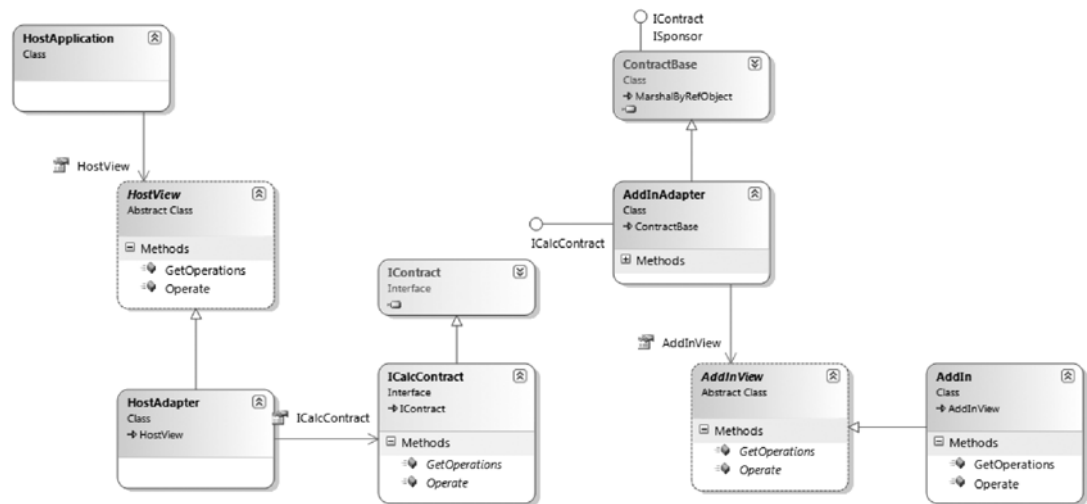


图 50-2

通过这个模型，插件端和宿主端可以完全独立地升级，只是需要调整映射层。例如，如果宿主的一个新版本使用全新的方法和属性，协定就仍可以保持不变，只有适配器需要更改。也可以定义新的协定。适配器可以更改，也可以同时使用几个协定。

50.1.2 发现

如何为宿主应用程序查找新插件？MAF 的体系结构使用一个预定义的目录结构来查找插件和管道的其他程序集。管道的组成部分必须存储在这些子目录中：

- HostSideAdapters
- Contracts
- AddInSideAdapters
- AddInViews
- AddIns

除了 AddIns 目录之外，其他目录都直接包含管道特定部分的程序集。AddIns 目录为每个插件程序集包含一个子目录。插件也可以存储在完全独立于其他管道组件的目录中。

管道的程序集需要使用反射来动态地加载，才能获得插件的所有信息。而且，对于许多插件，这还会增加宿主应用程序的启动时间。因此，MAF 使用一个缓存，来保存管道组件的信息。该缓存由安装插件的程序创建，如果宿主应用程序有管道目录的写入权限，该缓存就由宿主应用程序或安装插件的应用程序创建。

调用 AddInStore 类的方法来创建管道组件的缓存信息。Update()方法查找还没有列在存储文件中的新插件。Rebuild()方法用插件的信息重新生成整个二进制存储文件。

表 50-2 列出了 AddInStore 类的成员。

表 50-2

AddInStore 类的成员	说 明
Rebuild()、 RebuildAddIns()	Rebuild()方法为管道的所有组件重新生成缓存。如果插件存储在另一个目录下，就可以使用 RebuildAddIns()方法重新生成插件的缓存

(续表)

AddInStore 成员	说 明
Update()、 UpdateAddIns()	Rebuild()方法重建管道的完整缓存, Update()方法只用新管道组件的信息更新缓存。UpdateAddIns()方法只更新插件的缓存
FindAddIn()、 FindAddIns()	这些方法都使用缓存查找插件。FindAddIns()方法返回匹配宿主视图的所有插件集合。FindAddIn()方法返回一个特定的插件

50.1.3 激活和隔离

AddInStore 类的 FindAddIns()方法返回表示插件的 AddInToken 对象集合。使用 AddInToken 类可以访问插件的信息,如名称、描述、发布者和版本。使用 Activate()方法可以激活插件。表 50-3 列出了 AddInToken 类的特性和方法。

表 50-3

AddInToken 类的成员	说 明
Name、Publisher、 Version、Description	AddInToken 类的 Name、Publisher、Version 和 Description 属性返回用 AddInAttribute 特性赋予插件的信息
AssemblyName	AssemblyName 返回包含插件的程序集的名称
EnableDirectConnect	使用 EnableDirectConnect 属性可以设置一个值,指出宿主应直接连接到插件上,而不使用管道的组件。只有插件和宿主运行在同一个应用程序域中,插件视图和宿主视图的类型相同时,才能使用这个属性。该属性仍要求管道的所有组件都存在
QualificationData	插件可以用 QualificationDataAttribute 特性标记应用程序域和安全需求。插件可以列出安全需求和隔离需求。例如, [QualificationData (“Isolation”, ”NewAppDomain”)]表示插件必须驻留在新进程中。可以从 AddInToken 类中读取这些信息,激活有特定需求的插件。除了应用程序域和安全需求之外,还可以使用这个属性通过管道传递自定义信息
Activate()	插件用 Activate()方法激活,利用这个方法的参数,可以定义插件是否加载到新应用程序域或新进程中。还可以定义插件获得的权限

一个插件可能使整个应用程序崩溃。例如, IE 可能因一个失败的插件而崩溃。根据应用程序类型和插件类型,可以让插件运行在另一个应用程序域或另一个进程中,来避免这个问题。这里 MAF 给出了几个选项。可以在新应用程序域或新进程中激活插件。新应用程序域还可以有有限的权限。

AddInToken 类的 Activate()方法有几个重载版本,在这些版本中,可以传递应加载的插件的环境参数。表 50-4 列出了不同的选项。

表 50-4

AddInToken.Activate()方法的参数	说 明
AppDomain	可以传递应加载的插件的一个新应用程序域,这样可以使插件独立于宿主应用程序,还可以从应用程序域中卸载插件
AddInSecurityLevel	如果插件应使用不同的安全级别来运行,就传递 AddInSecurityLevel 枚举的一个值,其值可以是 Internet、Intranet、FullTrust 和 Host

(续表)

AddInToken.Activate()方法的参数	说 明
PermissionSet	如果预定义的安全级别不够专用,那么还可以给插件的应用程序域赋予 PermissionSet
AddInProcess	插件还可以运行在与宿主应用程序不同的进程中。可以给 Activate()方法传递一个新的 AddInProcess。如果卸载所有插件,新进程就可以关闭;否则新进程继续运行。这个选项可以用 KeepAlive 属性设置
AddInEnvironment	传递 AddInEnvironment 对象是定义在哪里加载插件的应用程序域的另一个选项。在 AddInEnvironment 的构造函数中,可以传递一个 AppDomain 对象。还可以用 AddInController 类的 AddInEnvironment 属性获得插件的已有 AddInEnvironment



