COM 高手总结的八个经验和教训

在日常工作中，我看到过许多由不同开发人员编写的 COM 代码。我为许多富于创造性的使用 COM 的工作方式感到惊讶，有一些使 COM 工作的巧妙代码可能连 Microsoft 都没有想到。同样，看到一些错误一次又一次地重犯，使我免不了心灰意懒。这些错误很多都与线程和安全有关，完全不成比例，而这也正是 COM 文档资料中最缺少的两个领域。如果不仔细计划，它们也是最可能遇到的并可能会绊住您的两个领域。

在下面的篇幅中，您将读到八位程序员的记述，这些教训都来自他们的痛苦经历。每个故事都是真实的，但为了保护无辜者，名字都已隐去。我的目的是，通过这些真实的 COM 故事，使您不再重蹈其他 COM 程序员的覆辙。它们还可能会帮助您在编写的代码中找出存在潜在问题的地方。无论情况如何，我想您都会获得愉快的阅读体验。

总是调用 CoInitialize(Ex)

几个月前，我收到了一封朋友的电子邮件，他就职于一家著名的硬件公司。他的公司编写了一个非常复杂的基于 COM 的应用程序，其中使用了许多进程内和本地（进程外）的 COM 组件。在开始时，应用程序创建了 COM 对象以服务于运行在多线程单元 (MTA) 中的各种客户端线程。该对象还可以托管给 MTA，这意味着接口指针可以在客户端线程之间自由交换。在测试中，我的朋友发现在应用程序准备关闭之前，一切都进行得不错。然后，不知是什么原因，对 Release 的调用（必须执行此调用，以便正确释放客户端占用的接口指针）被锁定了。他的问题是：“到底是哪里出了问题？”

其实答案非常简单。应用程序的开发人员其他都做得很对，只有一点例外，而这点又非常重要：他们没有在所有的客户端线程中调用 CoInitialize 或 CoInitializeEx。现代 COM 的基本原则之一，就是每个使用 COM 的线程都应该先调用 CoInitialize 或 CoInitializeEx 来初始化 COM。这条原则是无法免除的。除了其他事情以外，CoInitialize(Ex) 应将线程放入单元中，并初始化重要的每线程状态信息（这对于 COM 的正确操作是必需的）。调用 CoInitialize(Ex) 失败通常会在应用程序生命期早期以失败的 COM API 函数的形式表现出来，最常见的是激活请求。但有时问题很隐蔽，直到一切都太晚了（例如对 Release 的调用一去不复返了）才表现出来。当开发小组将 CoInitialize(Ex) 调用添加到所有接触 COM 的线程之后，他们的问题就迎刃而解了。

具有讽刺意义的是，Microsoft 竟是 COM 程序员有时不调用 CoInitialize(Ex) 的原因之一。Microsoft 知识库中包含的一些文档中说，调用 CoInitialize(Ex) 对基于 MTA 的线程来说不是必需的（有关示例，请参阅文章 Q150777）。是的，在很多情况下，我们可以跳过 CoInitialize(Ex) 而不会出现问题。但是，这样是不应该的，除非您知道自己在干什么，并且可以绝对肯定自己不会受到负面影响。调用 CoInitialize(Ex) 是没有害处的，因此我建议 COM 程序员始终从某个与 COM 相关的线程中调用它。

不要在线程之间传递原始接口指针

我咨询的首批 COM 项目之一就涉及到一个包含 100,000 行代码的分布式应用程序，该程

序是由美国西海岸的一个大型软件公司编写的。该应用程序在多个机器上创建了数十个 COM 对象，并从客户端进程启动的背景线程中调用这些对象。开发小组遇到问题了，调用要么消失得无影无踪，要么在没有明显原因的情况下失败。他们给我演示的最惊人的症状是：当一个调用无法返回时，在同一台机器上启动其他支持 COM 的应用程序（包括 Microsoft Paint 等）会频繁导致这些应用程序被锁定。

检查他们的代码后发现，他们违反了 COM 并发的一个基本规则，就是说，如果一个线程要与另一个线程共享一个接口指针，它应首先封送该接口指针。如果有必要，封送接口指针可使 COM 创建一个新的代理（以及一个新的信道对象，将代理和存根结对），以允许从另一个单元向外调用。不通过封送而将原始接口指针（内存中的一个 32 位地址）传递给另一个线程，会绕过 COM 的并发机制，并且如果发送和接收的线程位于不同的单元中，将出现各种不良行为。（在 Windows 2000 中，由于两个对象可以共享一个单元，但又位于不同的上下文中，因此如果线程位于同一个单元中，可能会使您陷入困境。）典型的症状包括调用失败和返回 RPC_E_WRONG_THREAD_ERROR。

