

本书由Meji制作,于广州,2002年11月.

注意,本书的一切所有权及版权归译者,著者及中国电力出版社所有,一切由本电子版引发之民事刑事纠纷与本人无关.本电子书的制作纯粹处于传播知识之目的,禁止将本电子书用作盈利之目的,若有恣意妄为者后果自负.

# 译序

关于这本书，我最想说的一句话是，它来得太迟了。

自 1993 年 COM 首次发布以来，COM 本身经历了重大的变化，但是它的基本思想和结构一直保持相对稳定，这也正说明了 COM 思想的魅力所在。尽管如此，由于 COM 的神秘本质，并不是每个人都能够很容易地理解 COM 的思想，理解 COM 的魅力。幸运的是，现在有了这本书，它会清楚地告诉你这一切是如何发生的。

本书无疑是 COM 领域著作中最负盛名的一本。两年前我从 MSDN Library 上看到这本书的第三章和第五章，当时的感觉是内容非常零乱，所以对于这本书的盛名非常不解。后来抽时间阅读了 amazon.com 上读者和专家们的评价之后，才意识到这本书对于提高 COM 的理解是多么重要。这次翻译使我有机会仔细地阅读本书的每一个细节，虽然是在原书出版三年之后阅读本书，但是其中的内容现在仍然很有参考价值。

这不是一本供新手学习的书，也不是一本教你如何开发 COM 组件的书，它的目的是帮助你在一定的基础上继续提高对 COM 的理解。如果用它来学习 COM，你可能会失望，因为许多内容需要有一定的经验基础才能看得明白；但是如果用它来加深自己对 COM 的理解，那么这本书再合适不过了，读完这本书之后，COM 对于你会“变得再明白不过”（这也正是《Inside OLE》的作者 Kraig Brockschmidt 和本书的作者 Don Box 在顿悟了 COM 和 OLE 之后的境界）。期望你会有同样的感受。

COM 是跨语言的组件对象模型，但是它与 C++ 和面向对象思想的渊源可谓深矣。本书会向你解释这些内容，但前提条件是你必须具备扎实的 C++ 基础知识，特别要理解 C++ 类的二进制结构以及类型转换机制。反过来，阅读本书也可以加强你在 C++ 和面向对象思想方面的理解。此外，本书还涉及到许多 COM 高级特性，例如线程模型（或者套间模型）、名字对象、连接点等，因此，要想从本书获得最大受益，你也需要有 COM 的基本知识做基础，我建议读者先系统地学习 COM 之后再阅读本书（可以参看《COM 技术内幕》或者《COM 原理与应用》）。

自 1997 年以来 COM 一直是 Microsoft 技术的热点，但是 COM 的热潮似乎随着 Microsoft .net 的推出而冷却了下来。实际上，COM 在 Microsoft Windows 平台上永远有它的一席之地，目前没有任何迹象表明要被其他的结构替换，它将与传统的 SDK API

一起在底层提供各种服务。在新的.net 结构中, COM+继承了 COM 所有的优势, 并对 COM 重新做了包装, 只是感觉上 COM 与应用层的距离更远一些, 因此, 本书中讲到的几乎所有内容都适用于未来的应用系统。如果读者对于 Windows 2000 中的 COM 感兴趣的话, 可以参阅 David Chappell<sup>1</sup> 撰写的《深入理解 Microsoft Windows 2000 分布式服务》和 David S.Platt 撰写的《深入理解 COM+》。

在开始翻译这本书之前, 我有幸得到了台湾著名电脑技术作家侯俊杰先生赠送的繁体版《COM 本质论》(侯俊杰译), 这为我的翻译工作提供了很大的帮助。而且我决定采用同样的书名, 以保持两个中文版本的一致性。但是本书的叙述风格仍然采用我在其他 COM 书籍中的做法, 并且主要术语也保持一致。在此我特别感谢侯先生所做的一切, 当然也要感谢 Don Box 为我们写了这么好的一本书。

潘爱民

2001 年 2 月于北京大学燕北园

---

<sup>1</sup> 《Understanding ActiveX and OLE 》一书的作者, 这是另一本重量级的 COM 著作, 可惜没有中译本。

# Charlie Kindel 序

---

当我坐下来写这篇序言的时候，我的脑子里有好几个想法：

- Don 的照片会出现在封底吗？如果是这样的话，他的头发会有多长？
- 本书的读者是否知道 Don 有一张非常个性化的车牌，上面写着“IUNKNWN”？
- 一个人在一本书的序言里到底能写些什么呢？

关于到底要在这篇序言里写些什么，我有两个想法。第一，我想写下很久以来就一直想写的、关于 COM 设计思想的内容；第二个想法是，就像 Don 为了让我高兴而请我写这篇序文一样，我也使劲捧一捧他。最后，我决定两样都做。

什么是 COM？为什么要发明 COM？Don 在第 1 章简洁地回答了这些问题。在介绍部分的最后，他这样说：“……本章展示了一个可重用的组件结构，这个结构允许多个独立开发的二进制组件能够动态、有效地组成一个复合系统。”这一章的其他部分带领读者走过 COM 设计者在 1988 年到 1993 年期间经历的过程，COM 最终是在 1993 年首次发布的。

我认为 COM 取得如此长久的成功，要归因于几个方面的因素。第一也是最重要的是实用性，然后是简单性，从而产生灵活性，或称可塑性。

## 实用性

COM 以一种非常务实的方式来处理软件设计。它不是根据学院派的面向对象编程理论来提供解决方案，而是充分考虑了人的自然因素以及资本的因素。经典面向对象理论中最好的、又最具商业价值的思想被设计小组挑选出来，并且与 Microsoft 内外在以前的软件工程中所获得的有关软件重用的经验结合在一起。

大多数经典的面向对象课本都认为，只要一个系统或者一种语言支持封装性（信息隐藏）、多态性和继承性，那么它就是面向对象的。继承性通常被强调为“实现重用性的主要工具”。但是 COM 的设计者并不同意这种观点。他们认为，这是一种过于简单的看法，实际上应该存在两种继承性。实现继承（implementation inheritance）表示真正

的实现（即行为）被继承；而接口继承（interface inheritance）则表示只有行为的规范被继承。正是后一种继承形式实现了多态性。COM 完全支持这种形式的继承。另一方面，实现继承只不过是重用现有实现的一种简单机制。然而，如果重用是最终目标的话，那么实现继承只是通向目标的简单途径，但不是目标本身。

无论是研究领域还是商业软件开发领域，人们普遍认为，实现继承是一种有效的、功能强大的工具，但是它也会导致基类和派生类之间的过分关联（或称耦合）。由于实现继承往往会造成基类实现细节的“泄漏”，从而违反了基类的封装特性，所以 COM 的设计者感觉到，应该把实现继承局限于组件内部的开发过程中。虽然 COM 不支持跨越组件的实现继承，但是它支持组件内部的实现继承。而且 COM 也完全支持接口继承（事实上，它完全依赖于接口继承）。

COM 设计者并没有特别强调继承本身，继承通常被认为是获得重用性的一条有效途径。在 COM 中，用于构造重用性的基础概念是封装，而不是继承。相反，COM 通过继承建立起功能相拟的对象之间的类型关系模型。在封装基础上建立起 COM 的重用模型之后，COM 设计者鼓励使用一种被称为“黑盒子”的重用，这种重用性能够符合预期中的组件市场。这种思想是，客户应该把对象看作不透明的组件，不管其内部是什么，或者它们是如何被实现的。COM 设计者相信，应该设计出一种结构来加强这种思想。为什么要用不同的重用模型来设计一个系统呢？问得好。事实上，这个世界充满了“没有使用黑盒子封装形式”的面向对象系统，甚至这些系统要实现这种黑盒子封装都非常困难。C++就是一个典型例子，在本书的第 1 章，Don 把我在这里所说的想法描述得非常清楚。

下面的等式说明了面向对象程序设计和面向组件程序设计之间的区别：

面向对象程序设计 = 多态性 + (某些) 迟绑定 + (某些) 封装性 + 继承

Object-Oriented Programming = Polymorphism + (Some) Late Binding + (Some)  
Encapsulation + Inheritance

面向组件程序设计 = 多态性 + (完全) 迟绑定 + (完全，也是强制性的) 封装性 +  
接口继承 + 二进制重用性

Component-Oriented Programming = Polymorphism + (Really) Late Binding +  
(Real, Enforced) Encapsulation + Interface Inheritance  
+ Binary Reuse

对我来说，在任何一种情况下，所有的争论都是非常有趣的。经常出现在 comp.object 和 comp.object.corba 的面向对象纯粹主义者会不屑地指着 COM 说“它并不是真正的面向对象”。你可以用以下两种方法来反驳这种观点：

1. “它当然是面向对象的。只不过你对面向对象的定义是错误的。”
2. “那又怎么样？COM 在商业上获得了巨大的成功，它使成千上万的独立软件开发人员通过插入、集成的手段建立起大型的软件系统。而且他们还赚到了钱，赚到

了不少钱。<sup>1</sup> 他们编写的软件组件在销售、在使用、在重复使用。这不正是任何一种技术的真正要点吗？除此之外，我还可以反驳说唯有 COM 才是真正面向组件的。<sup>2</sup> 确实如此！”

## 简单性产生可塑性

**malleable** 可塑的。形容词。它的意思是：

1. 能够在重击或者重压之下改变形状或者结构，譬如 a malleable metal (可塑金属)。
2. 容易被控制，或者受影响；易控制的 (tractable)。
3. 能够调整自身以适应环境的变化；易适应的 (adaptable)。譬如，the malleable mind of the pragmatist (实用主义者灵活的头脑)。<sup>3</sup>

COM 的第一个真正应用是作为 Microsoft 的复合文档结构第二个版本，也就是 OLE (Object Linking & Embedding, 对象链接和嵌入) 2.0 的底层基础。如果你能够考虑到今天 COM 的众多应用方式，那么你一定能够明白我所说的“COM 具有可塑性”的含义。开发人员利用 COM 为自己的应用提供插件结构、建立大规模商务交易的多层客户/服务器结构、在 Web 页面中提供点击服务、控制和监视制造过程，甚至通过远程控制望远镜阵列以跟踪间谍卫星。

这样的可塑性来自于这样的事实：COM 的设计者使它的模型核心尽可能的简单。可说明这一点的一个证据是 COM 程序设计相对比较“蹩脚”。C 或者 C++ 开发人员必须要处理各种“黏性物质”，包括 GUID (全局唯一标识符) 和引用计数。然而所有的复杂性都可以融入到 COM 中，并达到简化的目的。COM 设计师并不把重点放在让这个模型运转起来，他们知道只要他们成功了，那么各种支持工具就会随之而来。最近问世的快速易用 COM 工具使这样的假设得到了证实，这些工具包括 Visual Basic、Visual C++ 中的 COM 支持，以及 ATL (Active Template Library, 活动模板库)。当你阅读这篇序文的时候，Microsoft 已经宣布，未来计划通过一个公共的运行时程序库 COM+ (所有的工具都可使用) 来简化 COM 开发。

## 大众趣闻

任何一项像 COM 这样被广泛使用的技术一定会有许多趣闻。纯粹出于好玩，这里列出一些趣闻，你可能没有听到过它们，但有一些确实是真的。

<sup>1</sup> Giga (受 Microsoft 的委托) 的研究报告表明，在 1997 年基于 COM 的商业组件有 4.1 亿美元的市场。据估计，到 2001 年，这个市场将增加到 28 亿美元。这些数字并没有包括 Microsoft 自己的产品。

<sup>2</sup> 给读者一个练习：请你说出除了 COM 之外的、可供商业使用的对象系统，要求这个系统提供二进制重用模型、支持稳定的版本控制策略、位置透明性以及编程语言独立性。如果你回答说 CORBA，那么你已经受骗了，只不过你还不知道而已。

<sup>3</sup> 摘自《The American Heritage Dictionary of the English Language, Third Edition》(1996 年由 Houghton Mifflin 公司出版)。电子版本由 INSO Corporation 授权；进一步的复制和发行应遵循美国的版权法。

- Microsoft 有许多开发组的人都对 COM 的设计工作做出了贡献，但是 COM 最主要的设计师则是 Bob Atkinson、Tony Williams 和 Craig Wittenberg。他们三人仍然在 Microsoft 工作。
- Bob、Tony 和 Craig 属于一个跨部门的小组，这个小组专门致力于提交各种有关“比尔·盖茨的 IAYF (Information at Your Fingertips, 信息尽在掌握)”远景成真的核心技术。<sup>4</sup> 虽然这三个人对 COM 未来的威力非常清楚，但是他们却被束缚在“尽快推出 COM 的实用产品 OLE 2.0”上。这也说明了为什么经历如此长时间才形成 COM 文档。抱歉。
- COM 的第一个实现随 1993 年 5 月 OLE 2.0 的发布而问世。
- 根接口（当时并不被称作 IUnknown）有一个方法为 GetClassID。这个方法被移到 IPersist 接口中了，这也正好说明了“保持 COM 模型尽可能简单”的原则。
- 在当时，IUnknown 并没有 AddRef 方法。但是很快 COM 设计者看清楚了，不允许使用者复制接口指针将带来很大的限制。
- IUnknown 中的“Unkown”源自 Tony Williams 在 1988 年 12 月撰写的 Microsoft 内部报告，其标题为“Object Architecture: Dealing with the Unknown – or – Type Safety in a Dynamically Extensible Class Library”。
- 1991 年的前两个月才作出了“使用 RPC 作为进程间远程通信机制”的决定。Bob Atkinson 写了一份备忘录，名为“*IAYF Requirements for RPC*”，记下了所有的要求，并交给 RPC 小组，该小组后来被改名为“*IAYF 小组*”。这个小组负责建立起使“比尔·盖茨的 IAYF”远景成真的基础。这个基础就是 COM（只不过当时不是这个叫法）。
- 名字对象 (moniker) 的功能比你想像的要强大得多。
- 有些人把 COM 称作“Common Object Model”，这是 Mark Ryland 的错。他深感懊悔，并愿意致以最诚挚的歉意。
- “MEOW”并不真的代表 Microsoft Extensible Object Wire。这是 Rick 虚构出来的。
- Windows NT 3.5 包括第一个 32 位 COM 和 OLE 版本。有人意外地在某个（根）头文件中留下了“#pragma optimization off”。真是的！

如果有一本关于 COM、DCOM、OLE 或者 ActiveX 的（英文）书，并且我也读过的话，那么你几乎总是可以在技术评论者一栏中找到我的名字，以此作为该书的信誉保证。我自己也写了许多有关这些技术的文章，并且我也是 COM 规范的主要编辑。我给技术人员和非技术人员作过无数的关于 COM 的报告。很显然，我花了大量的时间和精力来找出表达 COM 的最佳方法。