应用程序域详见第 18 章。

应用程序的类型也会限制可以使用的选项。WPF 插件目前不支持跨进程。Windows 窗体不能在不同的应用程序域之间连接 Windows 控件。

下面列出调用 AddInToken 类的 Activate()方法时管道执行的步骤:

- (1) 用指定的权限创建应用程序域。
 - (2) 用 Assembly.LoadFrom()方法把插件的程序集加载到新的应用程序域中。
 - (3) 用反射调用插件的默认构造函数。因为插件派生自在插件视图中定义的基类,所以也加载视图的程序集。
 - (4) 构造插件端适配器的一个实例。因为把插件的这个实例传递给适配器的构造函数,所以适配器能连接协定和插件。因为插件适配器派生自基类 MarshalByRefObject,所以可以在应用程序域之间调用它。
 - (5) 激活代码给宿主应用程序的应用程序域返回插件端适配器的一个代理。因为插件适配器实现协定接口,所以该代理包含协定接口的方法和属性。
 - (6) 宿主端适配器的实例在宿主应用程序的应用程序域中构造。把插件端适配器的代理传递给该构造函数。激活代码会从插件令牌中查找宿主端适配器的类型。
- 宿主端适配器返回宿主应用程序。

50.1.4 协定

协定定义 MAF 体系结构中宿主端和插件端之间的边界。协定用一个接口来定义,该接口必须派生自基接口 IContract。协定必须仔细考虑,因为它根据需要提供支持的灵活的插件场景。

因为协定没有版本冲突,不能改变,以便插件以前的实现代码仍可以在新的宿主程序中运行。新版本应通过定义新协定来创建。

可以使用的协定类型有一些限制,其原因是版本问题,而且应用程序域要从宿主应用程序跨越到插件上。类型必须是安全的,且没有版本冲突,能在边界(应用程序域或跨进程)之间传递,也能在宿主和插件之间传递。

可以用协定传递的类型可以是:

- 基元类型

- 其他协定
 - 可序列化的系统类型
 - 简单的可序列化的自定义类型，包括基本类型、协定，以及没有实现代码的类型
- IContract 接口的成员如表 50-5 所示。

表 50-5

Icontract 接口的成员	说 明
QueryContract()	使用 QueryContract()方法可以查询协定，验证它是否也实现了另一个协定。一个插件可以支持几个协定
RemoteToString()	QueryContract()方法的参数需要协定的字符串表示。RemoteToString()方法返回当前协定的字符串表示
AcquireLifetimeToken() RevokeLifetimeToken()	客户端调用 AcquireLifetimeToken()方法来保存对协定的引用。AcquireLifetimeToken()方法会递增引用计数。RevokeLifetimeToken()方法递减引用计数
RemoteEquals()	RemoteEquals()方法可用于比较两个协定引用

协定接口在 System.AddIn.Contract、System.AddIn.Contract.Collections 和 System.AddIn.Contract.Automation 名称空间中定义。表 50-6 列出了可以用于协定的协定接口。

表 50-6

协 定	说 明
IListContract<T>	IListContract<T>可用于返回一个协定列表
IEnumeratorContract<T>	IEnumeratorContract<T>用于枚举 IListContract<T>的元素
IServiceProviderContract	一个插件可以为其他插件提供服务。提供服务并实现 IServiceProviderContract 接口的插件称为服务提供程序。通过 QueryService()方法，可以查询实现该接口的插件提供什么服务
IProfferServiceContract	IProfferServiceContract 是服务提供程序和 IServiceProviderContract 协定提供的接口。IprofferServiceContract 协定定义 ProfferService()和 Revoke Service()方法。ProfferService()方法给所提供的服务添加一个 IServiceProviderContract 协定，而 RevokeService()方法删除它
INativeHandleContract	这个接口允许使用 GetHandle()方法访问本地 Windows 句柄。这个协定由 WPF 宿主程序用于使用 WPF 插件

50.1.5 生命周期

插件需要加载多长时间？它使用多少时间？何时可以卸载应用程序域？解决上述问题有几个选项。一个选项是使用引用计数。每次使用插件都会递增引用计数。如果引用计数递减到 0，就可以卸载插件。另一个选项是使用垃圾回收器。如果垃圾回收器在运行，且没有再引用对象，该对象就是垃圾回收器的目标。.NET 远程处理使用租约机制，和使对象保持激活状态的主办方。只要租期到了，就询问主办方该对象是否应继续保持激活状态。



应用程序域详见第 18 章。

对于插件，卸载插件还有一个特殊的问题，因为插件运行在不同的应用程序域中和不同的进程中。但垃圾回收器不能跨不同进程工作。MAF 使用一个混合的模型来管理生命周期。在单个应用程序域中，使用垃圾回收机制。在管道内部，虽然使用一个隐式的承办机制，但引用计数可用于从外部控制主办方。

下面考虑一种情况：把插件加载到另一个应用程序域中。在宿主应用程序中，当不再需要引用时，垃圾回收器清理了宿主视图和宿主端适配器。而在插件端，协定定义 `AcquireLifetimeToken()` 和 `RevokeLifetimeToken()` 方法，来递增和递减主办方的引用计数。这两个版本不仅递增和递减一个值，还可以在某一端频繁调用 `RevokeLifetimeToken()` 方法时，提早释放对象。而 `AcquireLifetimeToken()` 方法返回一个表示生命周期令牌的标识符，这个标识符必须用于调用 `RevokeLifetimeToken()` 方法。所以这两个方法总是成对调用。

通常不必处理 `AcquireLifetimeToken()` 和 `RevokeLifetimeToken()` 方法的调用。然而，可以使用 `ContractHandle` 类，在构造函数中调用 `AcquireLifetimeToken()` 方法，在终结器中调用 `RevokeLifetimeToken()` 方法。



终结器详见第 13 章。

在把插件加载到新应用程序域的情形中，当不再需要插件时，可以删除加载的代码。如果不再需要这个插件，MAF 就使用一个简单的模型把一个插件指定为应用程序域的拥有者，来卸载应用程序域。如果在激活插件时创建应用程序域，这个插件就是应用程序域的拥有者。如果应用程序域是以前创建的，就不会自动卸载它。

在宿主端适配器中 `ContractHandle` 类用来增加插件的引用计数。这个类的成员如表 50-7 所示。

表 50-7

ContractHandle 类的成员	说 明
Contract	在 <code>ContractHandle</code> 类的构造过程中，可以指定一个实现 <code>Icontract</code> 接口的对象，来保存对它的引用。 <code>Contract</code> 属性返回这个对象
Dispose()	可以调用 <code>Dispose()</code> 方法，而不是等待垃圾回收器执行终结操作，来调用生命周期令牌
AppDomainOwner()	<code>AppDomainOwner()</code> 是 <code>ContractHandle</code> 类的一个静态方法，如果插件拥有通过该方法传递的应用程序域，该方法就返回插件适配器
ContractOwnsAppDomain()	使用静态方法 <code>ContractOwnsAppDomain()</code> ，可以验证指定的协定是否是应用程序域的拥有者。如果是，在删除协定时，就会卸载应用程序域