Windows NT 4.0 和更高版本可以使用一对名为 CoMarshalInterThreadInterfaceInStream 和 CoGetInterfaceAndReleaseStream 的 API 函数，在线程之间轻松地封送接口指针。假定您应用程序中的一个线程（线程 A）创建了一个 COM 对象，继而接收了一个 IFoo 接口指针，并且同一进程中的另一个线程（线程 B）想调用这个对象。在准备将接口指针传递给线程 B 时，线程 A 应该封送该接口指针，如下所示：

```
CoMarshalInterThreadInterfaceInStream (IID_IFoo, pFoo, &pStream);
```

在 CoMarshalInterThreadInterfaceInStream 返回后，线程 B 就可以安全地取消封送该接口指针：

```
IFoo* pFoo;
CoGetInterfaceAndReleaseStream (pStream, IID_IFoo, (void**) &pFoo);
```

在这些示例中，pFoo 是一个 IFoo 接口指针，pStream 是一个 IStream 接口指针。COM 在调用 CoMarshalInterThreadInterfaceInStream 时初始化 IStream 接口指针，然后在 CoGetInterfaceAndReleaseStream 内部使用和释放该接口指针。实际上，您通常要使用一个事件或其他同步化基元来协调这两个线程的行为？例如，让线程 B 知道接口指针已准备好，可以取消封送。

请注意，以这种方式封送接口指针不会出现任何问题，因为 COM 有足够的智能，在不需要进行封送时不会去封送（或重新封送）指针。如果在线程之间传递接口指针时这样做，使用 COM 就轻松多了。

如果调用 CoMarshalInterThreadInterfaceInStream 和 CoGetInterfaceAndReleaseStream 看起来太麻烦，您还可以通过将接口指针放在全局接口表 (GIT) 中，并让其他线程去那里检索它们，从而实现在线程之间传递接口指针。从 GIT 中检索到的接口指针在被检索时会自动封送。更多信息，请参阅 IGlobalInterfaceTable 中的文档。请注意，GIT 只存在于 Windows NT 4.0 Service Pack 4 及更高版本中。

STA 线程需要消息循环

上一部分中描述的应用程序还有另一个致命缺陷。看看您是否能指出来。

这个特殊的应用程序恰好是用 MFC 编写的。在一开始，它使用了 MFC 的 AfxBeginThread 函数启动一系列辅助线程。每个辅助线程要么调用 CoInitialize 要么调用 AfxOleInit（MFC 中类似 CoInitialize 的函数）来初始化 COM。某些辅助线程则调用 CoCreateInstance 来创建 COM 对象，并将所返回的接口指针封送到其他辅助线程。从创建这些对象的线程中调用对象将非常顺利，但从其他线程的调用却从不返回。您知道这是为什么吗？

如果您认为问题与消息循环（或缺少消息循环）相关，那么答案完全正确。事实确实如此。当一个线程调用 CoInitialize 或 AfxOleInit 时，它是放在单线程单元 (STA) 中。当 COM 创建一个 STA 时，它会创建一个随附的隐藏窗口。以 STA 中的对象为目标的方法调用将转换为消息，并放入与该 STA 关联的窗口的消息队列中。当运行在该 STA 中的线程检索到代表方法调用的消息时，隐藏窗口的窗口过程就会将消息转换回方法调用。COM 使用 STA 执行调用序列化。STA 中的对象一次不可能接收一个以上的调用，因为每个调用都要传递给一个而且是惟一一个运行在对象单元中的线程。

如果基于 STA 的线程无法处理消息会怎么样呢？如果它没有消息循环又会怎么样呢？针对该 STA 中对象的单元间方法调用将不再返回；它们将在消息队列中被永远搁置。MFC 辅助线程中没有消息循环，因此如果寄宿在这些 STA 中的对象要从其他单元的客户端接收方法调用，那么 MFC 辅助线程和 STA 是配合不好的。

这个故事的寓意何在呢？STA 线程需要消息循环，除非您肯定它们不会包含要从其他线程调用的对象。消息循环可以像这样简单：

```
MSG msg;
while (GetMessage (&msg, 0, 0, 0))
DispatchMessage (&msg);
```