---

<sup>4</sup> 你可以回顾一下比尔·盖茨于 1990 年在 Comdex 大会上关于 IAYF 的演讲。

当我读完这本书的草稿之后，我明白我的所有努力都白费了。没有人能比 Don Box 把 COM 阐释得更清楚。

我希望你能够像我一样享受这本书。

Charlie Kindel  
COM 专家, Microsoft 公司  
1997 年 9 月

# Grady Booch 序

---

有时候，关于一本书的好处太多了，它就值得说两次。这是为什么 Don 的书有两篇序的原因——它就是这样的一本好书。

如果在 Windows 95 或者 NT 上建立系统，那么你不可能躲开 COM。Visual Studio（特别是 Visual Basic）隐藏了 COM 的一些复杂性，但是如果你想真正了解后台所发生的事情，或者充分发掘 COM 的强大功能，那么 Don 的这本书很适合你。

让我特别喜欢这本书的一点是 Don 在说明 COM 时所采用的组织方式，摆在你面前的首先是构建分布式和并发性系统所面临的各种问题，然后向你展示 COM 是如何来处理这些问题的。即使你在读这本书之前对 COM 一无所知，你也会被 COM 中清晰而又简单的概念模型所导引，使你理解问题的症结所在，以及赋予 COM 结构和行为的力量所在。如果你是一位有经验的开发人员，那么你一定会特别赏识 Don 关于“用 COM 来解决常见问题”的各种论述。

对于开发分布式和并发性的系统而言，COM 是应用最为广泛的一种对象模型。本书将有助于你利用 COM 成功地建立起这样的系统来。

Grady Booch（译注）

---

译注：Grady Booch 是世界著名的软件工程大师，UML 语言的创始人之一，现为 Rational 公司首席科学家。

# 前 言

---

我的工作终于完成了。在终于把被许多人称之为“COM 的口述历史”写成一本书之后，我终于可以休息了。这本书反映了我自己对 Microsoft 于 1993 年展示给程序设计世界的这项恶梦般的技术的理解过程。虽然我没有参加最初的 OLE 专业开发人员研讨会，但是我却感到自己好像一直在从事 COM 有关的工作。四年来一直在与 COM 打交道，使我几乎不能记起在 COM 出现之前的计算时代。然而，我仍然清晰地记得 1994 年初我迁移到 COM 王国时的痛苦。

我几乎花了六个月的时间才对 COM 有所感觉、有所理解。在与 COM 朝夕共处的最初六个月时间里，我能够成功地编写 COM 程序，也总能够解释为什么这些程序能够工作。然而，我对于“为什么 COM 编程模型要被设计成这样”则不能够有整体上的理解。幸运的是，有一天（1994 年 8 月 8 日，在购买了《Inside OLE》一书大约半年之后），我突然有如得到神的启示一般，COM 对我变得再明白不过。这并不是说我理解了所有的接口和 API 函数，而是理解了隐藏在 COM 背后的动机。从这一刻起，“如何把 COM 编程模型应用到每天的编程工作中”对我也变得再明白不过。许多程序开发人员也有类似的体验。当我写这份前言的时候，又有三个 8 月过去了，但是开发人员仍然必须要走过这 6 个月的等待期，才能够成为 COM 社会中具有生产能力的一员。我期望这本书能够缩短这个等待周期，但是我不做任何承诺。

正如这本书所强调的，COM 更像是一门程序设计的学科，而不是一项技术。为此，我并不打算用每一个接口的每一个方法的每一个参数的细节描述来惊吓读者。相反，我尽量提取出 COM 的本质，让 SDK 文档来填补每一章留下的空隙。无论何地，只要有可能，我就尽量指出激发 COM 某一要素的底层张力（动机），而不是提供细节例子来展示如何把每个接口和 API 函数应用到人为构造的例子程序中。我自己的经验表明，只有理解了“为什么（why）”，那么关于“怎么做（how）”的问题自然迎刃而解。相反地，仅仅知道了“怎么做（how）”，则并不能够提供对文档背后内容的足够的洞察力。如果你希望跟上编程模型的演化与发展的话，那么这种洞察力非常重要。

对于建立一个分布式的面向对象系统而言，COM 是一个非常灵活的基础。为了充分发挥 COM 的灵活性，你必须经常超越 SDK 文档、文章或者书籍以外的限制。我个人的建议是，你可以假定所看到的任何事物（包括这本书）可能都是不正确的，或者已经过时了，然后设法形成你自己对编程模型的理解。理解编程模型最可靠的方法是抓住

COM 的基本词汇表，并精通这些词汇的语义。这恐怕只能通过“编写 COM 程序并分析这些程序为什么能够按预定的方式工作”才能够获得。阅读书籍、文章和文档当然有用，但是最终还是要将时间花在对 COM 四个核心概念（接口、类、套间和安全性）的思考上，这样才能使你成为一个有效的开发人员。

为了帮助读者专注于这些核心概念，我尽量包含尽可能多的代码，但是仅限于简单地提供这些源代码，而不作仔细推敲。为了确保 COM 编程技术也能够出现在文字之中，附录 B 包含一个完整的 COM 应用，它也是本书所讨论的众多概念的一个综合例子。而且，本书提供了一些可从网络下载的例子，其中包括一个 COM 工具代码库，这个代码库在我自己的开发工作中非常有用。这个库的某些部分在本书有详细的讨论，但是包含整个库，仅仅是为了演示如何构造标准的 C++ 实现。同时也请注意，每一章出现的许多代码都用到了 C 运行库的宏 assert 来强调某些必须满足的前后条件。在软件产品的代码中，这些 assert 语句大多应该用更宽大的错误处理代码来取代。

出版书籍的一个缺点是，当这本书到达各地书店时，它往往已经过时了。这本书也不例外。特别是，目前尚未定型的 COM+ 和 Windows NT 5.0（译注 1）必定使本书许多内容不正确或者不完整。我尽力预言 Windows NT 5.0 将会带来的有关模型的变化，然而，在我现在写这份前言的时候，Windows NT 5.0 尚未进入公开的 Beta 测试，所有的信息都有可能发生变化。COM+ 承诺将这个模型进一步发展；然而，我不可能既要把 COM+ 包括进来，同时仍能在今年交出书稿。我鼓励你在拿到了 Windows NT 5.0 和 COM+ 之后，认真地研究它们。

我必须做出一个痛苦的决定，如何在众多的商业库中选出一个库来实现 COM 组件（用 C++ 语言）。在观察了各个 Internet 讨论组上常见的问题之后，我决定放弃 ATL 和 MFC，而把焦点集中在每个开发人员必须要面对的 COM 主题上，不管他们使用了什么样的库都无法回避这些主题。越来越多的开发人员在产生了一大堆 ATL 代码之后，奇怪为什么这些代码不能正常工作。我坚信，一个人不可能在 ATL 和 MFC 的开发过程中学习 COM。这并不是说 ATL 和 MFC 不是开发 COM 组件的有效工具，我的意思是它们不适合用来演示或者学习 COM 编程概念和技术。这对于一本重点集中在 COM 编程模型的书籍来说，ATL 将显得很不合适。幸运的是，大多数开发人员发现，只要理解了 COM，那么短时期内就可以精通 ATL。

最后，每一章开始时的引语使我有机会写出我对书中这一部分的感觉。为了尽可能保持直接的写作风格，我限制这种狂野而跑题的故事在每一章不超过 15 行 C++ 代码。通常，这个引用代码代表了这一章将要面对的概念或者问题在 COM 之前的做法。或许读者根据这些代码可以分析出我根据这些提示写出这一章内容时的心态。

## 致谢

写作一本书非常艰难，至少对于我是这样的。与我一同受尽煎熬的两个人是我的妻

---

译注 1：现已改名 Windows 2000。

子 Barbara 和我的儿子 Max（尽管他年纪还很小，但是他喜欢 COM 更甚于其他的对象模型）。我要特别谢谢你们俩，当我努力写作时，能够容忍我的失陪，而且还要容忍我写作时的怪脾气。当本书完成了大部分之后，我的女儿出世了，她很幸运，一出生就有一个好脾气的父亲。同时我也要感谢 DevelopMentor 公司（译注 2）的全体同仁，在我全神贯注于本书写作过程中，你们帮助我完成了本该由我承担的许多事情。

许多关于分布式系统的初步想法早在九十年代初期就形成了，当时我在加州大学 Irvine 分校为 Tatsuya Suda 工作。Tatsuya 教会了我如何写作、如何阅读，如何对付东京不守规矩的火车旅客。非常感谢，也很抱歉。

感谢我的老同事 Doug Schmidt，把我引荐给《C++ Report》杂志的 Stan Lippman（译注 3）。尽管 Stan 拒绝了我的第一篇文章，但我从这里开始了写作生涯。谢谢两位。

感谢 Addison Wesley 公司的 Mike Hendrickson 和 Alan Feuer，是你们启动了这本书的工程。谢谢 Ben Ryan 和 John Wait，感谢你们的耐心。谢谢 Carter Shanklin 坚持到了工程的最后。

感谢《Microsoft Systems Journal》杂志的工作人员，在本书写作期间，你们一再容忍我延期递交文章。特别感谢 Joanne Steinhart、Gretchen Bilson、Dave Edson、Joe Flanigen、Eric Maffei、Michael Longacre、Joshua Trupin、Laura Euler 以及 Joan Levinson。我答应以后再也不会迟交了。

感谢 David Chappell 写了有关 COM 应用最好的一本书。我真诚地建议，每个人都应该买他的书，并且至少读它两遍。

感谢 CORBA 和 Java 的许多民间组织以及爱好者们，你们使我加入到许多 Usenet 新闻组上多年来的白热化论战中。你们的执著使我必须要更好地理解 COM。尽管事实上我仍然感到你们的许多观点有点华而不实，但是我还是尊重你们的愿望。

在过去几年中，Microsoft 有几个人对我的帮助很大，他们直接或者间接地帮助我写作这本书。Sara Williams 是我碰到的第一个来自 Microsoft 的 COM 成员，在说明了她自己并不认识 Bill 之后，于是她把我引荐给《Microsoft Systems Journal》的 Gretchen Bilson 和 Eric Maffei，以此作为补偿。Sara 永远是在大屋子里的“福音传教士”，我永远感谢她。Charlie Kindel 为这本书写下了一篇很好的序言，尽管他的工作日程总是排得满满的。Nat Brown 是第一个向我展示“套间（apartment）”的人，他用单词“schwing”无可挽回地玷污了我的词汇表。Kraig Brockschmidt 向我表述了 COM 中某个看起来很精致的方面，其实归根究底只是一个怪异的技巧而已。Dave Reed 把我引见给 Viper，并且每次我到 Redmond 时都认真听取我的想法。Pat Helland 花了 TechEd'97 整整一周的时间来扭转我的想法，并且强迫我重新检查我对 COM 的基本假设。Scott Robinson、Andreas Luther、Markus Horstmann、Mary Kirtland、Rebecca Norlander 以及 Greg Hope 使我走出了黑暗。Ted Hase 帮助我扩展了措辞。Rick Hill 和 Alex Armanasu 在技术前沿为我做了

---

译注 2：DevelopMentor 公司是 Dom Box 等软件开发大师组建的软件开发咨询公司。

译注 3：Stan Lippman 是名著《C++ Primer》的作者。该书中文版即将由中国电力出版社出版。

大量的支持工作。Microsoft 其他许多人成果也影响到了我的工作，这样的人包括 Tony Williams、Bob Atkinson、Craig Wittenberg、Crispin Goswell、Paul Leach、David Kays、Jim Springfield、Christian Beaumont、Mario Goertzel 和 Michael Montague。

DCOM 邮件列表所讨论的内容是本书最主要的思路来源，也是我灵感的源泉。特别要感谢以下的 DCOM 伙伴：声名狼藉的 Mark Ryland、COM 神童 Mike Nelson、Keith Brown、Tim Ewald、Chris Sells、Saji Abraham、Henk De Koning、Steve Robinson、Anton von Stratten 和 Randy Puttick。

本书的内容主要受到了我在过去几年中在 DevelopMentor 公司教授 COM 课程的影响。同时也得到了学生和其他授课教师的影响。尽管我个人非常希望能够感谢每一位学生（Addison Wesley 限制我的前言不得超过 20 页），不过这里我要感谢现在仍然在 DevelopMentor 的、曾经帮助我提炼出自己对 COM 的理解、并且在“Essential COM”课程中提供了宝贵的反馈信息的各位朋友：Ron Sumida、Fritz Onion、Scott Butler、Owen Tallman、George Shepherd、Ted Pattison、Keith Brown、Tim Ewald 和 Chris Sells。谢谢你们。也要感谢 DevelopMentor 的 Mike Abercrombie 营造了商业领域无法比拟的个人成长环境。

如果不是由于 Terry Kennedy 以及 Software AG 公司的朋友，这本书将可以更早出版。Terry 非常好，他邀请我在休息日帮助在德国的 DCOM/UNIX 方面的工作，我本打算利用这些休息日写作这本书。我无法拒绝 Terry 的邀请（这是我的错，不是 Terry 的错），所以本书晚了一年时间，但是我认为由于在此期间在这个工程上的工作，使这本书更加充实。特别是我在与以下人员的协同工作中获得了许多深层的理解：Harald Stidhl、Winnie Froehlich、Volker Denkhaus、Deitmar Gaertner、Jeff Lee、Deiter Kesler、Martin Koch、Blauer Aff、Uli Kaess、Steve Wild 和罪魁祸首 Thomas Volger。

特别要感谢下面的读者，他们在以前的印刷版本中发现了许多错误，他们是：Ted Neff、Dan Moyer、Purush Rudrakshala、Heng de Koneng、Dave Hale、George Reilly、Steve Delassus、Warren Young、Jeff Prosise、Richard Grimes、Barry Klawans、James Bowmer、Stephan Sas、Peter Zaborshi、Christopher L. Akerley、Robert Brooks、Jonathan Pryor、Allen Chambers、Timo Kettunen、Atulx Mohidekar、Chris Hyams、Max Rubinstein、Bradey Honsinger、Sunny Thomas、Gardner von Holt 和 Tony Bervilos。