50.1.6 版本问题

版本问题是插件的一个大问题。宿主应用程序可以利用插件进一步开发。插件的一个要求是宿主应用程序的新版本仍可以加载插件的旧版本。旧宿主程序仍可以运行插件的新版本。那么，协定该如何修改呢？

`System.AddIn` 完全独立于宿主应用程序和插件的实现，这通过包含 7 部分的管道概念实现。

50.2 插件示例

下面是一个宿主应用程序的简单示例，它可以加载计算器插件。插件支持其他插件提供的不同计算操作。

我们需要创建一个解决方案，它包含 6 个库项目和一个控制台应用程序。示例应用程序的项目如表 50-8 所示。这个表列出了需要引用的程序集。在解决方案中引用其他项目，还需要把 Copy Local 属性设置为 False，这样程序集就不会复制。但 HostApp 控制台项目例外，它需要引用 HostView 项目。必须复制这个程序集，这样才能在宿主应用程序中找到它。另外，还需要更改所生成的程序集的输出路径，以便程序集复制到管道的正确目录下。

表 50-8

项 目	引 用	输 出 路 径	说 明
CalcContract	System.AddIn.Contract	.\Pipeline\ Contracts\	这个程序集包含与插件通信的协定。协定用接口定义
CalcView	System.AddIn	.\Pipeline\ AddInViews\	CalcView 程序集包含插件引用的一个抽象类，这是协定的插件端
CalcAddIn	System.AddIn.CalcView	.\Pipeline\AddIns\ CalcAddIn\	CalcAddIn 是引用插件视图程序集的插件项目。这个程序集包含插件的实现代码
CalcAddInAdapter	System.AddIn、 System.AddIn.Contract CalcView、 CalcContract、	.\Pipeline\ AddInSideAdapters\	CalcAddInAdapter 连接插件视图和协定程序集，并把协定映射到插件视图上
HostView			包含宿主视图的抽象类的程序集不需要引用任何插件程序集，也不引用解决方案中的其他项目
HostAdapter	System.AddIn、 System.AddIn.Contract、 HostView、 CalcContract、	.\Pipeline\ HostSideAdapters\	宿主适配器把宿主视图映射到协定上。因此需要引用这些项目
HostApp	System.AddIn、 HostView		宿主应用程序激活插件

50.2.1 插件协定

下面实现协定程序集。协定程序集包含一个协定接口，该接口定义了宿主和插件之间的通信协议。

下面的代码是为计算器示例应用程序定义的协定。应用程序给协定定义 GetOperations() 和 Operate() 方法。GetOperations() 方法返回计算器插件支持的一个数学操作列表。数学操作由

`IOperationContract` 接口定义, `IOperationContract` 接口本身就是一个协定。`IOperationContract` 接口定义只读特性 `Name` 和 `NumberOperands`。

`Operate()` 方法调用插件中的操作, 它需要 `IOperation` 接口定义的一个操作和通过 `double` 数组提供的操作数。通过这个接口, 插件就可以支持需要任意多个 `double` 操作数的操作, 且返回一个 `double`。`AddInStore` 类使用 `AddInContract` 属性生成缓存。`AddInContract` 属性把类标记为插件协定接口。



这里使用的协定的功能与第 28 章中插件的协定类似。但 MEF 协定没有 MAF 协定的限制。MAF 协定的限制很严, 因为宿主和插件之间有应用程序域边界和进程边界。而 MEF 体系结构没有使用不同的应用程序域。



可从
wrox.com
下载源代码

```
using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    [AddInContract]
    public interface ICalculatorContract: IContract
    {
        IListContract<IOperationContract> GetOperations();
        double Operate(IOperationContract operation, double[] operands);
    }
    public interface IOperationContract: IContract
    {
        string Name { get; }
        int NumberOperands { get; }
    }
}
```

代码段 CalcContract/ICalculatorContract.cs

50.2.2 计算器插件视图

插件视图重新定义了插件可以识别的协定。该协定定义 `ICalculatorContract` 和 `IOperationContract` 接口。为此, 插件视图定义了 `Calculator` 抽象类和 `Operation` 具体类。

在 `Operation` 中, 没有每个插件都需要的特定实现代码, 因为该类已经用插件视图程序集实现了。这个类用 `Name` 和 `NumberOperands` 属性描述数学计算的一个操作。

`Calculator` 抽象类定义需要由插件实现的方法。虽然协定定义了需要在应用程序域边界和进程边界之间传递的参数和返回类型, 但插件视图不需要。这里可以使用类型, 以便于插件开发人员编写插件。`GetOperations()` 方法返回 `IList<Operation>`, 而不是 `IListOperation<IOperationContract>`, 这与协定程序集不同。

`AddInBase` 属性把类标识为插件视图, 便于存储。



可从
wrox.com
下载源代码

```
using System.AddIn.Pipeline;
using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
    [AddInBase]
```

```

public abstract class Calculator
{
    public abstract IList<Operation> GetOperations();
    public abstract double Operate(Operation operation, double[] operand);
}
public class Operation
{
    public string Name { get; set; }
    public int NumberOperands { get; set; }
}
}

```

代码段 CalcView/CalculatorView.cs

50.2.3 计算器插件适配器

插件适配器把协定映射到插件视图上。这个程序集引用协定和插件视图程序集。适配器的实现代码需要把协定中的 `IListContract<IOperationContract>` 集合的 `GetOperations()` 方法映射到 `IList<Operation>` 集合的 `GetOperations()` 视图方法上。

该程序集包含 `OperationViewToContractAddInAdapter` 类和 `CalculatorViewToContractAddInAdapter` 类。这两个类实现 `IOperationContract` 和 `ICalculatorContract` 接口。`IContract` 基接口的方法可以通过派生自 `ContractBase` 基类来实现。这个基类提供默认的实现代码。`OperationViewToContractAddInAdapter` 类实现 `IOperationContract` 接口的其他成员，并把调用转发给在构造函数中指定的 `Operation View`。

`OperationViewToContractAddInAdapter` 类还包含静态辅助方法 `ViewToContractAdapter()` 和 `ContractToViewAdapter()`，前者把 `Operation` 映射到 `IOperationContract` 上，后者把 `IOperationContract` 映射到 `Operation` 上。



可从
wrox.com
下载源代码

```

using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    internal class OperationViewToContractAddInAdapter: ContractBase,
        IOperationContract
    {
        private Operation view;
        public OperationViewToContractAddInAdapter(Operation view)
        {
            this.view = view;
        }
        public string Name
        {
            get { return view.Name; }
        }
        public int NumberOperands
        {
            get { return view.NumberOperands; }
        }
        public static IOperationContract ViewToContractAdapter(Operation view)
        {
            return new OperationViewToContractAddInAdapter(view);
        }
        public static Operation ContractToViewAdapter(
            IOperationContract contract)
        {
        }
    }
}