另一种方案是将 COM 线程移到 MTA 中（或者在 Windows 2000 中，移到中立线程单元，即 NTA 中），这里没有消息队列依赖项。

单元模型对象必须保护共享数据

另一个困扰 COM 开发人员的通病是标记为 ThreadingModel=Apartment 的进程内对象。这项指定告诉 COM，对象的实例必须只能在 STA 中创建。它还可让 COM 自由地将这些对象实例放在任何主机进程的 STA 中。

假设客户端应用程序有五个 STA 线程，每个线程都使用 CoCreateInstance 来创建同一个对象的一个实例。如果线程是基于 STA 的，且对象标记为 ThreadingModel=Apartment，则这五个对象实例将在对象创建者的 STA 中创建。因为每个对象实例都在占用其 STA 的线程上运行，因此所有五个对象实例都可以并行运行。

到目前为止，一切良好。现在考虑一下，如果这些对象实例共享数据会发生什么情况。因为对象都在并发线程上执行，两个或更多的对象可能会同时尝试访问同一个数据。除非所有这些访问都是读取访问，否则就会酿成灾难。问题可能不会很快显现出来；它们会以和时间紧密相关的错误形式出现，因此很难诊断和重现。这就解释了以下事实的原因：ThreadingModel=Apartment 对象应该包括可同步对共享数据的访问的代码，除非您能够确定对象的客户端不会对执行访问的方法进行重叠调用。

问题在于，太多的 COM 开发人员相信 ThreadingModel=Apartment 能够使他们免于编写线程安全的代码。事实并非如此 ？ 至少不完全如此。ThreadingModel=Apartment 并不意味着对象必须是完全线程安全的，它代表的是一个对 COM 的承诺，即访问两个或更多对象实例共享的数据（或此对象和其他对象的实例共享的数据）时是以线程安全的方式进行的。而提供该线程安全性的任务应该由您，即对象实现者来负责。共享数据的类型和大小多种多样，但大多是以全局变量、C++ 类中的静态成员变量和函数中声明的静态变量的形式出现。即使是以下这样无害的语句也会在 STA 中出问题：

```
static int nCallCount = 0;
nCallCount++;
```

因为这个对象的所有实例都将共享一个 nCallCount 实例，编写这些语句的正确方式如下：

```
static int nCallCount = 0;
InterlockIncrement (&nCallCount);
```

注意：您可以使用临界区、互锁函数或您希望的任何方式，但不要忘了访问基于 STA 的对象共享的数据时要进行同步化！

谨慎启动用户

这里还有一个问题让许多 COM 开发人员都吃过苦头。去年春天，有一家公司向我紧急呼救，他们的开发人员使用 COM 构建了一个分布式应用程序，其中客户端进程运行在与远程服务器的 Singleton 对象相连接的网络工作站上。在测试过程中，他们遇到了一些非常奇怪的行为。在一种测试场景中，客户端对 CoCreateInstanceEx 的调用可使它们与 Singleton 对象正常连接。而在另一个场景中，对 CoCreateInstanceEx 的相同调用产生了多个对象实例和多个服务器进程，使客户端无法与同一个对象 实例连接，从而实际影响了应用程序。在这两个场景中，硬件和软件是完全相同的。

此问题似乎与安全有关。当处理远程激活请求的 COM 服务控制管理器 (SCM) 在另一台机器上启动一个进程时，它会为该进程分配一个标识。除非另外指定，它选择的标识就是启动用户的标识。换句话说，分配给服务器进程的标识与启动它的客户端进程的标识相同。在这种情况下，如果 Bob 登录机器 A，并使用 CoCreateInstanceEx 连接机器 B 上的 Singleton 对象，而 Alice 也在机器 C 上如法炮制，就会启动两个不同的服务器进程（至少在两台不同的 WinStation 上），实际上使客户端无法再用 Singleton 语义与共享的对象实例连接。

两个测试场景之所以会产生大相径庭的结果，其原因就是在一个场景（那个可以工作的场景）

中，所有测试人员都使用只为测试而设置的一个特殊帐户以同一个人的身份登录。而在另一个场景中，测试人员都使用他们的普通用户帐户登录。当两个或更多的客户端进程具有相同标识时，它们可以成功连接到配置为假定启动用户标识的服务器进程。但是，如果客户端有不同的标识，SCM 会使用多个服务器进程（每个唯一客户端标识一个）分隔分配给不同对象实例的标识。