最后，感谢 Shah Jehan 和可口可乐公司分别为我提供了可口的印度食品和软饮料，使我的工作能不断进行下去。

Don Box

美国加州

Redondo Beach

1997 年 8 月

<http://www.develop.com/dbox>

# 目 录

---

## 译 序

Charlie Kindel 序

Grady Booch 序

## 前 言

<b>第 1 章 COM 是一个更好的 C++ .....</b>	<b>1</b>
1.1 软件分发和 C++ .....	2
1.2 动态链接和 C++ .....	4
1.3 C++和可移植性 .....	5
1.4 封装性和 C++ .....	7
1.5 把接口从实现中分离出来 .....	10
1.6 抽象基类作为二进制接口 .....	12
1.7 运行时多态性 .....	18
1.8 对象扩展性 .....	20
1.9 资源管理 .....	26
1.10 我们走到哪儿了？ .....	29
<b>第 2 章 接口 .....</b>	<b>30</b>
2.1 再谈接口与实现 .....	30
2.2 IDL .....	32
2.3 方法和结果 .....	34
2.4 接口和 IDL .....	36
2.5 IUnknown .....	39
2.6 资源管理和 IUnknown .....	44
2.7 类型强制转换和 IUnknown .....	47
2.8 实现 IUnknown .....	50
2.9 使用 COM 接口指针 .....	55
2.10 优化 QueryInterface .....	57

2.11	数据类型 .....	60
2.12	IDL 属性和 COM 属性 .....	72
2.13	异常 .....	73
2.14	我们走到哪儿了？ .....	78
<b>第 3 章</b>	<b>类 .....</b>	<b>79</b>
3.1	再谈接口与实现 .....	79
3.2	类对象 .....	81
3.3	激活 .....	84
3.4	使用 SCM .....	86
3.5	类和服务器 .....	89
3.6	一般化(generalization) .....	96
3.7	优化(Optimization) .....	99
3.8	再论接口与实现 .....	105
3.9	名字对象和组合 .....	110
3.10	名字对象和永久性 .....	113
3.11	服务器生命周期 .....	116
3.12	类和 IDL .....	118
3.13	类模仿(class emulation) .....	121
3.14	组件类别 .....	123
3.15	我们走到哪儿了？ .....	128
<b>第 4 章</b>	<b>对象 .....</b>	<b>129</b>
4.1	再谈 IUnknown .....	130
4.2	QueryInterface 是对称的 .....	131
4.3	QueryInterface 是可传递的 .....	132
4.4	QueryInterface 是自反的 .....	134
4.5	对象具有静态类型 .....	136
4.6	唯一性和对象实体身份 .....	137
4.7	QueryInterface 和 IUnknown .....	138
4.8	多重接口和方法名字 .....	141
4.9	动态复合 .....	149
4.10	二进制复合 .....	155
4.11	包容 .....	165
4.12	我们走到哪儿了？ .....	166
<b>第 5 章</b>	<b>套间 .....</b>	<b>167</b>
5.1	再谈接口和实现 .....	167

5.2 对象、接口和套间 .....	170
5.3 跨套间访问 .....	173
5.4 进程内列集辅助函数 .....	179
5.5 标准列集结构 .....	183
5.6 实现接口列集器 .....	188
5.7 标准列集、线程和协议 .....	191
5.8 生命周期管理和列集 .....	197
5.9 自定义列集 .....	204
5.10 自由线程列集器 .....	209
5.11 我们走到哪儿了？ .....	217
<b>第 6 章 应用 .....</b>	<b>218</b>
6.1 进程内激活的缺陷 .....	218
6.2 激活和 SCM .....	219
6.3 再谈服务器生命周期 .....	224
6.4 应用 ID .....	229
6.5 COM 和安全性 .....	233
6.6 通过编程实现安全性 .....	240
6.7 访问控制 .....	250
6.8 令牌管理 .....	256
6.9 我们走到哪儿了？ .....	262
<b>第 7 章 杂项 .....</b>	<b>263</b>
7.1 指针基础 .....	264
7.2 指针和内存 .....	266
7.3 数组 .....	274
7.4 流程控制 .....	290
7.5 动态与静态调用 .....	294
7.6 双向接口协议 .....	298
7.7 IDL 中的别名技术 .....	310
7.8 异步方法 .....	314
7.9 我们走到哪儿了？ .....	314
<b>附录 A 对象技术的演变 .....</b>	<b>316</b>
<b>附录 B 代码摘录 .....</b>	<b>322</b>
COM Chat：一个基于 COM 的聊天程序 .....	322

# 第1章

## COM是一个更好的C++

```
template <class T, class Ex>
class list_t : virtual protected CPriateAlloc {
    list<T**> m_list;
    mutable TWnd m_wnd;
    virtual ~list_t(void);
protected:
    explicit list_t(int nElems,...);
    inline operator unsigned int *(void) const
        { return reinterpret_cast<int*>(this); }
template <class X> void clear(X& rx) const throw(Ex);
};

Anonymous, 1996
```

C++已经伴随我们走过很长一段时间了。C++程序员群体非常庞大，并且许多有关这门语言的陷阱和缺陷也都已经为人们所熟知。起初，C++得益于Bell实验室的一群文质彬彬的程序员，他们不仅创造了第一个C++开发产品（CFRONT），而且也公布了许多关于C++的核心工作。大多数C++经典书籍都出版于80年代后期以及90年代早期。在这段时间内，许多C++开发人员（包括几乎每一本重要C++书籍的作者）都在UNIX工作站平台上工作，并且利用当时的编译器和链接器技术建立起许多独立的应用。这一代开发人员所用的环境基本上奠定了C++社团的思考方向。

C++的一个主要目标是允许程序员建立用户自定义类型（UDT，user-defined type），

而且这些类型可以在原始实现环境之外被重复使用。这条原则巩固了我们今天所熟知的类库或者框架的基础。由于 C++ 的引入，C++ 类库的市场就出现了，只是速度有点慢。这个市场之所以不能按预期那样快速成长，原因在于 C++ 开发人员之中的 NIH (not-invented-here) 因素。重用其他开发人员的代码往往看起来比简单地重新实现这些代码要付出更多的代价。有时候，这种感觉完全是由于开发人员的傲慢。而另一些时候，对重用的抗拒则是因为精神上的负担：为了实现重用，必须要理解其他开发人员的设计和编程风格。对于封装风格的类库 (wrapper-style library) 尤其是这样，为了使用这种类库，使用者不仅需要理解封装在库内部的底层技术，还需要理解库本身强加的抽象概念。

许多库的文档都假定库的用户把库的源码当作最根本的参考材料，这就进一步加剧了这种倾向。这种白盒子形式的重用往往会导致客户应用和类库之间过分的耦合关系，也使整个代码随着时间的推移而变得更加脆弱。耦合效应也降低了类库的模块化特征，使客户很难适应底层库的变化。这使客户往往并不把类库当作一个可重用的模块化组件，而是工程源代码本身的一部分。实际上，开发人员往往修改商业类库的源代码，使它更加符合自己的需要，产生一个更加适合程序员应用的“私有 build 版本”，而不是真正的原始版本。

重用性一直是面向对象的主要动机之一。但是事实上，要编写一个很容易被重用的 C++ 类非常困难。C++ 领域中，除了设计时和开发时重用的许多障碍，在运行时也有大量的障碍，使 C++ 对象模型不能够成为“构建可重用软件组件”的理想底层基础。这些障碍主要来自 C++ 本身所假设的编译和链接模型。这一章将引导大家看一看，要把 C++ 类做成可重用的组件将会面临怎样的挑战。每一个挑战都将以当前可用的技术来展示。通过这些技术的规范应用，本章展示了一个可重用的组件结构，这个结构允许多个独立开发的二进制组件能够动态、有效地组成一个复合系统。

## 1.1 软件分发和 C++

为了理解“把 C++ 作为组件的基础底层结构”所带来的问题，我们来看一看在 80 年代后期 C++ 类库是如何被分发的，这对于我们的理解会非常有帮助。想像有一个库厂商，它已经开发了一个算法，可以在 O(1) 时间阶内完成子串搜索运算，O(1) 时间阶的意思是指搜索时间为常数，与目标串的长度没有正比关系。这往往被公认为是一个不算太琐碎的任务。为了使算法的用法尽可能地简单，这家软件厂商创建一个字符串类，客户可以利用该类中的算法快速地找到文字信息。为了做到这一点，软件商生成了一个头文件，其中包含如下的类定义：

```
// faststring.h /////////////////////////////////
class FastString {
```

```

char *m_psz
public:
    FastString(const char *psz);
    ~FastString(void);
    int Length(void) const;           // 返回字符数目
    int Find(const char *psz) const;  // 返回偏移量
};

```

除了类定义，软件商还将在另外一个单独的文件中实现类的成员函数：

```

// faststring.cpp /////////////////////////////////
#include "faststring.h"
#include <string.h>
FastString::FastString(const char *psz)
: m_psz(new char[strlen(psz)+1]) {
    strcpy(m_psz,psz);
}

FastString::~FastString(void) {
    delete[] m_psz;
}

int FastString::Length(void) const {
    return strlen(m_psz);
}

int FastString::Find(const char *psz) const {
    // O(1) 查找代码,为简明起见从略1
}

```

从传统意义上讲，C++库一直以源代码的形式分发。类库的用户可以把实现代码加入到他们的系统工程中，然后用他们的 C++ 编译器在本地重新编译库的源代码。假定类库只用到 C++ 编程语言中被广泛支持的一个子集，那么这将是一种非常可行的做法。这么做的结果是，类库的可执行代码将成为客户应用中不可分割的一部分。

假设对于前面提到的 FastString 类，它的四个方法所产生的机器码将在目标可执行文件中占有 16MB 的空间[不要忘了，为了完成 O(1) 时间阶搜索算法，有可能需要大量的代码，并且还要采用大多数算法经常使用的以空间换时间的策略]。如图 1.1 所示，如果三个应用都使用 FastString 库，那么每个可执行文件都将包含 16MB 的类库代码。这意味着，如果一个最终用户安装了所有这三个应用程序，那么 FastString 实现将会占用 48MB 的磁盘空间。更糟糕的是，如果最终用户同时运行这三个程序，那么 FastString 代码将会占用 48MB 的虚拟内存，因为操作系统不能够检测到多个可执行文件中重复出现的代码。

<sup>1</sup> 此时我手头没找到合适的算法。如何具体实现留给读者做练习吧。

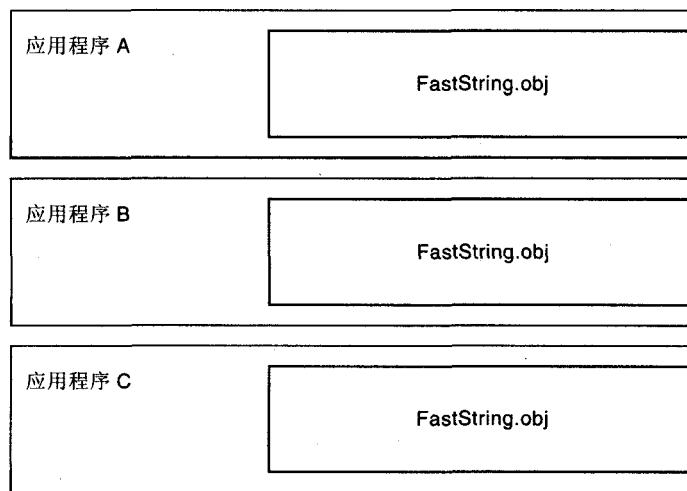


图 1.1 FastString 的三个客户

这种情况下另一个问题是，一旦类库的厂商发现了 FastString 类中的缺陷，没有任何办法能替换部分实现代码。一旦 FastString 代码已经被链接到客户应用中，我们就不可能在最终用户的机器上直接替换 FastString 代码。相反，类库厂商必须向每个客户应用的开发人员广播更新后的源代码，并且希望他们能够重新编译他们的应用，以便用上更新后的库代码。很显然，一旦客户运行了链接器并且产生了最终可执行代码之后，FastString 组件就没有模块化特征了。

## 1.2 动态链接和 C++

解决上面问题的一种技术是把 FastString 类以动态链接库（DLL，Dynamic Link Library）的形式包装起来。有很多种方法可以完成这样的工作。最简单的技术是用一个类层次上的编译器指示符，强迫 FastString 的所有方法都从 DLL 中引出去。Microsoft C++ 编译器为这种用法提供了 `__declspec(dllexport)` 关键字：

```
class __declspec(dllexport) FastString {
    char *m_psz;
public:
    FastString(const char *psz);
    ~FastString(void);
    int Length(void) const;           // 返回字符数目
    int Find(const char *psz) const; // 返回偏移量
};
```

在使用这项技术的时候，FastString 的所有方法都将被加到 FastString.DLL 的引出表（export list）中，允许在运行时把每个方法的名字解析到内存中对应的地址。而且，链接器将会产生一个引入库（import library），这个库暴露了 FastString 的方法成员的符号。引入库并没有包含实际的代码，它只是简单地包含一些引用，这些引用指向 DLL 的文件名和被引出的符号的名字。当客户链接引入库时，有一些存根会被加入到可执行文件中，它在运行时通知装载器动态装载 FastString DLL，并且把所有被引入的符号解析到内存中相应的位置上。当操作系统启动客户程序时，便会引发这个解析过程，但是整个过程完全是透明的。

图 1.2 演示了当 FastString 位于 DLL 中时，它的运行时模型。注意，引入库往往非常小（差不多是引出符号文本的两倍大小）。当 FastString 从 DLL 中引出时，它的机器码在用户的硬盘上只保留一份就可以了。当多个客户访问库中的代码时，操作系统的装载器可以很灵活地让所有的客户程序共享同一份“包含 FastString 只读可执行代码”的物理内存页。而且，如果库的厂商发现了库中的错误的话，理论上讲它可以给最终用户发放一个新的 DLL，使所有的客户应用同时修正原来的错误实现代码。很显然，把 FastString 库放到 DLL 中，这是从原始的 C++ 类走向可替换的、有效的可重用组件的重要一步。

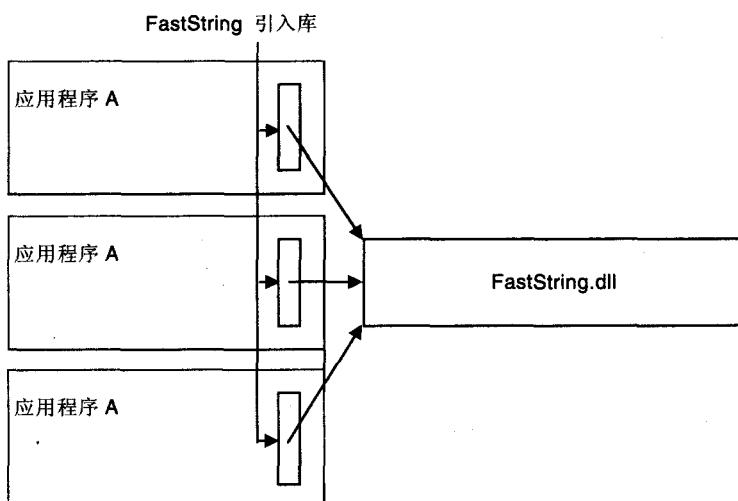


图 1.2 DLL 形式的 FastString

### 1.3 C++和可移植性

一旦已经确定了要以 DLL 的形式发布一个 C++类，你就将面临 C++的基本弱点之

一: C++缺少二进制一级的标准。虽然 ISO/ANSI C++草案工作文档 (Draft Working Paper, DWP (译注 1) 企图把“哪个程序将会编译、什么样的语义会影响到什么运行特性”编写成规范, 但是它并没有打算建立 C++的二进制运行时标准模型。这个问题第一次明显地暴露出来, 正是当客户试图在 C++的开发环境中链接 FastString DLL 的引入库的时候, 而不是当我们建立 FastString DLL 的时候。