```

```

        {
            return (contract as OperationViewToContractAddInAdapter).view;
        }
    }
}

```

代码段 CalcAddInAdapter/OperationViewToContractAddInAdapter.cs

CalculatorViewToContractAddInAdapter 类非常类似于 OperationViewToContractAddInAdapter 类：它派生自 ContractBase 基类，来继承 IContract 接口的默认实现代码，它还实现一个协定接口。但这里的 ICalculatorContract 接口用 GetOperations() 和 Operate() 方法实现。

适配器的 Operate() 方法调用 Calculator 视图类的 Operate() 方法，其中 IOperationContract 需要转换为 Operation。这是使用 OperationViewToContractAddInAdapter 类定义的静态辅助方法 ContractViewToAdapter() 完成。

GetOperations() 方法的实现需要把 IListContract<IOperationContract> 集合转换为 IList<Operation> 集合。对于这个集合转换，CollectionAdapters 类定义了转换方法 ToIList() 和 ToIListContract()。其中 ToIListContract() 方法用于这个转换。

AddInAdapter 属性把类标识为插件端适配器，用于插件存储器。



可从
wrox.com
下载源代码

```

using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    [AddInAdapter]
    internal class CalculatorViewToContractAddInAdapter: ContractBase,
        ICalculatorContract
    {
        private Calculator view;
        public CalculatorViewToContractAddInAdapter(Calculator view)
        {
            this.view = view;
        }
        public IListContract<IOperationContract>GetOperations()
        {
            return CollectionAdapters.ToIListContract<Operation,
                IOperationContract>(view.GetOperations(),
                OperationViewToContractAddInAdapter.ViewToContractAdapter,
                OperationViewToContractAddInAdapter.ContractToViewAdapter);
        }
        public double Operate(IOperationContract operation, double[] operands)
        {
            return view.Operate(
                OperationViewToContractAddInAdapter.ContractToViewAdapter(
                    operation), operands);
        }
    }
}

```

代码段 CalcAddInAdapter/OperationViewToContractAddInAdapter.cs



因为适配器类由.NET 反射调用,所以这些类可以使用内部的访问修饰符。因为这些类要具体实现,所以最好使用 `internal` 访问修饰符。

50.2.4 计算器插件

插件现在包含具体功能的实现代码。它用 `CalculatorV1` 类实现。插件程序集依赖于插件视图程序集,因为它需要实现 `Calculator` 抽象类。

`AddIn` 属性把类标记为插件,用于插件存储器,并添加发布者、版本和描述信息。在宿主端,这些信息可以从 `AddInToken` 类中访问。

`CalculatorV1` 在 `GetOperations()` 方法中返回一个支持的操作列表。`Operate()` 方法根据操作计算操作数。



可从
wrox.com
下载源代码

```
using System;
using System.AddIn;
using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
    [AddIn("CalculatorAddIn", Publisher="Wrox Press", Version="1.0.0.0",
        Description="Sample AddIn")]
    public class CalculatorV1: Calculator
    {
        private List<Operation> operations;
        public CalculatorV1()
        {
            operations = new List<Operation>();
            operations.Add(new Operation() { Name = "+", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "-", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "/", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "*", NumberOperands = 2 });
        }
        public override IList<Operation> GetOperations()
        {
            return operations;
        }
        public override double Operate(Operation operation, double[] operand)
        {
            switch (operation.Name)
            {
                case "+":
                    return operand[0] + operand[1];
                case "-":
                    return operand[0] - operand[1];
                case "/":
                    return operand[0] / operand[1];
                case "*":
                    return operand[0] * operand[1];
                default:
                    throw new InvalidOperationException(
                        String.Format("invalid operation {0}", operation.Name));
            }
        }
    }
}
```

```

    }
}

```

代码段 CalcAddIn/Calculator.cs

50.2.5 计算器宿主视图

下面看看宿主端的宿主视图。与插件视图类似，宿主视图也定义一个抽象类，其方法类似于协定。但是，这里定义的方法由宿主应用程序调用。

Calculator 和 **Operation** 类都是抽象类，因为其成员由宿主适配器实现。这两个类只需要定义宿主应用程序使用的接口：



```

using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
    public abstract class Calculator
    {
        public abstract IList<Operation> GetOperations();
        public abstract double Operate(Operation operation,
            params double[] operand);
    }
    public abstract class Operation
    {
        public abstract string Name { get; }
        public abstract int NumberOperands { get; }
    }
}

```

代码段 HostView/CalculatorHostView.cs

50.2.6 计算机宿主适配器

宿主适配器程序集引用宿主视图和协定，从而把视图映射到协定上。**OperationContractToViewHostAdapter** 类实现 **Operation** 抽象类的成员。**CalculatorContractToViewHostAdapter** 类实现 **Calculator** 抽象类的成员。

在 **OperationContractToViewHostAdapter** 类中，在构造函数中指定对协定的引用。适配器类还包含一个 **ContractHandle** 实例，该实例添加了对协定生命周期的引用，这样只要宿主应用程序需要，插件就保持加载状态。



```

using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    internal class OperationContractToViewHostAdapter: Operation
    {
        private ContractHandle handle;
        public IOperationContract Contract { get; private set; }
        public OperationContractToViewHostAdapter(IOperationContract contract)
        {
            this.Contract = contract;
            handle = new ContractHandle(contract);
        }
    }
}

```

```

        public override string Name
        {
            get
            {
                return Contract.Name;
            }
        }
        public override int NumberOperands
        {
            get
            {
                return Contract.NumberOperands;
            }
        }
    }
    internal static class OperationHostAdapters
    {
        internal static IOperationContract ViewToContractAdapter(Operation view)
        {
            return ((OperationContractToViewHostAdapter)view).Contract;
        }
        internal static Operation ContractToViewAdapter(
            IOperationContract contract)
        {
            return new OperationContractToViewHostAdapter(contract);
        }
    }

```

代码段 HostAdapter/OperationContractToViewHostAdapter.cs

CalculatorContractToViewHostAdapter 类实现抽象宿主视图类 Calculator 的方法，并把调用转发给协定。同样，该类还有一个 ContractHandle 实例，它包含对协定的引用，这类似于插件端类型转换的适配器。但这次仅需要从插件适配器向宿主端的类型转换。

HostAdapter 属性把类标记为一个需要在 HostSideAdapters 目录下安装的适配器。



```

using System.Collections.Generic;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    [HostAdapter]
    internal class CalculatorContractToViewHostAdapter: Calculator
    {
        private ICalculatorContract contract;
        private ContractHandle handle;
        public CalculatorContractToViewHostAdapter(ICalculatorContract contract)
        {
            this.contract = contract;
            handle = new ContractHandle(contract);
        }
        public override IList<Operation> GetOperations()
        {
            return CollectionAdapters.ToIList<IOperationContract, Operation>(
                contract.GetOperations(),
                OperationHostAdapters.ContractToViewAdapter,