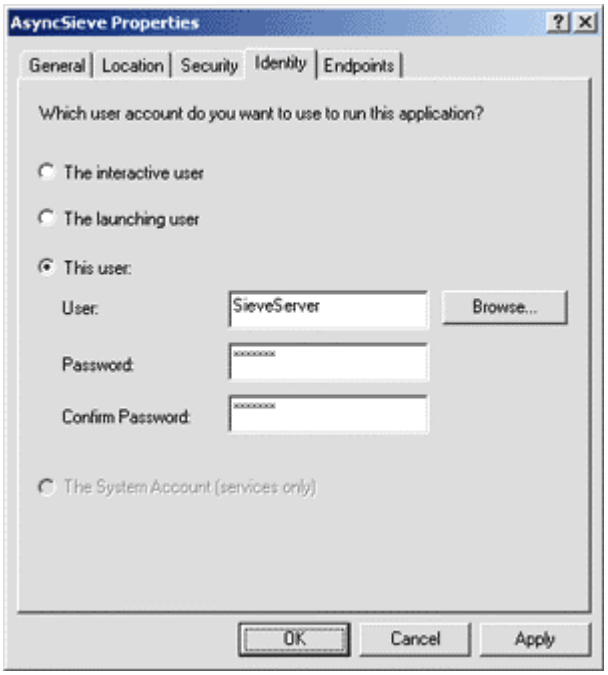

图 1 DCOMCNFG 中的用户帐户

找到问题以后，解决起来就很简单了：配置 COM 服务器，让其使用特定的用户帐户而不是假定启动用户的标识。完成这一任务的一种方式是在服务器机器上运行 DCOMCNFG（Microsoft 的 DCOM 配置工具），并将“launching user ”更改为“This user”（请参见图 1）。如果您喜欢通过编程方式进行更改（可能从安装程序着手），请在主机注册表的 HKEY_CLASSES_ROOT\AppID 部分的 COM 服务器项中添加 RunAs 值（请参见图 2）。

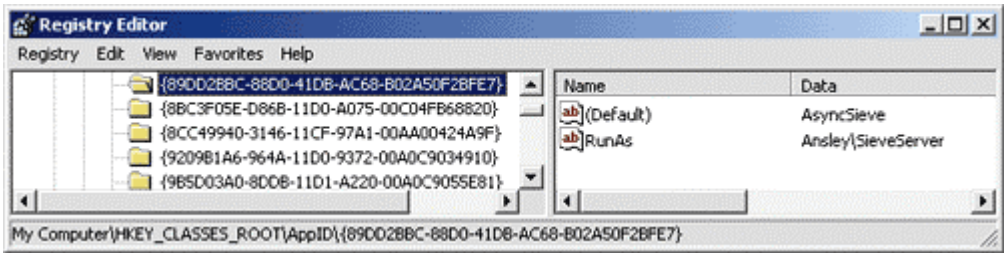

图 2 添加 RunAs 值到注册表中

您还需要使用 LsaStorePrivateData 将 RunAs 帐户的密码存储为 LSA 密钥，并使用 LsaAddAccountRights 确保帐户拥有“Logon as batch job”的权限。（有关具体操作的示例，请参见 Platform SDK 中的 DCOMPERM 示例。请特别注意名为 SetRunAsPassword 和 SetAccountRights 的函数。）

DCOM 不适于防火墙

关于 DCOM 特性和功能的一个常见问题是："它能跨 Internet 工作吗？"DCOM 能够很好地跨 Internet 工作，只要将它配置为使用 TCP 或者 UDP，并且通过授予任何人启动和访问权限，可将服务器配置为允许匿名方法调用。毕竟，Internet 是一个巨大的 IP 网络。但矛盾的是，如果您将一个现有的 DCOM 应用程序（在公司的内部网络或 intranet 中工作得很好）改为跨 Internet 工作，它很有可能失败得很惨。可能是什么原因呢？防火墙。

DCOM 生来与防火墙的关系就如油与水的关系。原因之一是 COM 的 SCM 使用端口 135 与其他机器上的 SCM 通信。防火墙限制了它可以使用的端口和协议，可能会拒绝通过端口 135 传入的通信量。但更大的问题在于，为了避免与使用套接字、管道和其他 IPC 机制的应用程序冲突，DCOM 没有固定使用特定范围的端口，相反，它在运行时才选择所使用的端口。默认情况下，它可以使用从 1,024 到 65,535 范围内的任何端口。