为了允许操作符重载和函数重载, C++编译器往往任意篡改每一个入口点的符号名, 以便允许同一个名字 (具有不同的参数类型, 或者是不同的作用域) 有多个用法, 而不会打破现有的基于 C 的链接器。这项技术通常被称为名字改编 (name mangling)。尽管《C++ Annotated Reference Manual》记录了 CFront 采用的编码方案, 但是许多编译器厂商选择了自己专有的名字改编方案。由于 FastString 引入库和 DLL 引出符号使用了创建该 DLL 的编译器 (比如说 GNU C++) 的名字改编方案, 所以使用其他编译器 (比如说 Borland C++) 产生的客户将无法与引入库成功链接。消除名字改编现象的经典技术是使用 `extern "C"`, 但是这项技术对于 FastString 这种情况并没有用, 因为它引出的是成员函数, 而非全局函数。

有一项技术能够减轻这个问题, 就是在客户的链接器上做一些文章, 使用模块定义文件 (Module Definition File, 通称为 DEF 文件)。DEF 文件的一个特征是, 它允许引出符号被化名为不同的引入符号。给定足够的时间和关于每个编译器改编方案的信息, 则库厂商可以为每个不同的编译器产生一个专门定制的引入库。这样做虽然有些冗余, 但是毕竟使任何一个编译器都能够获得“与 DLL 在链接层次上的兼容性”。

一旦我们已经克服了与链接相关的问题, 下一步就要处理与最终产生的代码相关的更多问题了。除了最简单的语言结构之外, 所有的编译器厂商都会选择自己专有的方式来实现语言的其他特征, 从而使其他的编译器无法使用它所产生的代码。异常 (exception) 就是这些语言特征的一个典型例子。Microsoft 编译器编译产生的函数, 它所抛出的异常不能被 Watcom 编译器编译生成的客户程序捕捉到。因为 C++草案工作文档并没有规定语言的特征在运行时看起来应该怎么样, 所以每个编译器厂商按自己制定的方式来实现语言特征完全是合法的。对于创建单独运行的可执行代码来说, 这当然不是什么问题, 因为所有的代码都使用统一的开发环境编译和链接。但是, 对于创建多个二进制的、基于组件的可执行代码来说, 这是个很严重的问题, 因为每个组件很有可能是用不同的编译器和链接器建立起来的。C++缺少二进制标准, 这限制了语言的特征在跨越 DLL 边界时的应用。这意味着简单地从 DLL 中引出 C++成员函数, 还不足以创建“厂商独立的组件软件”。

---

译注 1: C++标准已于 1998 年 7 月 27 日定案。请访问: <http://webstore.ansi.org/>。

## 1.4 封装性和C++

假定有人已经克服了上一节所说的编译器和链接器问题，那么在 C++ 中建立二进制组件的下一个障碍则与封装有关。考虑这样的情形：一个组织在应用中使用了 **FastString**，而他们又想在产品发货前两个月完成开发和测试，这几乎是不可能的。假设在这两个月期间，某些具有特殊怀疑精神的开发人员决定在他们的应用上运行一个性能评测工具，以便测试 **FastString** 的 O(1) 搜索算法。使他们惊讶的是，事实上 **FastString::Find** 执行得非常快，并且与字符串的长度无关。然而，**Length** 操作不是很理想。**FastString::Length** 使用 C 运行库的 **strlen** 过程，它要执行一个线性搜索，在字符串的数据中查找空 (null) 结束符。这是一个 O(n) 算法，当字符串数据很大的时候，它会很慢。客户应用可能要多次调用 **Length** 操作，于是这些开发人员要求库的厂商提高 **Length** 操作的执行速度，使它在常数时间内完成。但是现在有一个问题，开发人员已经完成了开发，他们不希望再改动任一行源代码以便使用新增强的 **Length** 方法。而且，其他一些厂商可能已经发放了基于当前 **FastString** 版本的产品。即使从道义上讲，库厂商也不应该影响这些已经面市的产品。

这时候，我们有必要查看 **FastString** 类的定义，并决定哪些可以改变，哪些不能改变以免影响已有的应用程序。幸运的是，在设计 **FastString** 的时候开发人员已经考虑到了封装的性质，它的所有数据成员都是私有的 (private)。这就提供了很大的灵活性，因为没有一个客户可以编写直接访问 **FastString** 数据成员的代码。既然类的四个公共 (public) 成员无需改变，那么客户应用也无需做任何变化。基于这样的想法，库厂商很快就实现了 **FastString** 的 2.0 版本。

一个很显然的改进是，在构造函数中把字符串的长度缓存起来，然后在新版本的 **Length** 方法中返回被缓存起来的长度。因为在构造函数之外不能修改字符串，所以根本不需要多次重复计算长度。实际上，在构造函数分配缓冲区的时候长度值已经被计算出来了，所以只需要少数几个附加的机器指令即可。下面是修改之后的类定义：

```
// faststring.h 2.0 版
class __declspec(dllexport) FastString {
    const int m_cch; // 字符数
    char *m_psz;
public:
    FastString(const char *psz);
    ~FastString(void);
    int Length(void) const; // 返回字符数
    int Find(const char *psz) const; // 返回偏移量
};
```

注意，唯一的修改是增加了一个私有的数据成员。为了初始化这个成员，构造函数需要作如下修改：

```
FastString::FastString(const char *psz)
: m_cch(strlen(psz)), m_psz(new char[m_cch + 1]) {
    strcpy(m_psz, psz);
}
```

有了这个缓存的长度信息，则 Length 方法就非常简单了：

```
int FastString::Length(void) const {
    return m_cch; // 返回缓存长度
}
```

做了这三个修改之后，库厂商现在可以重新编译链接（build）FastString DLL，以及相应的测试工具（它可以全面测试 FastString 类）。库厂商将会很高兴地发现，测试工具的源代码不需要做任何修改。一旦新的 DLL 已经通过检验可以正常工作了，库厂商就可以向客户发放 FastString 的 2.0 版本了，可以确信所有的工作都已经完成了。

当客户收到了更新后的 FastString 时，他们把新的类定义和 DLL 集成到他们的源代码控制系统中，然后发出编译命令，并测试新的增强的 FastString 版本。与库厂商一样，他们也惊奇地发现，为了使用新版本的 Length，根本无需做任何修改。有了这样的经验作鼓舞，开发小组决定把新的 DLL 加到最后的“金盘母片 CD”中，准备制作最终的产品。在极少的情况下，管理人员会由于热心的开发人员的愿望而把新的 DLL 加到最终的产品中。与大多数的安装程序一样，客户产品的安装脚本只是简单地覆盖最终用户机器上可能有的 FastString 老版本 DLL。这往往不会有问题，因为修改并没有影响到类的公开接口，所以在机器范围内简单地更新到 FastString 2.0 版本，应该只会增强原先已经安装的客户应用。

现在请想像这样的情形：最终用户终于收到了他们的产品。每个最终用户立刻打开软件包装，把新的应用程序安装到他们的机器上，并试着运行新的应用程序。由于能够以极快的速度执行文本搜索过程，用户非常激动，然后他或她又回到正常的工作中，并启动以前安装的一个应用，碰巧它也要用到 FastString DLL。在最初几分钟，一切都很正常。然后，突然出现了一个对话框，提示发生了一个异常，并且用户的所有工作都丢失了。当他或她试着再次启动这个应用程序时，这次异常对话框马上就出来了。用户已经习惯于使用现代的商业软件了，他或她重新安装了操作系统和所有的应用程序，但即使这样也不能避免再次发生异常。到底怎么回事？

这里发生的事情是由于库厂商过于相信 C++ 支持的封装性了。C++ 通过 private 和 public 关键字确实支持语法上的封装性，但是 C++ 草案标准并没有定义二进制层次上的封装性。这是因为 C++ 的编译模型要求客户的编译器必须能够访问与对象的内存布局有关的所有信息，这样才能构造类的实例，或者调用类的非虚成员函数。这些信息包括对象的私有成员和公共成员的大小和顺序。考虑如图 1.3 所示的情形。1.0 版本的 FastString

要求每个实例 4 个字节（假定 `sizeof(char*) == 4`）。针对 1.0 版本类定义编写的客户分配 4 个字节的内存，并传给类的构造函数。2.0 版本的构造函数、析构函数和方法（这是最终用户机器上当前的版本）都假定客户为每个实例分配了 8 个字节内存（假定 `sizeof(int) == 4`），并且毫无保留地写入所有这 8 个字节。不幸的是，在 1.0 版本的客户中，对象的后 4 个字节实际上是属于别人的，在这个位置上写入一个指向文本串的指针，这是非常粗暴的行为，于是异常对话框就出来了。

针对这种情况一个通用的解决方案是每次新的版本问世时，就把 DLL 改成其他的名字。这也正是 MFC（Microsoft Foundation Classes，微软基础类）采用的策略。当我们把版本号放到 DLL 的文件名中（比如 `FastString10.DLL`、`FastString20.DLL`）时，客户总是可以装入与创建时相对应的 DLL 版本，而无须顾虑当前系统中是否有其他的版本存在。不幸的是，随着时间的推移，由于软件配置混乱，用户系统中各种版本的 DLL 极有可能会超过实际客户应用的数量。你只需在任何一台使用期已经超过 6 个月的机器上检查它的系统目录，就可以知道我这么说是有道理的。

从根本上讲，版本问题的根源在于 C++ 的编译模型，这种模型不能支持独立二进制组件的设计。C++ 的编译模型要求客户必须知道对象的布局结构，从而导致了客户和对象可执行代码之间的二进制耦合关系。通常情况下，二进制耦合对于 C++ 非常有好处，因为这使得编译器可以产生非常高效的代码。但是不幸的是，这种紧密耦合性使得在不重新编译客户的情况下，类的实现无法被替换。由于这种耦合性，以及上一节提到的编译器和链接器的不兼容性，“简单地把 C++ 类的定义从 DLL 中引出来”这种方案并不能提供合理的二进制组件结构。

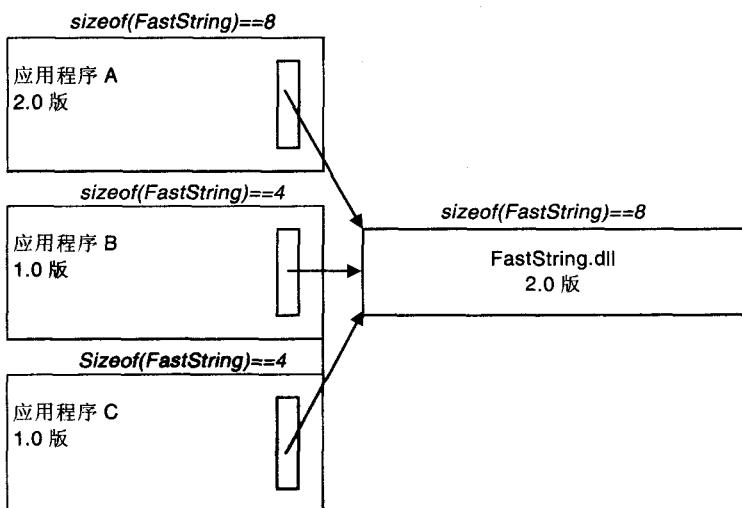


图 1.3 C++ 和封装性

## 1.5 把接口从实现中分离出来

封装（encapsulation）的概念以把“一个对象的外观（接口）同其实际工作方式（实现）分离开来”为基础。C++的问题在于这条原则并没有被应用到二进制层次上，因为C++类既是接口也是实现。这个弱点可以这样来解决：构造一个模型，把两个抽象概念（即接口和实现）做成两个分离的实体，即C++类。定义一个C++类使它代表指向一定数据类型的接口；定义另一个C++类作为数据类型的实际实现，于是从理论上讲，对象的实现者可以修改实现类的细节，而接口类保持不变。我们需要一种方法，它能够把接口和实现联系起来，但是又不会向客户暴露任何实现细节。

接口类应该只描述实现者希望客户知道的底层数据类型的面貌。由于接口没有暴露任何实现细节，所以C++接口类不应该包含任何用于对象实现的数据成员。相反，接口类应该只包含对象的每个操作的方法声明。C++实现类将包含实际的数据成员，这些数据成员可用于实现对象的功能。一个简单的办法是用一个句柄类作为接口。句柄类只包含一个实体指针，在客户的范围内它的类型永远也不需要被完全定义出来。下面的类定义说明了这项技术：

```
// faststringitf.h
class __declspec(dllexport) FastStringItf {
    class FastString; // 导入实现类的名称
    FastString *m_pThis; // (大小保持固定)
public:
    FastStringItf(const char *psz);
    ~FastStringItf(void);
    int Length(void) const; // 返回字符数
    int Find(const char *psz) const; // 返回偏移量
};
```

注意，这个接口类的二进制布局结构并不会随着实现类FastString中数据成员的加入或者删除而改变。而且，使用这样的声明也意味着FastString的类声明不需要被包含在这个头文件中就可以进行编译。这就很有效地把FastString的实现部分隐藏起来，客户的编译器不需要知道这些细节。在使用这项技术的时候，接口方法的机器码变成了对象DLL的唯一入口点，它们的二进制结构形式永远也不会再改变。接口类方法的实现只是把方法调用传递给实际的实现类：

```
// faststringitf.cpp // (DLL的一部分, 不是客户程序) //
#include "faststring.h"
#include "faststringitf.h"
FastStringItf::FastStringItf(const char *psz)
```

```

: m_pThis(new FastString(psz)) {
    assert(m_pThis !=0);
}

FastStringItf::~FastStringItf(void) {
    Delete m_pThis;
}

int FastStringItf::Length(void) const {
    return m_pThis->Length();
}

int FastStringItf::Find(const char *psz) const {
    return m_pThis->Find(psz);
}

```

以上这些传递调用的方法将被编译成为 FastString DLL 的一部分，所以当 C++ 实现类 FastString 的内存结构发生变化时，对 FastStringItf 构造函数中的 new 操作符的调用也要被重新编译，以确保总是分配足够的内存。而且，客户永远也不会包含 C++ 实现类 FastString 的类定义。这使得 FastString 实现者非常灵活，可以随着时间的推移改变它的实现过程，而不会打断现有的客户。