```

```

        OperationHostAdapters.ViewToContractAdapter);
    }
    public override double Operate(Operation operation, double[] operands)
    {
        return contract.Operate(OperationHostAdapters.ViewToContractAdapter(
            operation), operands);
    }
}
}
}

```

代码段 HostAdapter/OperationContractToViewHostAdapter.cs

50.2.7 计算器宿主

示例宿主应用程序使用 WPF 技术。这个应用程序的用户界面如图 50-3 所示。其顶部是可用的插件列表。左边是活动插件的操作。选择要调用的操作时，就会显示操作数。输入操作数的值后，就可以调用插件的操作。

底部一行的按钮用于重建和更新插件存储器，以及退出应用程序。

下面的 XAML 代码显示了用户界面的树型结构。在 `ListBox` 元素中，给项模板使用不同的样式，为插件列表、操作列表和操作数列表指定特定的表示方式。

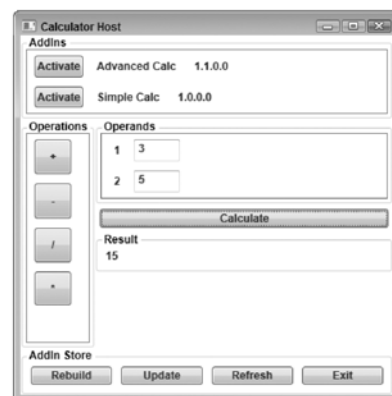


图 50-3



项模板的内容可参见第 35 章。



可从
wrox.com
下载源代码

```

<DockPanel>
  <GroupBox Header="AddIn Store" DockPanel.Dock="Bottom">
    <UniformGrid Columns="4">
      <Button x:Name="rebuildStore" Click="RebuildStore"
        Margin="5"> Rebuild</Button>
      <Button x:Name="updateStore" Click="UpdateStore"
        Margin="5">Update</Button>
      <Button x:Name="refresh" Click="RefreshAddIns"
        Margin="5"> Refresh</Button>
      <Button x:Name="exit" Click="App_Exit" Margin="5">Exit</Button>
    </UniformGrid>
  </GroupBox>
  <GroupBox Header="AddIns" DockPanel.Dock="Top">
    <ListBox x:Name="listAddIns" ItemsSource="{Binding}"
      Style="{StaticResource listAddInsStyle}"/>
  </GroupBox>
  <GroupBox DockPanel.Dock="Left" Header="Operations">
    <ListBox x:Name="listOperations" ItemsSource="{Binding}"
      Style="{StaticResource listOperationsStyle}"/>
  </GroupBox>
  <StackPanel DockPanel.Dock="Right" Orientation="Vertical">
    <GroupBox Header="Operands">

```

```

        <ListBox x:Name="listOperands" ItemsSource="{Binding}"
            Style="{StaticResource listOperandsStyle}">
        </ListBox>
    </GroupBox>
    <Button x:Name="buttonCalculate" Click="Calculate" IsEnabled="False"
        Margin="5">Calculate</Button>
    <GroupBox DockPanel.Dock="Bottom" Header="Result">
        <Label x:Name="labelResult" />
    </GroupBox>
</StackPanel>
</DockPanel>

```

代码段 HostAppWPF/CalculatorHostWindow.xaml

在代码隐藏中，在 Window 的构造函数中调用 FindAddIns() 方法。FindAddIns() 方法使用 AddInStore 类获得 AddInToken 对象的集合，把它们传递给 listAddIns 列表框的 DataContext 属性，以显示它们。AddInStore.FindAddIns() 方法的第一个参数传递宿主视图定义的 Calculator 抽象类，从存储器中查找应用于协定的所有插件。第二个参数传递从应用程序配置文件中读取的管道目录。运行 Wrox 下载网站(<http://www.wrox.com>)上的示例应用程序时，需要修改应用程序配置文件中的目录，以匹配自己的目录结构。



可从
wrox.com
下载源代码

```

using System;
using System.AddIn.Hosting;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Windows;
using Wrox.ProCSharp.MAF.Properties;
namespace Wrox.ProCSharp.MAF
{
    public partial class CalculatorHostWindow: Window
    {
        private Calculator activeAddIn = null;
        private Operation currentOperation = null;
        public CalculatorHostWindow()
        {
            InitializeComponent();
            FindAddIns();
        }
        void FindAddIns()
        {
            try
            {
                this.listAddIns.DataContext =
                    AddInStore.FindAddIns(typeof(Calculator),
                        Settings.Default.PipelinePath);
            }
            catch (DirectoryNotFoundException ex)
            {
                MessageBox.Show("Verify the pipeline directory in the " +
                    "config file");
                Application.Current.Shutdown();
            }
        }
    }
}

```

```

    }
    //...

```

代码段 HostAppWPF/CalculatorHostWindow.xaml

要更新插件存储器的缓存，UpdateStore()和 RebuildStore()方法应映射到 Update 和 Rebuild 按钮的 Click 事件上。在这些方法的实现代码中，使用 AddInStore 类的 Update()或 Rebuild()方法。如果程序集存储在错误的目录下，这些方法就返回一个警告字符串数组。由于管道结构比较复杂，因此第一次把程序集复制到正确的目录下时，其项目配置不太可能完全正确。阅读这些方法返回的信息，可以清楚地了解出错原因。例如，“在程序集\Pipeline\AddInSideAdapters\CalcView.dll 中没有找到可用的 AddInAdapter 部件”消息表示，CalcView 程序集存储在错误的目录下。

```

private void UpdateStore(object sender, RoutedEventArgs e)
{
    string[] messages = AddInStore.Update(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
            "AddInStore Warnings", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}
private void RebuildStore(object sender, RoutedEventArgs e)
{
    string[] messages =
        AddInStore.Rebuild(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
            "AddInStore Warnings", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}

```

在图 50-3 中，可看到插件的旁边有一个 Activate 按钮。单击这个按钮会调用处理程序方法 ActivateAddIn()。在这个方法的实现代码中，使用 AddInToken 类的 Activate()方法激活插件。其中插件加载到用 AddInProcess 类创建的一个新进程中。AddInProcess 类启动 AddInProcess32.exe 进程。把该进程的 KeepAlive 属性设置为 false，只要最后一个插件引用被垃圾回收了，该进程就停止。AddInSecurityLevel.Internet 参数使插件在有限的权限下运行。ActivateAddIn()方法的最后一条语句调用 ListOperations()方法，ListOperations()方法又调用插件的 GetOperations()方法。GetOperations()方法把返回的列表赋予 listOperations 列表框的 DataContext 属性，用于显示所有操作。

```

private void ActivateAddIn(object sender, RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "ActivateAddIn invoked from the wrong " +
        "control type");

    AddInToken addIn = el.Tag as AddInToken;
    Trace.Assert(el.Tag != null, String.Format(
        "An AddInToken must be assigned to the Tag property " +
        "of the control {0}", el.Name));
}