允许 DCOM 应用程序通过防火墙的一种方式是，为 DCOM 要使用的协议打开端口 135 和端口 1,024-65,535。（默认情况下，Windows NT 4.0 是 UDP 协议，Windows 2000 是 TCP 协议。）但是，这比移除所有防火墙好不了多少。对此，您公司的 IT 人员可能要发表意见了。

另一种更安全和更现实的解决方案是，限制 DCOM 使用的端口范围，并只为 DCOM 通信量打开一组小范围端口。根据实践原则，您应该为每个服务器进程分配一个端口，将连接导出到远程 COM 客户端（不是每个接口指针一个端口或每个对象一个端口，而是每个服务器进程一个）。将 DCOM 配置为使用 TCP 而不是 UDP 是一个好方法，特别是在服务器对其客户端执行回调时。

DCOM 用于远程连接的端口范围和所用的协议可通过注册表进行配置。在 Windows 2000 和 Windows NT 4.0 Service Pack 4 或更高版本上，您可以用 DCOMCNFG 应用这些配置更改。以下是将 DCOM 配置为通过防火墙工作的办法。

图 3 选择协议

在服务器（在防火墙后寄存远程对象的机器）上，将 DCOM 配置为使用 TCP 作为其所选协议，如图 3 中所示。

在服务器上，限制 DCOM 将使用的端口范围。记住为每个服务器进程至少分配一个端口。图 4 中的示例将 DCOM 限制为端口 8,192 到 8,195。

打开您在步骤 2 中选择的端口，使 TCP 通信量能够通过防火墙。同时打开端口 135。

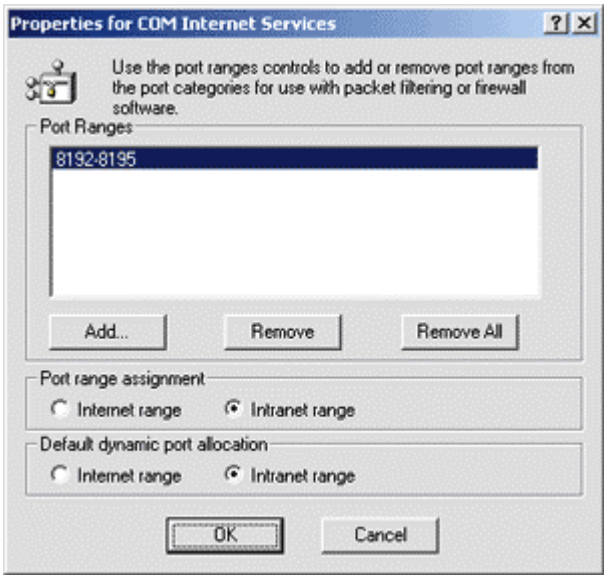

图 4 选择端口

执行这些步骤，DCOM 就可以很好地跨防火墙工作了。如果您愿意，SP4 和更高版本还可让您为单独的 COM 服务器指定终结点。更多信息，请阅读 Michael Nelson 关于 DCOM 和防火墙的优秀论文，该论文可在 MSDN Online 站点上找到（请参见 http://msdn.microsoft.com/library/enus/dndcom/html/msdn_dcomfirewall.asp）。

还应注意的是，通过在服务器上安装 Internet 信息服务 (IIS)，并使用 COM Internet 服务 (CIS) 通过端口 80 路由 DCOM 通信量，SP4 和更高版本的用户还可以使用 CIS 来提供与防火墙兼容的 DCOM。有关该主题的更多信息，请参阅 http://msdn.microsoft.com/library/en-us/dndcom/html/cis.asp。

使用线程或异步调用来避免 DCOM 超时设定太长

总是有人问我当 DCOM 无法完成远程实例化请求或方法调用时出现的超时设定太长的问题。典型的场景如下：客户端调用 CoCreateInstanceEx 来实例化远程机器上的一个对象，但是这台机器临时离线了。在 Windows NT 4.0 上，激活请求不会立即失败，DCOM 可能会花上一分钟或更长时间来返回失败的 HRESULT。DCOM 还可能花费很长时间，使指向已不再存在或其主机已离线的远程对象的方法调用失败。如果可能，开发人员应该如何避免这些较长的超时设定呢？