图 1.4 显示了采用句柄类把接口与实现分开的运行时模型图。注意，接口类引入的间接性相当于在客户和对象实现之间强制加入了一道二进制防火墙。这二进制防火墙是一个非常精确的协议，它描述了客户与对象实现之间如何进行通信。客户与对象之间的所有通信都要通过接口类才能进行，这就相当于强加了一个简单的二进制协议，只有通过这个协议才能进入对象的实现区域。这个协议并不依赖于 C++ 实现类的任何细节。

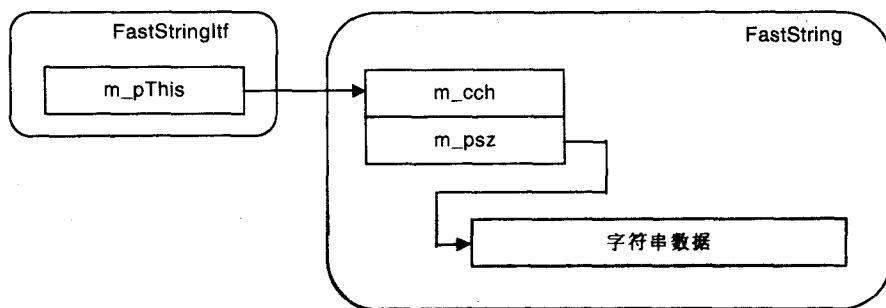


图 1.4 句柄类作为接口

虽然使用句柄类的方法有它的优点，而且使我们能够安全地把一个类从 DLL 中暴露出来，但是它也有自身的弱点。注意，接口类必须要把每个方法调用显式地传递给实现类。对于一个简单的类，比如 FastString，它只有两个公共（public）操作、一个构造

函数和一个析构函数，这当然不是负担。但是对于大的类库，它可能有上百个方法甚至上千个方法，那么光是编写这些方法传递过程就有可能非常冗长，也增加了出错的可能性。而且，对于性能非常关键的应用，每个方法增加两个函数调用（一个调用到接口，另一个嵌套调用到实现部分）的开销并不理想。最后，句柄类并没有完全解决编译器/链接器兼容性的问题，而这正是我们实现可重用组件所必须要解决的问题。

## 1.6 抽象基类作为二进制接口

使用“接口与实现分离”的技术确实可以解决 C++ 的编译器/链接器的兼容性问题；然而，接口类的定义必须使用特殊的形式。正如前面所说的，兼容性问题起源于不同的编译器对于以下两个方面有不同的考虑方案：（1）如何在运行时表现语言的特征；（2）在链接时刻如何表达符号名字。如果我们能够设计一项技术，把编译器和链接器的实现细节隐藏在某种二进制接口的后面，那么这将使基于 C++ 的 DLL 可被更多的客户使用。

保证 C++ 接口类所强加的二进制防火墙只使用与编译器无关的语言特征，这可以解决编译器/链接器的依赖性问题。为了实现这种独立性，我们必须首先确定语言的哪些方面具有统一的实现形式（即对于所有的编译器都一致）。当然，复合类型（比如 C 风格的 struct）在运行时的表现形式对于不同的编译器往往保持不变。这是一种基本的假设，任何基于 C 的系统调用都必须遵守这种假设，有时候利用带条件编译的类型定义、编译指示（pragma）或者其他编译器指示符（compiler directive）可以做到这一点。第二个假设是，所有的编译器都强制使用同样的顺序（从右到左，或者从左到右）传递函数参数，并且堆栈的清理也必须按照统一的方式进行。如同结构（struct）兼容性一样，这也是一个已经被解决的问题，通常使用条件编译器指示符以便强制使用统一的堆栈规则。Win32 API 的 WINAPI/WINBASEAPI 宏就是这项技术的一个例子。从系统 DLL 引出的每个函数都使用这些宏来定义：

```
WINBASEAPI void WINAPI Sleep(DWORD dwMsecs);
```

每个编译器厂商都定义了这些预编译符号，以便产生兼容的堆栈结构（stack frame）。虽然在生产环境中，我们可能希望对所有的方法定义都使用类似的技术，但是这一章的代码片断并没有使用这项规范堆栈的技术。

编译器不变性的第三个假设也是最关键的假设，正是这个假设才使得“二进制接口的定义”能够真正有效：某个给定平台上的所有 C++ 编译器都实现了同样的虚函数调用机制。实际上，这个一致性假设只需要适用于“没有数据成员、并且至多只有一个基类（基类也没有数据成员）”的类即可。这个假设意味着，对于下面的简单类定义：

```
class calculator {
```

```

public:
    virtual void add1(short x);
    virtual void add2(short x,short y);
};

```

对于下面的客户代码片断，在某个给定的平台上，所有的编译器必须产生等价的机器码序列：

```

extern calculator *pcalc;
pcalc->add1(1);
pcalc->add2(1,2);

```

虽然所有的编译器产生的机器码不必完全相同（*identical*），但是必须等价（*equivalent*）。这意味着对于每个编译器来说，一个类的对象在内存中如何表示，以及在运行时虚函数如何被动态调用的，都必须遵从同样的假设。

看起来距离并不太遥远。在 C++ 中，虚函数的运行时实现采用了 vptr 和 vtbl 的形式，几乎所有的编译器产品都是这样的。这项技术的基础是这样的：编译器在后台为每个包含虚函数的类产生一个静态函数指针数组。这个数组也被称为虚函数表（或 vtbl），在这个类或者它的基类中定义的每个虚函数都有一个相应的函数指针。该类的每个实例包含一个不可见的数据成员，被称为虚函数指针（或者 vptr），这个指针被构造函数自动初始化，指向类的 vtbl。当客户调用虚函数的时候，编译器产生代码反指向到 vptr，索引到 vtbl 中，然后在指定的位置上找到函数指针，并发出调用。这就是 C++ 中实现多态性以及动态调用分发的过程。图 1.5 显示了前面给出的计算器类（calculator class）在运行时 vptr 和 vtbl 的内存结构。

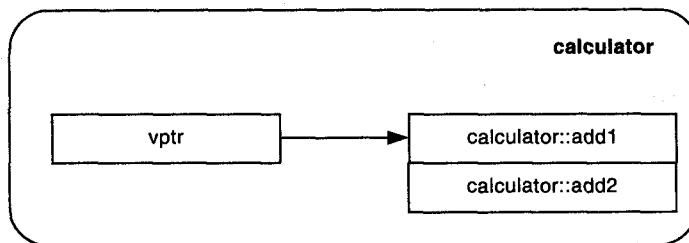


图 1.5 vptr 和 vtbl 的内存结构

几乎每一个当前正在使用的、算得上软件产品的 C++ 编译器都用到了 vptr 和 vtbl 的基本概念。对于 vtbl 的布局结构存在两种基本的技术：一种是 CFront 技术，另一种则是 adjustor thunk 技术。这两种技术各自都有处理多重继承细微之处的手段。幸运的是，在给定的一种平台上，往往会有种技术占主导地位（Win32 编译器使用 adjustor thunk 技术，而 Solaris 编译器使用 CFront 风格的 vtbl）。同样幸运的是，这两种 vtbl 格式都不会影响到程序员必须要编写的 C++ 源代码，因为 vtbl 只是内部产生的代码。请

参考 Stan Lippman 撰写的一本非常不错的书籍《Inside the C++ Object Model》，书中讨论了这两项技术。

根据前面所作的假设，现在我们有可能解决编译器依赖性的问题。假定在给定平台上所有的编译器都使用同样的方式实现虚函数调用机制，于是我们可以这样来定义 C++ 接口类：所有对数据类型的公共（public）操作都被定义成虚函数，这样可以保证所有的编译器将为客户端的方法调用产生等价的机器码。这种一致性假设要求这样两个条件：接口类不能包含数据成员，并且接口类不能直接从多个其他接口类派生。因为接口类没有数据成员，所以没有人能够用任何实在的方式来实现这些方法。

为了进一步加强这种思想，把接口成员定义为纯虚函数会非常有用，这说明接口类只是定义了“调用这些方法的可能性”，而没有定义实际的方法实现。

```
// ifaststring.h //////////
class IFastString {
public:
    virtual int Length(void) const = 0;
    virtual int Find(const char *psz) const = 0;
};
```

把方法定义为纯虚函数相当于告诉编译器，接口类不要求实现这些方法。当编译器为接口类产生 vtbl 时，每个纯虚函数的表项或者是空（null），或者指向 C 运行库中某个特殊的过程（在 Microsoft C++ 中是 \_pcrecall）的指针（当这个过程被调用的时候，它会激发一个断言，assertion）。如果不把方法定义声明为纯虚函数，那么编译器将企图用接口类方法的实现过程来填充相应的 vtbl 表项，当然这样的实现过程并不存在，所以结果就会导致链接错误。

刚才定义的接口类就是抽象基类。对应的 C++ 实现类必须从接口类继承，并且重载每个纯虚函数，以及实现这些函数。这种继承关系将导致对象的内存结构是接口类（实际上接口类只不过是 vptr/vtbl）的内存结构的二进制超集。这是因为在 C++ 中，派生类和基类之间的“is-a（是一个）”关系应用在二进制层次上，如同应用在面向对象设计的模型层次上一样：

```
class FastString : public IFastString {
    const int m_cch; // 字符数
    char *m_psz;
public:
    FastString(const char *psz);
    ~FastString(void);
    int Length(void) const; // 返回字符数
    int Find(const char *psz) const; // 返回偏移量
};
```

因为 FastString 从 IFastString 继承，所以 FastString 对象的二进制结构必须是

**IFastString** 的二进制结构的超集。这意味着，**FastString** 对象将包含一个 vptr，它指向一个与 **IFastString** 兼容的 vtbl。由于 **FastString** 是一个实际的、可被实例化的数据类型，所以它的 vtbl 将包含指向 **Length** 和 **Find** 方法实际实现的指针。这个关系如图 1.6 所示。

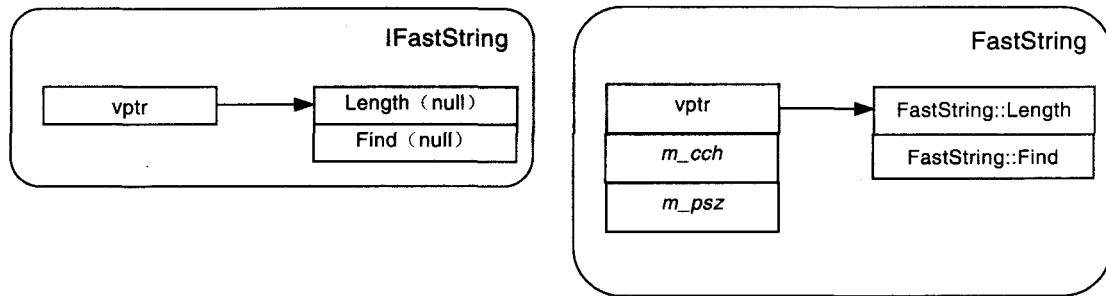


图 1.6 接口类和实现类的二进制结构

即使数据类型的公共操作已经被提升成为抽象基类中的纯虚函数了，如果客户没有实现类的类定义的话，它仍然不能构造 **FastString** 类的对象。把实现类的定义暴露给客户等于绕过了接口的二进制封装，从而破坏了使用接口类的基本意图。为了使客户能够构造 **FastString** 对象，一种合理可行的技术是让 DLL 引出一个全局函数，由它代表客户调用 **new** 操作符。这个函数必须以 **extern "C"** 的方式被引出来，因此任何一个 C++ 编译器都可以访问这个函数。

```

// ifaststring.h //////////
class IFastString {
public:
    virtual int Length(void) const = 0;
    virtual int Find(const char *psz) const = 0;
};

extern "C"
IFastString *CreateFastString(const char *psz);

// faststring.cpp (DLL的一部分) //////////
IFastString *CreateFastString (const char *psz) {
    Return new FastString(psz);
}

```

如同句柄类的方法一样，**new** 操作符仅仅在 **FastString** DLL 内部被调用，这意味着对象的大小和布局结构将使用与“编译实现类的所有方法”相同的编译器建立起来。

最后一个有待于进一步克服的障碍与**对象的析构函数**有关。下面的客户代码能够编译通过，但是结果并非如预期的那样：

```

int f(void) {
    IFastString *pfs = CreateFastString("Deface me");
    int n = pfs->Find("ace me");
    delete pfs;
    return n;
}

```

这种不可预期的行为源于这样的事实：接口类的析构函数并不是虚函数。这意味着对 `delete` 操作符的调用并不会动态地找到最终派生类的析构函数，并从最外层类型向基类型递归地销毁对象。因为 `FastString` 析构函数永远也不会被调用到，所以前述的例子会发生内存泄漏，用于保存字符串“`Deface me`”的内存将被遗漏。

针对这个问题一个显而易见的解决方案是把接口类的析构函数做成虚函数。不幸的是，这样将会破坏接口类的编译器独立性，因为虚析构函数在 `vtbl` 中的位置随着编译器的不同而不同。对于这个问题，一个可行的解决方案是显式地增加一个 `Delete` 方法，作为接口类的另一个纯虚函数，并且让派生类在这个方法实现中删除自身。这样做可以导致正确的析构过程。更新后的接口头文件如下所示：

```

// ifaststring.h //////////
class IFastString {
public:
    virtual void Delete(void) = 0;
    virtual int Length(void) const = 0;
    virtual int Find(const char *psz) const = 0;
};

extern "C"
IFastString *CreateFastString (const char *psz);

```

对应的实现类的定义如下：

```

// faststring.h /////////////////////////////////
#include "ifaststring.h"
class FastString : public IFastString {
    const int m_cch; // 字符数
    char *m_psz;
public:
    FastString(const char *psz);
    ~FastString(void);
    void Delete(void); // 销毁实例
    int Length(void) const; // 返回字符数
    int Find(const char *psz) const; // 返回偏移量
};

// faststring.cpp /////////////////////////////////
#include <string.h>
#include "faststring.h"
IFastString* CreateFastString (const char *psz) {
    Return new FastString(psz);
}

```

```

}

FastString::FastString(const char *psz)
: m_cch(strlen(psz)), m_psz(new char[m_cch + 1]) {
    strcpy(m_psz, psz);
}
void FastString::Delete(void) {
    delete this;
}
FastString::~FastString(void) {
    Delete[] m_psz;
}

int FastString::Length(void) const {
    return m_cch;
}

int FastString::Find(const char *psz) const {
    // O(1) 查找代码, 从略
}