```

```

        AddInProcess process = new AddInProcess();
        process.KeepAlive = false;

        activeAddIn = addIn.Activate<Calculator> (process,
            AddInSecurityLevel.Internet);
        ListOperations();
    }
    void ListOperations()
    {
        this.listOperations.DataContext = activeAddIn.GetOperations();
    }

```

激活插件并在 UI 上显示操作列表后,用户就可以选择操作。在 Operations 类别中 Button 的 Click 事件赋予处理程序方法 OperationSelected()。在这个方法的实现代码中,检索赋予 Button 的 Tag 属性的 Operation 对象,获得操作所需要的操作数个数。为了允许用户给操作数添加值,把一个 OperandUI 对象数组绑定到 listOperands 列表框上。

```

private void OperationSelected(object sender, RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "OperationSelected invoked from " +
        "the wrong control type");
    Operation op = el.Tag as Operation;
    Trace.Assert(el.Tag != null, String.Format(
        "An AddInToken must be assigned to the Tag property " +
        "of the control {0}", el.Name));
    currentOperation = op;
    ListOperands(new double[op.NumberOperands]);
}
private class OperandUI
{
    public int Index { get; set; }
    public double Value { get; set; }
}
void ListOperands(double[] operands)
{
    this.listOperands.DataContext =
        operands.Select((operand, index) =>
            new OperandUI()
            { Index = index + 1, Value = operand }).ToArray();
}

```

通过 Calculate 按钮的 Click 事件调用 Calculate()方法。这里从 UI 中检索操作数,把操作和操作数传递给插件的 Operate()方法,结果显示为标签的内容:

```

private void Calculate(object sender, RoutedEventArgs e)
{
    OperandUI[] operandsUI = (OperandUI[])this.listOperands.DataContext;
    double[] operands = operandsUI.Select(opui => opui.Value).ToArray();
    labelResult.Content = activeAddIn.Operate(currentOperation,
        operands);
}

```

50.2.8 其他插件

现在完成了艰苦的工作。创建了管道组件和宿主应用程序。管道现在可以正常工作，还可以轻松地在宿主应用程序中添加其他插件，例如，下面的 Advanced Calculator 插件：



可从
wrox.com
下载源代码

```
[AddIn("Advanced Calc", Publisher = "Wrox Press", Version = "1.1.0.0",  
      Description = "Another AddIn Sample")]  
public class AdvancedCalculatorV1: Calculator
```

代码段 AdvancedCalcAddIn/AdvancedCalculator.cs

50.3 小结

本章学习了新颖的.NET 3.5 技术的概念：Managed Add-In Framework。

MAF 使用管道概念使宿主程序集和插件程序集的创建完全独立。清楚定义的协定把宿主视图和插件视图分开。适配器可以使宿主端和插件端独立地修改。

下一章介绍如何通过.NET Enterprise Services 使用.NET 组件。

第 51 章

Enterprise Services

本章内容:

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

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

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



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

51.1 使用 Enterprise Services

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

之后,概述这种技术提供的不同服务,以便探讨应用程序可以使用的功能。

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

51.1.1 简史

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

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

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

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

51.1.2 使用 Enterprise Services 的场合

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

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

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

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

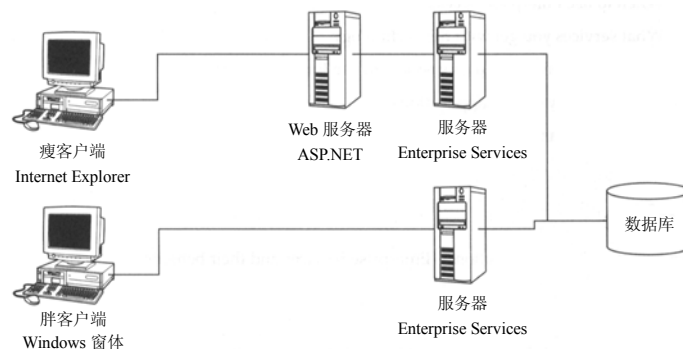


图 51-1

51.1.3 重要功能

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

1. 环境

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

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

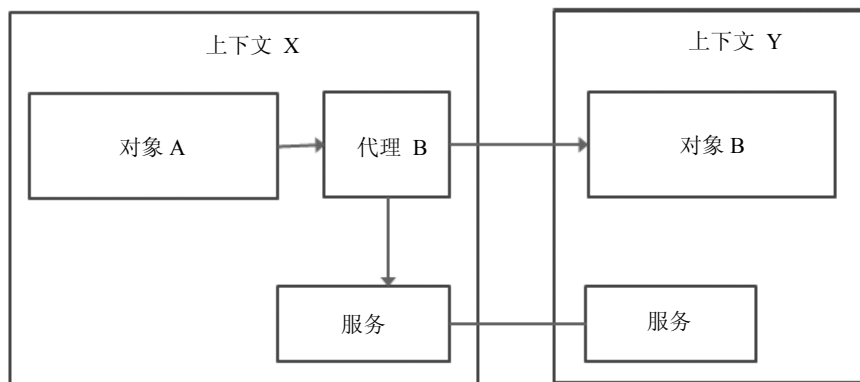


图 51-2

2. 自动事务处理

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

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

3. 分布式事务处理

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

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

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



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

本章后面将讨论如何创建需要事务处理的组件。

4. 对象池

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

5. 基于角色的安全性

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

6. 队列组件

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

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

7. 松散耦合事件

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

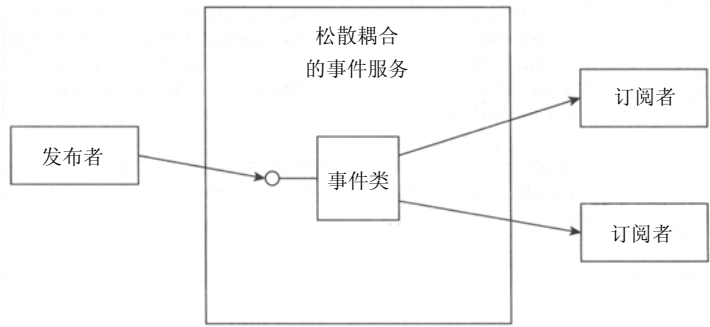


图 51-3

51.2 创建简单的 COM+应用程序

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

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

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

51.2.1 ServicedComponent 类

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

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

表 51-1

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

51.2.2 程序集的属性

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

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

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

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

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

表 51-2

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

51.2.3 创建组件

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



可从
wrox.com
下载源代码

```
using System.Runtime.InteropServices;

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

代码段 SimpleServer/IGreeting.cs

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

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



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

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

代码段 SimpleServer/SimpleComponent.cs

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

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

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

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

表 51-3

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

51.3 部署

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

51.3.1 自动部署

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

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

51.3.2 手动部署

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