要回答这个问题，几句话是讲不清楚的。DCOM 高度依赖于基础网络协议和 RPC 子系统。并没有什么神奇的设置可让您限制 DCOM 超时设定的持续时间。但是，我经常使用两种技巧来避免较长超时设定的负作用。

在 Windows 2000 中，当调用在 COM 信道中挂起时，您可以使用异步方法调用来释放调用线程。（有关异步方法调用的介绍，请参 MSDN Magazine 2000 年 4 月刊的“ Windows 2000: Asynchronous Method Calls Eliminate the Wait for COM Clients and Servers Alike”。如果异步调用在合理时间内没有返回，您可以通过调用用于初始化调用的调用对象上的 ICancelMethodCalls::Cancel 来取消它。

Windows NT 4.0 不支持异步方法调用，甚至在 Windows 2000 中也不支持异步激活请求。怎么解决呢？从背景线程调用远程对象（或是实例化该对象的请求）。使主线程在事件对象上阻塞，并指定超时设定值以反映您愿意等待的时间长度。当调用返回时，让背景线程来设置事件。假设主线程使用 WaitForSingleObject 阻塞，当 WaitForSingleObject 返回时，返回值可以告诉您，返回是因为方法调用或激活请求返回，还是因为您在 WaitForSingleObject 调用中指定的超时设定到期。您不能在 Windows NT 4.0 中取消挂起调用，但是至少主线程可以自由地执行自己的任务。

下面的代码演示了基于 Windows NT 4.0 的客户端如何才能从背景线程调用对象。

```
//////////////////////////////////////////////////
```

```
CloseHandle (g_hEvent);
```

在此示例中，线程 A 封送了一个 IFoo 接口指针，并启动线程 B。线程 B 取消封送了该接口指针，并调用 IFoo::Bar。无论调用返回所花费的时间有多长，线程 A 都不会阻塞超过 5 秒钟，因为它在 WaitForSingleObject 的第二个参数中传递的是 5,000 （单位为微秒）。这并不是太好的办法，但是如果“无论在线路的另一端发生什么情况，线程 A 都不会挂起”这一点很重要的话，忍受这种麻烦也算值得。

共享对象并不容易

从我收到的邮件和在会议上被问到的问题判断，困扰许多 COM 程序员的一个问题是如何将两个或更多的客户端与一个对象实例连接。要回答这个问题，写出长篇大论（或是一本小册子）都很容易，但其实只要说明与现有对象的连接既不容易也不自动化，就足够了。COM 提供了大量创建对象的方式，包括很受欢迎的 CoCreateInstance(Ex) 函数。但是 COM 缺乏一种通用的命名服务，允许使用名称或 GUID 来标识对象实例。而且它没有提供内置的方式来创建对象，然后将它标识为调用的目标以检索接口指针。

这是不是意味着将多个客户端与单一对象实例连接就不可能了呢？当然不是。实现这一点有五种方式。在这些资源链接中，您可以找到更多信息甚至是示例代码，来指导您的操作。请注意，这些技术从一般意义上讲不能互换；通常，环境因素会决定哪种方式（如果有）适用于手边的任务： Singleton 对象 Singleton 对象就是只实例化一次的对象。可能会有 10 个客户端调用 CoCreateInstance 来“创建”Singleton 对象，但实际上，它们都是接收指向同一对象的接口指针。ATL COM 类可通过在其类的声明中添加 DECLARE_CLASSFACTORY_SINGLETON 语句，来转换为 Singleton。

文件名字对象 如果一个对象实现了 IpersistFile，并在运行中对象表 (ROT) 中使用文件名字对象（它封装了传递给对象的 IPersistFile::Load 方法的文件名称）注册了自己，那么客户端就可以使用文件名字对象连接对象的现有实例了。实际上，文件名字对象允许使用文件名称来命名对象实例，对象可在这些文件名称中存储它们的持久性数据。它们甚至能够跨机器工作。

CoMarshalInterface 和 CoUnmarshalInterface 保存接口指针的 COM 客户端可以与其他客户端共享这些接口指针，只要它们愿意封送指针。COM 为愿意将接口指针封送给同一进程中其他线程的线程提供了优化（请参见教训 2），但是如果客户端线程属于其他进程，CoMarshalInterface 和 CoUnmarshalInterface 就是实现接口共享的关键途径了。有关讨论和示例代码，请参阅 MSJ 1999 年 8 月刊中我的超酷代码专栏。