```

图 1.7 显示了 FastString 的运行时内存结构。

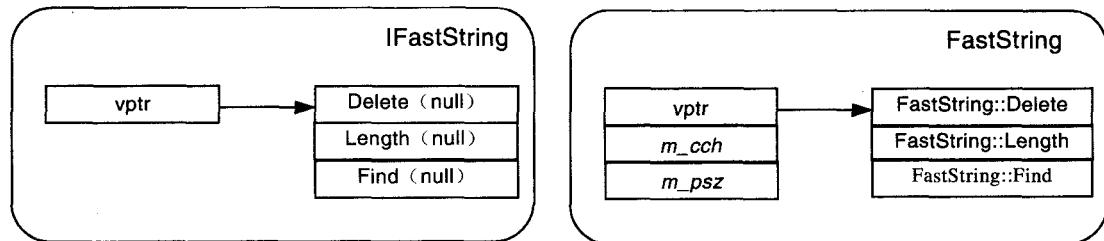


图 1.7 使用抽象类作为接口的 FastString

为了使用 FastString 数据类型，客户只需简单地包含接口定义文件，并且调用 CreateFastString 就可以开始工作：

```

#include "ifaststring.h"

int f(void) {
    int n = -1;
    IFastString *pfs = (CreateFastString("Hi Bob! "));
    If (pfs) {
        n = bfs->Find("ob");
        bfs->Delete();
    }
    return n;
}

```

注意，在 FastString DLL 中，除了一个入口函数之外，其他所有的入口函数都是虚函数。接口类的虚函数总是通过保存在 vtbl 中的函数指针被间接调用，客户程序不需要在开发时候链接这些函数的符号名。这意味着接口方法可以不受“不同编译器之间符号名改编方式的不同”的影响。唯一需要通过名字显式链接的入口函数是 CreateFastString，这个全局函数使得客户可以进入到 FastString 的世界中。然而，请注意，这个函数使用 extern “C” 被引出，从而避免了符号名改编方式的冲突。这也隐含着所有的 C++ 编译器都期望引入库和 DLL 引出同样的符号。使用这项技术的一个直接结果是，我们可以安全地在一个 C++ 环境中暴露 DLL 中的类，并且在另一个 C++ 开发环境中访问这个类。对于建立一个与编译器厂商无关的可重用组件来说，这种能力是非常重要、非常关键的。

## 1.7 运行时多态性

用抽象基类作为接口来实现对象，可以说打开了一个全新的世界，这个世界在运行时有着各种各样的可能性。注意，FastString DLL 只引出了一个符号 CreateFastString。这使得客户可以很方便地按需动态装入 DLL（使用 LoadLibrary），并且使用 GetProcAddress 得到唯一的入口函数：

```
IFastString *CallCreateFastString(const char *psz) {
    Static IFastString * (*pfn)(const char *) = 0;
    if (!pfn) { // 如是首次就初始化 ptr
        const TCHAR szDLL[] = __TEXT("FastString.DLL");
        const char szFn[] = " CreateFastString";
        HINSTANCE h = LoadLibrary(szDll);
        if (h)
            *(FARPROC*)&pfn = GetProcAddress(h, szFn);
    }
    return pfn? pfn(psz) : 0;
}
```

这项技术有几个可能的应用。使用这项技术的一个动机是，当客户程序在没有安装对象实现的机器上运行时，它可以避免操作系统产生错误。有些应用使用可选的系统组件，比如 WinSock 或者 MAPI，它们实际上使用了类似的技术，从而可以在最低配置的机器（也就是没有安装 WinSock 或者 MAPI 的机器）上运行。因为客户并不需要链接 DLL 的引入库，所以它对 DLL 并没有依赖性，即使机器上没有安装相应的 DLL，它也一样可以执行。使用这项技术的另一个动机是，它可以减少进程地址空间初始化的工作。而且，因为 DLL 并没有在进程初始化的时候被自动装载进来，所以，如果对象实现代码没有被真正使用到的话，那么 DLL 永远也不会被装载进来。使用这项技术的其他好处包括：加快客户的启动速度、长时间运行的进程可以保留更多的地址空间（如果这个进

程并没有真正使用这些 DLL 的话) 等。

这项技术最有趣的应用之一可能是, 允许客户在同一接口的不同实现之间动态地做出选择。如果 `IFastString` 接口的定义已经公开了, 那么 `FastString` 的原始实现者或者其他第三方厂商就可以利用同样的接口开发出另外的实现类。与原始的实现类 `FastString` 一样, 这些新的实现类的布局与原始的接口类的布局在二进制结构上完全兼容。为了访问这些“可插入并且兼容的”实现, 客户所必须要做的工作就是指定这些实现的 DLL 的正确文件名。

为了理解这项技术是如何被应用的, 我们假定原来的 `IFastString` 实现按从左到右的顺序执行搜索过程。这种行为方式对于从左到右解析的语言(比如英语、法语和德语等)非常合适。但是对于从右到左解析的语言, 则使用从右到左顺序执行搜索的 `IFastString` 实现会更合适。我们可以把第二种做法做成第二个 DLL, 使用不同的名字(比如 `FastStringRL.DLL`)。假定这两个 DLL 同时被安装在最终用户的机器上, 那么客户可以在运行时动态地选择它所期望的 `IFastString` 实现, 它只要装载对应的 DLL 即可。

```
IFastString*
CallCreateFastString(const char *psz,
                     bool bLeftToRight = true) {
    static IFastString * (*pfnlr)(const char *) = 0;
    static IFastString * (*pfndl)(const char *) = 0;

    IFastString *(*ppfn)(const char *) = &pfnlr;
    const TCHAR *pszDll = __TEXT("FastString.DLL");

    if (!bLeftToRight) {
        pszDll = __TEXT("FastStringRL.DLL");
        ppfn = &pfndl;
    }

    if (!(*ppfn)) { // 如是首次, 初始化
        const char szFn[] = "CreateFastString";
        HINSTANCE h = LoadLibrary(pszDll);
        if (h)
            *(FARPROC*)ppfn = (GetProcAddress(h, szFn));
    }
    return (*ppfn) ? (*ppfn)(psz) : 0;
}
```

当客户调用这个函数, 但是没有指定第二个参数时:

```
pfs = (CallCreateFastString("Hi Bob!"));
n = pfs->Find("ob");
```

则原来的 `FastString` DLL 被装载进来, 并且搜索过程按从左到右的顺序进行。如果客户指定它的字符串属于从右到左解析的语言:

```
pfs = CallCreateFastString("Hi Bob!", false);
n = pfs->Find("ob");
```

那么另一个 DLL 版本 (FastStringRL.DLL) 会被装载进来，并且搜索过程将从字符串最右边的位置开始。关键的一点在于，CallCreateFastString 的调用者并不关心是用哪个 DLL 来实现对象的方法的。它在意的是，函数返回一个指向 IFastString 兼容的 vptr 的指针，并且这个 vptr 提供了非常有用的功能。这种运行时多态性对于“利用二进制组件建立动态的复合系统”来说，非常有帮助。

## 1.8 对象扩展性

到现在为止所展示的技术使得客户可以动态地选择并装载二进制组件，从而可以随着时间的推移不断升级它们的实现，而客户无须重新编译。对于创建动态的复合系统来说，这绝对是非常有用的。然而，对象的接口却不能随着时间不断进化。这是因为，客户在编译过程中需要有精确的接口类定义，对接口定义的任何变化都要求客户程序重新编译，以便适应这种变化。更糟糕的是，改变接口的定义完全违背了对象的封装性，因为对象的公开接口已经被改变了，而且这样做也会伤害到现有的客户程序。即使是最无伤大雅的变化，比如改变了一个方法的语义但是保持它的原型不变，也会导致已安装的客户不能再发挥作用。这意味着接口是绝对不能改变的，它既是语义的约定，也是二进制结构的约定。为了拥有一个稳定的、行为可预测的运行时环境，这种不变性是很重要的。

尽管接口具有不变性的原则，但是我们通常要在接口已经被设计好之后，希望能够加入原先没有预见到的新功能。我们可以利用 vtbl 布局结构的知识，只是简单地把新的方法追加在现有接口定义的尾部。考虑下面初始版本的 IFastString 接口：

```
class IFastString {
public:
    virtual void Delete(void) = 0
    virtual int Length(void) = 0
    virtual int Find(const char *psz) = 0
};
```

简单地修改接口类，在现有的方法声明之后加入新的虚函数声明，这样得到的二进制 vtbl 格式是原先 vtbl 版本的超集，因为新的 vtbl 表项总是出现在原先方法的表项之后。在针对新接口定义编译得到的对象实现中，这些新的方法表项也将被加在原先 vtbl 布局结构的尾部：

```
class IFastString {
```

```

public:
// faux version 1.0
    virtual void Delete(void) = 0;
    virtual int Length(void) = 0;
    virtual int Find(const char *psz) = 0;
// faux version 2.0
    virtual int FindN(const char *psz,int n) = 0;
};

```

这种方案可以正常工作。在老的接口版本基础上编译得到的客户完全忽略 vtbl 中前三个表项之外的任何信息。当老客户得到了包含 FindN 表项的新对象时，它们仍然能够正常工作。然而，新的客户总是期望 IFastString 有第四个方法，当它碰巧使用老的对象时，因为老的对象并没有实现客户期望的方法，所以问题就发生了。当客户针对“基于原先的接口定义编译得到的对象”调用 FindN 方法时，其结果显而易见，程序崩溃了。

这项技术的问题在于，它修改了公开的接口，从而打破了对象的封装性。就像改变 C++ 类的公开接口会引起“在重编客户代码时发生编译时的错误”一样，改变二进制接口定义也会引起“客户代码再次执行时产生运行时错误”。这意味着接口必须是不可改变的，一旦公开之后，接口就不能再变化。这个问题的解决办法是“允许实现类暴露多个接口”。这可以通过两种途径获得：设计一个接口使它继承另一个相关的接口，或者让实现类继承多个不相关的接口类。无论是哪种途径，客户总是可以使用 C++ 的运行时类型识别（RTTI, Runtime Type Identification）功能，在运行时询问对象，以便确保当前正在使用的对象确实支持客户所请求的功能。

考虑从一个接口扩展另一个接口的简单情形。为了在 IFastString 接口中增加 FindN 操作，使客户可以找到一个子串第 n 次出现的位置，我们可以从 IFastString 接口派生出另一个接口，然后在这个接口中加入新方法的定义：

```

class IFastString2 : public IFastString {
public:
// real version 2.0
    virtual int FindN(const char *psz, int n) = 0;
};

```

客户可以在运行时询问对象，以确定对象是否与 IFastString2 相容。客户可以使用 C++ 的 dynamic\_cast 操作符：

```

int Find10thBob(IFastString *pfs) {
    IFastString2 *pfs2=dynamic_cast<IFastString2*>(pfs);
    if (pfs2) // IFastString2 派生的对象
        return pfs2->FindN("Bob",10);
    else { // 不是 IFastString2 派生的对象
        error("Cannot find 10th occurrence of Bob");
        return -1;
    }
}

```

如果对象确实是从扩展接口 `IFastString2` 派生的，那么 `dynamic_cast` 操作符将返回一个指向对象的 `IFastString` 兼容部分的指针，然后客户就可以调用对象的扩展方法。如果对象并不是从扩展接口 `IFastString2` 派生的，那么 `dynamic_cast` 操作符将返回一个空指针，然后客户可以选择其他的实现技术，或者记录一条错误信息，或者无动于衷地继续往下执行。这种由客户来决定的平滑降级能力，对于建立稳定的动态系统是非常关键的，这种动态系统可以随时间的推移不断提供新的扩展功能。

当对象需要暴露新的正交功能（译注 2）时，一个更为有趣的现象发生了。考虑一下当我们需要为 `FastString` 实现类增加永久性支持时所需要做的工作。尽管我们可以在 `IFastString` 扩展版本的接口中加入 `Load` 和 `Save` 方法，这当然是没有问题的，但是极有可能其他非 `IFastString` 兼容的对象也需要永久性。如果简单地创建一个新的接口，使它从 `IFastString` 派生：

```
class IPersistentObject : public IFastString {
public:
    virtual bool Load(const char *pszFileName) = 0;
    virtual bool Save(const char *pszFileName) = 0;
};
```

那么，这就要求所有的永久对象同时也要支持 `Length` 和 `Find` 操作。对于少量的对象来说，这样做可能是有意义的。然而，为了尽可能地使 `IPersistentObject` 接口具有通用性，实际上它应该是一个独立的接口，不需要从 `IFastString` 继承：

```
class IPersistentObjet {
public:
    virtual void Delete(void) = 0;
    virtual bool Load(const char *pszFileName) = 0;
    virtual bool Save(const char *pszFileName) = 0;
};
```

这样的定义并没有妨碍 `FastString` 实现也可以具有永久性，只是永久版本的 `FastString` 必须同时实现 `IFastString` 和 `IPersistentObject` 接口。

```
class FastString : public IFastString,
                  public IPersistentObject {
    int    m_cch; // 字符数
    char *m_psz;
public:
    FastString(const char *psz);
    ~FastString(void);
    // 一般方法
    void Delete(void); // 删除此实例
    // IFastString 方法
    int Length(void) const; // 返回字符数
```

---

译注 2：垂直功能，也就是与原来的功能不相关的新功能。

```

int Find(const char *psz) const; // 返回偏移量
// IPersistentObject 方法
bool Load(const char *pszFileName);
bool Save(const char *pszFileName);
};

```

为了把 FastString 保存到磁盘上，客户只要简单地使用 RTTI 得到一个指向对象暴露出来的 IPersistentObject 接口即可：

```

bool SaveString(IFastString *pfs,const char *pszFN) {
    bool bResult = false;
    IPersistentObject *ppo =
        dynamic_cast<IPersistentObject*>(pfs);
    if (ppo)
        bResult = ppo->Save(pszFN);
    return bResult;
}

```

这项技术可以正常工作，因为编译器具有关于“实现类的布局结构和类型层次”的足够知识，从而可以在运行时检查对象，以便确定对象是否继承自 IPersistentObject。但是这里仍然有问题。

RTTI 是一个与编译器极为相关的特征。我有必要再一次强调，C++草案工作文档规定了 RTTI 的语法和语义，但是每个编译器厂商对 RTTI 的实现则是独有的，也是私有的。这就大大破坏了“以抽象基类作为接口而获得的编译器独立性”。对于厂商中立的组件结构而言，这一点是不可被接受的。对于这个问题，一个很简捷的办法是，平衡处理 dynamic\_cast 的语义，不使用实际与编译器相关的语言特征。从每一个接口显式地暴露一个广为人知的方法，由这个方法完成与 dynamic\_cast 语义等价的功能，这样也可以获得同样的效果，而且不要求所有各方都使用同样的 C++ 编译器：

```

clsaa IPersistentObject
public:
    virtual void *Dynamic_Cast(const char *pszType) = 0;
    virtual void Delete(void) = 0;
    virtual bool Load(const char *pszFileName) = 0;
    virtual bool Save(const char *pszFileName) = 0;
};

class IFastString {
public:
    virtual void *Dynamic_Cast(const char *pszType) = 0;
    virtual void Delete(void) = 0;
    virtual int Length(void) = 0;
    virtual int Find(const char *psz) = 0;
};