```
regcvcs SimpleServer.dll
```

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

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



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

51.3.3 创建安装软件包

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

要创建安装软件包,可以启动组件服务管理

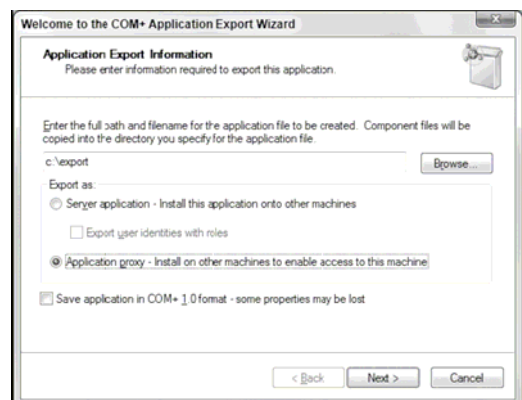


图 51-4

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



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

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

51.4 组件服务管理器

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

通过这个应用程序还可以设置其他一些选项：

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

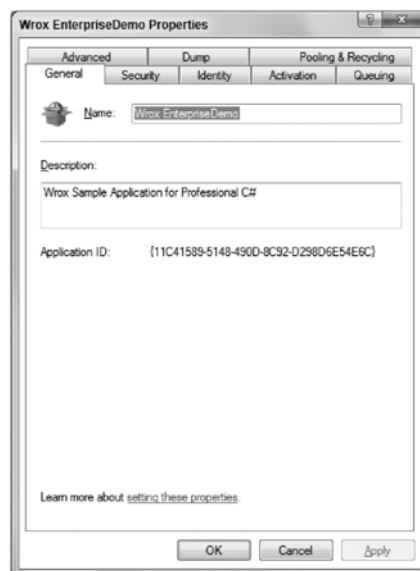


图 51-5

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

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

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

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

使用这个对话框，可以配置如下选项：

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

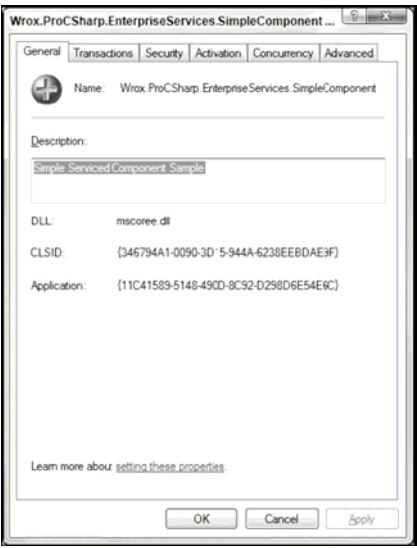


图 51-6

51.5 客户端应用程序

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

 可从
wrox.com
下载源代码

```
using System;

namespace Wrox.ProCSharp.EnterpriseServices
{
```

```

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

```

代码段 ClientApplication/Program.cs

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

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

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

51.6 事务

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



事务的概念详见第 23 章。

51.6.1 事务的特性

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

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

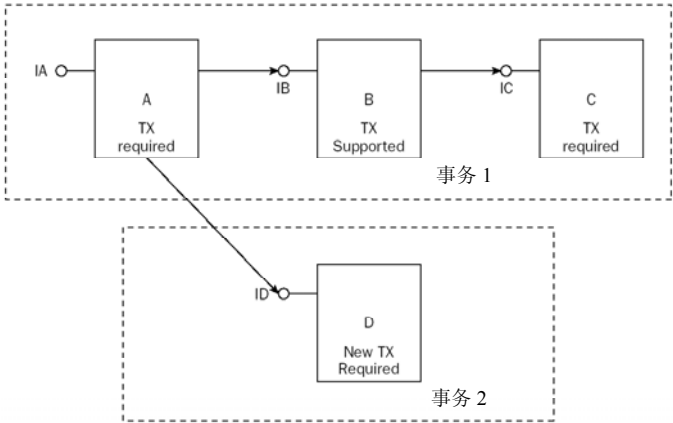


图 51-7

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

表 51-4

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

51.6.2 事务的结果

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

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

表 51-5

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

EnableCommit()	true	false
DisableCommit()	false	false

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

51.7 示例应用程序

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

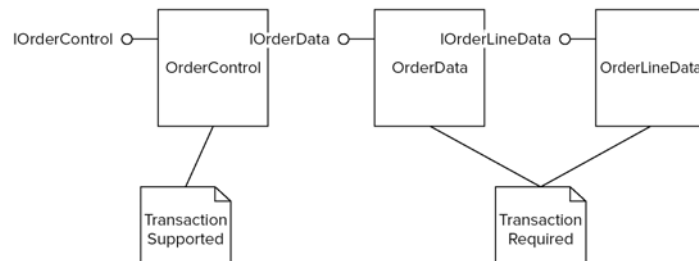


图 51-8

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



可从
wrox.com
下载源代码

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

代码段 NorthwindComponents/AssemblyInfo.cs

51.7.1 实体类

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



可从
wrox.com
下载源代码

```
using System;
```

```

using System.Collections.Generic;

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

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

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

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

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

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

```

代码段 NorthwindComponents/Order.cs

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



E112 可从
wrox.com
下载源代码

```
using System;
```

```

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

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

```

代码段 NorthwindComponents/OrderLine.cs

51.7.2 OrderControl 组件

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



可从
wrox.com
下载源代码

```

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

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

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

```

```

        data.Insert(order);
    }
}
}
}

```

代码段 NorthwindComponents/OrderControl.cs

51.7.3 OrderData 组件

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



可从
wrox.com
下载源代码

```

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

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

```

代码段 NorthwindComponents/OrderData.cs

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

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

```

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

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

```

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

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

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

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

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

    connection.Open();

    command.ExecuteNonQuery();
```

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

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

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

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

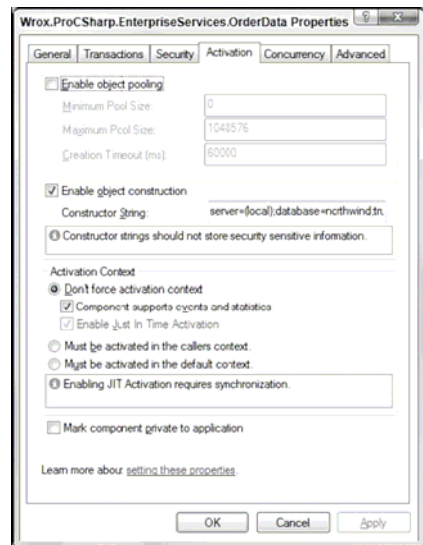


图 51-9

最后，无论 try 块中的代码成功执行，还是出现异常，都会关闭连接。

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

51.7.4 OrderLineData 组件

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



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

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

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

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

代码段 NorthwindComponents/OrderLineData.cs

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

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

```

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

        connection.Open();

        command.ExecuteNonQuery();

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

```

51.7.5 客户端应用程序

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

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



```

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

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