```

既然所有的接口都需要暴露这个方法以及 Delete 方法，于是很自然的想法是把这些

公共的方法提升到一个基接口中，然后所有其他的接口都从这个接口继承得到：

```
class IExtensibleObject {
public:
    virtual void *Dynamic_Cast(const char* pszType) = 0;
    virtual void Delete(void) = 0;
};

class IPersistentObject : public IExtensibleObject {
public:
    virtual bool Load(const char *pszFileName) = 0;
    virtual bool Save(const char *pszFileName) = 0;
};

class IFastString : public IExtensibleObject {
public:
    virtual int Length(void) = 0;
    virtual int Find(const char *psz) = 0;
};
```

有了这样的类型层次之后，客户就可以利用编译器独立的结构，动态地查询对象是否实现了某个指定的接口：

```
bool SaveString(IFastString *pfs, const char *pszFN) {
    bool bResult = false;
    IPersistentObject *ppo = (IPersistentObject*)
        pfs->Dynamic_Cast("IPersistentObject");
    if(ppo)
        bResult = ppo->Save(pszFN);
    return bResult;
}
```

假定客户使用了上述方法，并且类型发现所要求的语义和机制也已经准备就绪，则每个实现类还必须手工实现此功能：

```
class FastString : public IFastString,
                    public IPersistentObject {
    int m_cch;                                // 字符数
    char *m_psz;
public:
    FastString(const char *psz);
    ~FastString(void);
// IExtensibleObject 方法
    void *Dynamic_Cast(const char *pszType);
    void Delete(void);                         // 销毁此实例
// IFastString 方法
    int Length(void) const;                   // 返回字符数
    int Find(const char *psz) const;           // 返回偏移量
```

```
// IPersistentObject 方法
bool Load(const char *pszFileName);
bool Save(const char *pszFileName);
};
```

Dynamic\_Cast 的实现通过操纵对象的类型层次结构，模拟出 RTTI 的功能。图 1.8 给出了刚才显示的 FastString 类的类型层次结构。因为实现类是从它所暴露的每个接口派生而来的，所以 FastString 中 Dynamic\_Cast 的实现可以简单地使用显式的静态类型转换功能，把 this 指针转换到客户所请求的子类型：

```
void *FastString::Dynamic_Cast(const char *pszType) {
    if (strcmp(pszType, "IFastString") == 0)
        return static_cast<IPastString*>(this);
    else if (strcmp(pszType, "IPersistentObject") == 0)
        return static_cast<IPersistentObject*>(this);
    else if (strcmp(pszType, "IExtensibleObject") == 0)
        return static_cast<IFastString*>(this);
    return 0; //未支持接口的请求
}
```

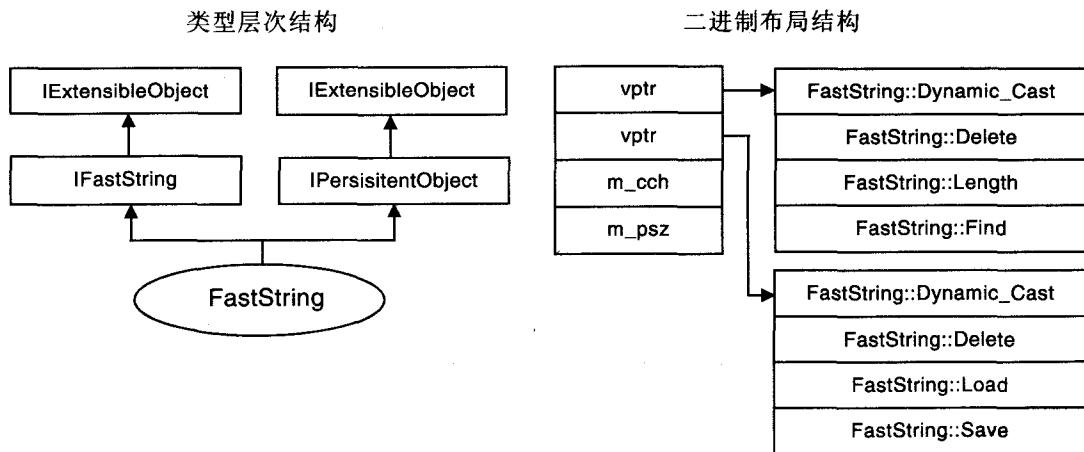


图 1.8 FastString 类型层次结构

由于对象是从 cast 指令中用到的类型派生而来的，所以 cast 指令的编译版本只是简单地在对象的 this 指针上加上固定的偏移，就可以找到基类的布局结构的起始处。

请注意，当客户请求公共的基接口 IExtensibleObject 时，实现类将自己静态转换为 IFastString。这是因为直觉上来看，下面的语句：

```
return static_cast<IExtensibleObject*>(this);
```

有二义性，因为 IFastString 和 IPersistentObject 都是从 IExtensibleObject 继承过来的。

如果 **IExtensibleObject** 是 **IFastString** 和 **IPersistentObject** 的虚基类，那么这个转换不会有二义性，所以这条语句能够正确编译。然而，引入虚基类将导致在结果对象中加入不必要的运行时复杂性，而且也会带来编译器相依性。这是因为虚基类是另一项“编译器厂商可以选择自己的实现方法”的 C++ 语言特征。

## 1.9 资源管理

当我们检查客户对 **Dynamic\_Cast** 方法的使用模式时，我们可以清楚地发现，有关单个对象支持多个接口还有一个问题需要解决。考虑下面的客户代码：

```
void f(void) {
    IFastString *pfs = 0;
    IPersistentObject *ppo = 0;
    pfs = CreateFastString("Feed BOB");
    if (pfs) {
        ppo = (IPersistentObject *)
            pfs->Dynamic_Cast("IPersistentObject");
        if (!ppo)
            pfs->Delete();
        else {
            ppo->Save("C:\\\\autoexec.bat");
            ppo->Delete();
        }
    }
}
```

尽管对象最初是通过其 **IFastString** 接口与客户联系起来的，但是客户代码却是通过 **IPersistentObject** 接口调用 **Delete** 方法的。有了 C++ 中多重继承的行为方式，这不成问题，因为所有属于该类的、从 **IExtensibleObject** 派生的 vtbl 都将指向 **Delete** 方法的唯一一份实现代码。然而，客户现在必须要记录下哪个指针是与哪个对象联系在一起的，并且每个对象只能调用一次 **Delete** 方法。对于上面给出的简单代码，这不是繁重的负担。但是在复杂的客户代码中，管理这些关系会变得非常复杂，并且很容易出错。简化客户的任务的一种办法是，把管理对象生命周期的责任推给对象实现部分。毕竟，允许客户显式地删除一个对象，这种做法会泄露出另一个实现细节：对象是被分配在堆（**heap**）上的事实。

对于这个问题，一个简单的解决方法是让每个对象都维护一个引用计数，当接口指针被复制的时候，该计数值增加；当接口指针被销毁的时候，该计数值减少。这意味着要把下面的 **IExtensibleObject** 定义：

```
class IExtensibleObject {
public:
    virtual void *Dynamic_Cast(const char* pszType) = 0;
    virtual void Delete(void) = 0;
};
```

改变为下面的定义：

```
class IExtensibleObject {
public:
    virtual void *Dynamic_Cast(const char* pszType) = 0;
    virtual void DuplicatePointer(void) = 0;
    virtual void DestroyPointer(void) = 0;
};
```

有了这些方法之后，现在 `IExtensibleObject` 的所有客户必须遵守下面两条要求：(1) 当接口指针被复制的时候，要求调用 `DuplicatePointer`；(2) 当接口指针不再有用时，要求调用 `DestroyPointer`。

这些方法可以在每一个对象中很方便地加以实现，只要时时注意活动指针的数目，如果不存在未完结的指针则将对象销毁：

```
class FastString : public IFastString,
                    public IPersistentObject {
    int m_cPtrs;           // 指针数目
    :
public:
    // 将指针计数初始化为零
    FastString(const char *psz) : m_cPtrs(0) {}
    void DuplicatePointer(void) {
        // 注意指针复制了
        ++m_cPtrs;
    }
    void DestroyPointer(void) {
        // 指针销毁时，将对象销毁
        if (--m_cPtrs == 0)
            delete this;
    }
    :
};
```

很容易可将这份样板代码放到基类中，或者放到 C 预编译宏中，供所有实现类使用。

为了支持这些方法，所有操作或者管理接口指针的代码必须遵守关于 `DuplicatePointer/DestroyPointer` 的两条简单规则。对于 `FastString` 实现来说，这意味着要修改两个函数。`CreateFastString` 函数要获得由 C++ `new` 操作符返回的初始指针，并复制到栈 (stack) 中，以便返回给客户。这意味着调用 `DuplicatePointer` 是不可缺少的：

```

IFastString* CreateFastString(const char *psz) {
    IFastString *pfsResult = new FastString(psz);
    if (pfsResult)
        pfsResult->DuplicatePointer();
    return pfsResult;
}

```

另一个复制指针的地方是在 Dynamic\_Cast 方法中：

```

void *FastString::Dynamic_Cast(const char *pszType) {
    void *pvResult = 0;
    if (strcmp(pszType, "IFastString") == 0)
        pvResult = static_cast<IFastString*>(this);
    else if (strcmp(pszType, "IPersistentObject") == 0)
        pvResult = static_cast<IPersistentObject*>(this);
    else if (strcmp(pszType, "IExtensibleObject") == 0)
        pvResult = static_cast<IFastString*>(this);
    else
        return 0; // 未支持接口的请求
    // pvResult 现在含有一个复制的指针,
    // 因此我们必须在返回前调用 DuplicatePointer
    ((IExtensibleObject*)pvResult)->DuplicatePointer();
    return pvResult;
}

```

有了这两处修改之后，对应的客户代码现在也变得更加统一、更加明确了（没有了歧义）：

```

void f(void) {
    IFastString *pfs = 0;
    IPersistentObject *ppo = 0;
    pfs = CreateFastString("Feed BOB");
    if (pfs) {
        ppo = (IPersistentObject *)
            pfs->Dynamic_Cast("IPersistentObject");
        if (ppo) {
            ppo->Save("C:\\\\autoexec.bat");
            ppo->DestroyPointer();
        }
        pfs->DestroyPointer();
    }
}

```

因为每个指针都被看作一个具有独立生命周期的实体，所以客户并不需要把哪个指针与哪个对象联系起来。相反，客户只需简单地遵守两条简单的规则，从而允许对象自己管理它的生命周期。如果你愿意的话，对 DuplicatePointer 和 DestroyPointer 的调用可

以很容易地被隐藏到 C++ 智能指针 (C++ smart pointer) 的后面。

利用引用计数的方案使得一个对象可以以非常一致的方式暴露多个接口。这种“单个实现类暴露多个接口”的能力允许一个数据类型（译注 3）可以参与到多种环境中。例如，一个新的永久子系统可能定义了自己独有的接口，通过这个接口客户可以告诉对象把它装载或者保存到某种特殊的存储介质中。`FastString` 类只要从该子系统的永久接口继承，就可以加入对这种功能的支持。增加这样的支持并不会影响到原先已经安装的客户，这些客户仍然可以使用原来的永久机制，从磁盘上装载字符串，或者把字符串保存到磁盘中。有了这种在运行时进行接口协商的机制，这将为创建一个“能够随时间不断进化”的动态组件系统奠定了基础。

## 1.10 我们走到哪儿了？

这一章从一个简单的 C++ 类开始，然后一步步讨论如何把这个类做成可重用的二进制组件。第一步是以动态链接库 (DLL) 的形式来发布这个类，以便从物理上把这个类的包装与客户的包装脱离开来。然后我们使用接口和实现的概念，把数据类型的实现细节封装到二进制防火墙后面，使得对象的布局结构能够随时间而进化，但无须要求客户重新编译。在采用抽象基类作为定义接口的方法之后，这道防火墙便以 `vptr` 和 `vtbl` 的形式出现了。接下去，我们使用 `LoadLibrary` 和 `GetProcAddress`，在运行时动态地选择同一接口的不同实现（呈现了运行时的多态性）。最后，我们使用与 RTTI 类似的结构，在运行时动态地询问对象，以确定对象实际上是否实现了指定的接口。这种结构使我们能够扩充接口的现有版本，并且也可以从单个对象暴露多个不相关的接口。

简而言之，我们刚刚设计了组件对象模型 (COM, Component Object Model)。

---

译注 3：这里指接口。

## 第 2 章

# 接 口

```
void *pv = malloc(sizeof(int));
int *pi = (int*)pv;
(*pi)++;
free(pv);
```

Anonymous, 1982

上一章展示了一系列 C++ 程序设计技术，这些技术使可重用的二进制组件的开发成为可能，并且这些组件能够随着时间的推移不断演化。这些技术与 COM 所使用的技术在精神上是一致的。上一章讲述的技术与 COM 所使用的技术之间轻微的差异几乎全部在于细节部分，而且总是由于特定的理由而被强加的。然而，上一章已经在基本概念方面告诉了你关于 COM 的故事，其中最首先也最重要的是将接口从实现中分离出来。

### 2.1 再谈接口与实现

把接口与实现分开的动机，是要把对象内部工作的细节（对客户而言）都隐藏起来。这条基本原则提供了一层间接性，允许实现类内部的数据成员的数量和顺序都可以发生变化，但是客户程序无需重新编译。这条原则也允许客户可以在运行时询问对象以便发

现对象的扩展功能。最后，这条原则还允许 DLL 和客户不必使用同样的 C++ 编译器。

尽管这最后一条是很有用的，但是它还不足以“为二进制组件提供一个普遍的底层基础”。其原因在于，虽然客户可以使用任何一个 C++ 编译器，但是最终它们必须要使用一个 C++ 编译器。前一章讲述的技术提供了编译器独立性。但是为了创建一个真正的二进制组件通用底层基础，最终我们还需要语言独立性。为了达到语言独立性的目的，“把接口与实现分离”的原则有必要再一次加以应用。

我们考虑上一章用到的接口定义。在 C++ 头文件中，每个接口定义都采用了 C++ 抽象基类定义的形式。定义接口的文件只能被一种语言解析，这暴露了对象实现的另一个细节：用来生成这个对象的语言。从本质上讲，我们应该从任何一种语言都可以访问这个对象，而不仅仅是实现对象所选用的语言。如果仅仅提供 C++ 兼容的接口定义，那么等于是对象实现者强迫组件的目标用户也必须要在 C++ 环境下工作。

尽管 C++ 是一门非常有用的编程语言，但事实上在许多程序设计领域中，其他的语言往往更适合于当前的工作。就好像“链接兼容问题”可以通过“为每一种可能的编译器都提供一个模块定义文件”的方式得到解决，使用同样的方案，我们也可以把 C++ 版本的接口定义翻译到每一种可能的编程语言中。由于接口的二进制原型特征只不过是一组简单的 vptr/vtbl，所以对于大多数的语言我们都可以做到这一点。