```

代码段 ClientApplication/Program.cs

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

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



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



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

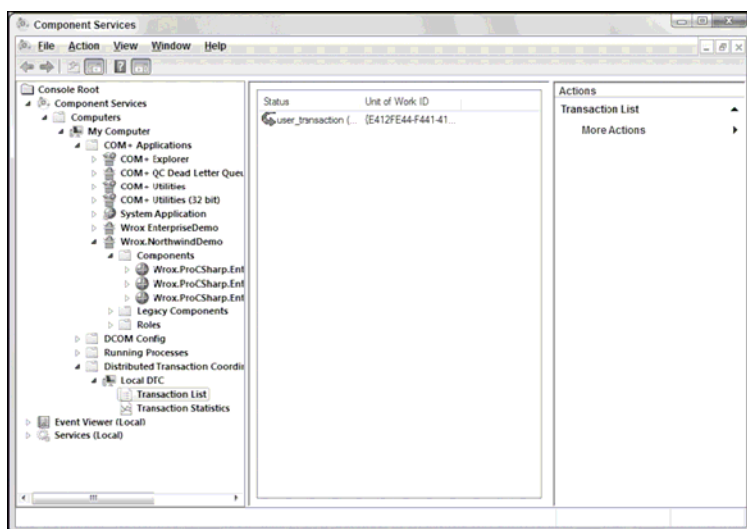


图 51-10

51.8 集成 WCF 和 Enterprise Services

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

51.8.1 WCF 服务外观

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

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



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

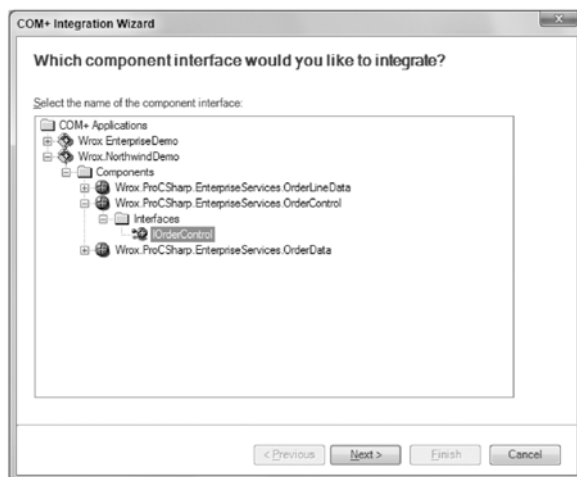


图 51-11

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

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

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

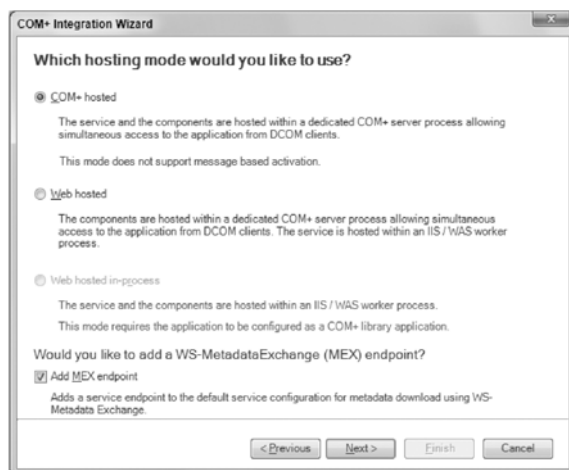


图 51-12

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



MEX 参见第 43 章。

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

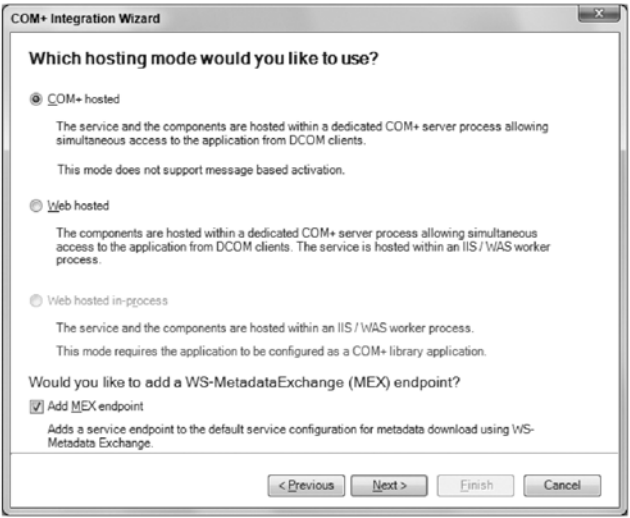


图 51-13

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

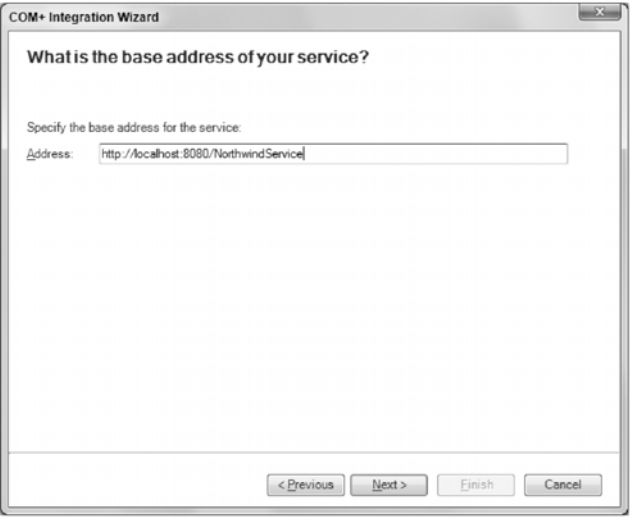


图 51-14

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

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

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

```

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

```

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

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

51.8.2 客户端应用程序

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

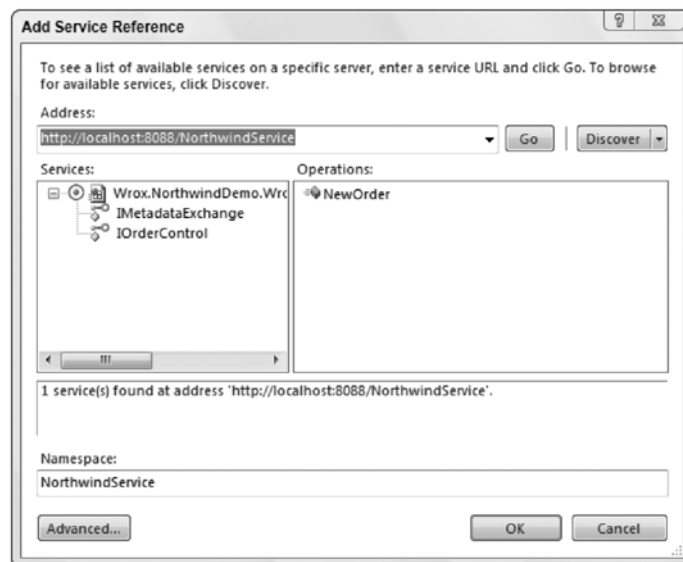


图 51-15



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

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

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

```

static void Main()
{

```