为所有已知的接口产生相应的语言映射版本，将需要大量的工作，而且要想跟上每个年代中有关编程语言的大量发明创新几乎是不可能的。理想的做法是，有人编写了一个工具，它可以把 C++ 类定义解析成某种抽象的中间状态。再根据这种中间状态，通过针对特定编程语言的后台产生器，产生相应的语言映射版本。当新的语言变得很重要的时候，新的后台发生器可以被加入进来，只需加入一次，那么所有以前定义的接口在新的编程环境中就都可以使用了。

不幸的是，C++ 编程语言充满了模棱两可的因素，这使它无法很好地映射到所有可能的语言。这些模棱两可的因素大多与指针、内存和数组之间松散的关系有关。当调用方和被调用方都是用 C 或者 C++ 编译的代码时，当然不会有问题是；但是如果增加附加条件的话，大多数语言都没有与它们能够自然地相对应的特征。为了把“接口定义”与“特定实现过程所用到的语言”之间的关联尽可能地断开，我们必须把“定义接口使用的语言”与“定义实现使用的语言”分开来。如果所有参与的各方统一使用一种语言来定义接口，那么就有可能只定义一次接口，然后在必要的时候导出新的、与实现语言相关的接口定义来。COM 提供了这样一种语言，它只用到了基本的、大家都很熟悉的 C 语法，同时加入了某些用来“消除 C 语言中二义性特征”的能力（从而能够把这些特征精确地翻译到其他语言中）。这种语言被称为接口定义语言（Interface Definition Language, IDL）。

## 2.2 IDL

COM IDL 以开放软件基金会（Open Software Foundation, OSF）的 DCE RPC（Distributed Computing Environment Remote Procedure Call，分布式计算环境中的远程过程调用）IDL 为基础。DCE IDL 使我们可以用一种与语言无关的方式来描述远程调用，也使 IDL 编译器能够产生相应的网络代码，在各种各样的网络传输环境中透明地传输所描述的操作。COM IDL 只是在 DCE IDL 的基础上，加入了一些与 COM 相关的扩展，以便支持 COM 中面向对象的特性（比如继承性、多态性）。不巧的是，当通过执行环境（execution context）<sup>1</sup> 或者机器边界（machine boundary）访问 COM 对象时，客户和对象之间的所有通信都使用 MS-RPC（这是 Windows NT 和 Windows 95 中 DCE RPC 的一个实现）作为底层传输机制。

Win32 SDK 包含一个被称为 MIDL.EXE 的 IDL 编译器，它可以解析 COM IDL 文件，产生许多有关的信息文件。如图 2.1 所示，MIDL 产生 C/C++兼容的头文件，其中包含与“原始 IDL 文件中定义的接口”相对应的抽象基类的定义。这些头文件也包含与 C 语言兼容的、基于结构（struct）的定义，这就允许 C 程序也可以访问或者实现 IDL 描述的接口。MIDL 能够自动产生 C/C++头文件，这就意味着我们不应该在 C++中用手工方式定义 COM 接口。这种单点定义接口（即只在一个地点定义接口）的方式可以避免“同一个接口拥有多个不相容的版本”的情况，也不必同时维护多个版本使它们总是保持同步。MIDL 产生的源代码也允许接口被用于跨越线程、进程，甚至跨越机器的情形。第 5 章将会讨论这些代码。最后，MIDL 也产生一个二进制文件，允许其他“可感知 COM（COM-aware）的”环境能够为原始 IDL 文件中定义的接口产生相应的、针对特定语言映射的接口定义。这个二进制文件也被称为类型库（type library），它包含了符号化的 IDL，这是一种更有效的解析形式。类型库往往被作为组件实现的可执行文件的一部分而发放出去，从而使其他的语言比如 Visual Basic、Java 或者 Object Pascal 也可以使用这些组件出来的接口。

为了理解 IDL，我们有必要从两个角度来看待接口：逻辑的角度以及物理的角度。对接口的方法以及它们所执行的操作的讨论，焦点集中在接口的逻辑方面；对内存、堆栈结构、网络数据包或者其他运行时现象的讨论，通常指的是接口的物理方面。接口的某些物理要素可直接从逻辑描述（比如 vtbl 顺序、堆栈中参数的顺序）导出；但是其他一些物理要素（比如数组边界、复杂数据类型的网络表示方式）则要求附加的条件。

IDL 允许接口的设计者使用 C 风格的语法，但是设计者的主要工作却在逻辑领域。

---

<sup>1</sup> COM 规范使用的术语执行环境(execution context)，后来被改名为套间(apartment)。套间既不是线程，也不是进程，但是它与两者有许多共性。第 5 章全面讨论了套间。

然而，IDL 也允许接口设计者精确地指定接口的任何方面，即使这些方面不能够直接从 C 风格的逻辑描述中导出；具体做法是使用正式名称为属性（attribute）的注记。IDL 属性位于方括号中，多个属性之间以逗号作为分割符，即所谓的属性列表，以区别于基本的 IDL 文本流。属性总是出现在相应主题的定义之前。比如，在下面的 IDL 片断中：

```
[  
    v1_enum, helpstring("This is a color!")  
]  
enum COLOR { RED, GREEN, BLUE };
```

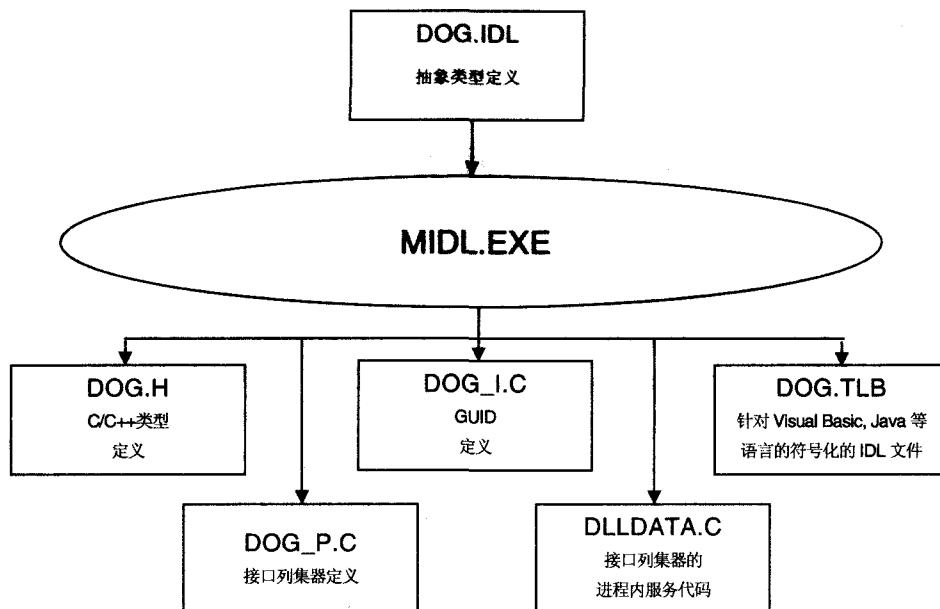


图 2.1 使用 MIDL

v1\_enum 属性应用于枚举（enum）定义 COLOR。该属性告诉 IDL 编译器，COLOR 的网络表示形式应该是 32 位，不是 16 位（16 位是缺省值）。helpstring 属性也应用到 COLOR 上，在最终产生的类型文件中插入字符串“This is a color！”作为该枚举类型的文档描述。如果我们忽略了 IDL 文件的属性，那么 IDL 语法就是简单的 C 语法。IDL 支持结构（struct）、联合（union）、数组、枚举（enum），以及与 C 语言中等价的 typedef 类型定义形式。

当我们在 IDL 中定义 COM 方法时，我们必须要显式地指明调用方或者被调用方将写入或者读出每一个方法参数。这可以通过使用参数属性[in]和[out]来完成：

```
void Method1([in] long arg1,  
            [out] long *parg2,  
            [in,out] long *parg3);
```

对于这个 IDL 片断，调用方应该在 arg1 中传给对象一个值，在 parg3 所指的位置中也要传给对象一个值。方法完成之后，对象应该通过 parg2 和 parg3 所指示的位置传回给调用者相应的结果。这意味着对于下面的调用序列：

```
long arg2 = 20, arg3 = 30;
p->Method1(10,&arg2,&arg3);
```

对象通过 parg2 接收到的实际值并不保证是 20。如果对象与调用方运行在同一个执行环境中，并且双方都是用 C++ 实现的，那么在方法入口处，\*parg2 将真正包含值 20。然而，如果客户从另外不同的执行环境中访问这个对象，或者某一方使用的编程语言对“只引出（out-only）”参数的初始化过程做了优化，那么调用者的初始信息将会丢失。

## 2.3 方法和结果

方法的执行结果正是 COM 中逻辑世界与物理世界分离的一个方面。几乎所有的 COM 方法都会返回一个 HRESULT 类型的错误号。使用统一的错误类型允许 COM 的远程结构可以重载方法的错误值，也可以指示调用过程中的通信错误，只要为 RPC 错误保留一段返回值即可。HRESULT 是 32 位整数，它向调用者的运行时环境提供关于“发生了什么类型的错误”的信息，比如网络错误、服务器失败等。对于许多 COM 兼容的实现语言（比如 Visual Basic、Java）而言，这些 HRESULT 被运行时库或者虚拟机截取，然后被映射为语言中特定的异常（exception）。



图 2.2 HRESULT

如图 2.2 所示，HRESULT 被分为三部分：严重程度位（severity code）、操作码（facility code）、信息码（information code）。严重程度位指示了操作成功还是失败，操作码指示了 HRESULT 对应于什么技术，信息码是在给定的严重程度和相应的技术情况下精确的结果值。SDK 头文件定义了两个宏可以简化 HRESULT 有关的代码：

```
#define SUCCEEDED(hr) (long(hr) >= 0)
#define FAILED(hr) (long(hr) < 0)
```

这两个宏充分利用了“当我们把 HRESULT 当作有符号的整数时，严重程度位也就是符号位”这一事实。

SDK 头文件也包含了所有标准 HRESULT 的定义。这些 HRESULT 的符号名对应于 HRESULT 的三个组成部分，格式如下：

```
<facility>_<severity>_<information>
```

例如，HRESULT 值 STG\_S\_CONVERTED 表明操作码为 FACILITY\_STORAGE（这说明这个结果与结构化存储或者永久性有关）；严重程度位为 SEVERITY\_SUCCESS（意思是这个调用能够成功地执行指定的操作），在这种情况下，该操作已经成功地把底层文件转换为支持结构化存储的文件。如果 **HRESULT** 是通用的，并不与特定的技术相关，则可以使用 **FACILITY\_NULL**，其符号名并不包含操作码前缀。一些通用的 **FACILITY\_NULL** HRESULT 如下：

- S\_OK——一般操作，成功执行
- S\_FALSE——成功地返回逻辑错误
- E\_FAIL——一般性失败
- E\_NOTIMPL——方法没有实现
- E\_UNEXPECTED——在不准确的时间调用了方法

**FACILITY\_ITF** 用于与接口相关的 HRESULT，对于用户自定义的 HRESULT 来说，**FACILITY\_ITF** 也是唯一合法的操作码。在特定接口的环境中，**FACILITY\_ITF** 值只需要保证唯一性即可。标准头文件定义了 **MAKE\_HRESULT** 宏，可根据三个域的信息组成用户自定义的 HRESULT。例如：

```
const HRESULT CALC_E_IAMHOSED =
    MAKE_HRESULT(SEVERITY_ERROR,
                  FACILITY_ITF, 0x200 + 15);
```

使用用户自定义的 HRESULT 有一个惯例，那就是选择大于 0x200 的信息码值，以便避免重用了系统已经使用的 HRESULT 值。尽管这并不是至关重要的，但是这样做可以避免为标准接口中已经具有特定含义的值再赋予新的含义。例如，大多数 HRESULT 都有友好的文本描述，在运行时使用 API 函数 **FormatMessage** 可以解析得到这些描述信息。我们只需选择与“系统定义的 HRESULT”没有冲突的值，就不会发生错误消息不正确的情况。

为了让方法返回一个与“方法的物理 HRESULT”不相关的逻辑结果，COM IDL 支持 **retval** 参数属性。**retval** 属性含义是，相关联的物理方法参数实际上是操作的逻辑结果，在支持 **retval** 的环境中，该参数应该被映射为操作的结果。例如，给定下面的 IDL 方法定义：

```
HRESULT Method2([in] short arg1,
                [out, retval] short *parg2);
```

在 Java 语言中，应该被映射为下面的函数：

```
public short Method2(short arg1);
```

而在 Visual Basic 中，方法定义如下：

```
Function Method2(arg1 as Integer) As Integer
```

因为 C++ 没有专门的运行时库来支持 COM 接口的访问操作，所以 Microsoft C++ 把这个方法映射成下面的函数：

```
virtual HRESULT __stdcall Method2(short arg1,
                                  short *parg2) = 0;
```

这意味着下面的 C++ 客户代码：

```
short sum = 10;
short s;
HRESULT hr = pItf->Method2(20,&s);
If (FAILED(hr))
    throw hr;
sum += s;
```

大致上等同于下面的 Java 代码：

```
short sum = 10;
short s = itf.Method2(20);
sum += s;
```

如果该方法返回的 HRESULT 指示一个不正常的结果，那么 Java 虚拟机将把 HRESULT 映射为 Java 异常。C++ 代码片断必须要手工检查方法返回的 HRESULT，然后相应地处理不正常结果。

## 2.4 接口和 IDL

IDL 中的方法定义只是经过简单注记的 C 函数原型。IDL 中的接口定义要求对 C 进行扩展，因为 C 对接口的概念没有内在的支持。IDL 的 interface 关键字用来作为接口定义的开始。接口定义有四个部分：接口名字、基接口名字、接口体和接口属性。接口体只是一组方法定义的简单集合，并且它支持类型定义语句。接口定义如下：

```
[ attribute1,attribute2,...]
interface IThisInterface: IBaseInterface {
    typedef1;
    typedef2;
    :
```

```

method1;
method2;
:
}

```

每个 COM 接口必须要有两个 IDL 属性。[object] 属性是必需的，它说明该接口定义是一个 COM 接口，而不是 DCE 风格的接口。第二个非可选属性指明了接口的实质名字（在前面的 IDL 片断中，IThisInterface 是接口的逻辑名字）。

为了理解为什么 COM 接口需要一个不同于逻辑名字的实质名字，请考虑下面的情形。两个开发人员各自独立地决定开发一个接口，用来作为手持计算器的模型。在给定的共同问题域中，两个接口定义极有可能非常类似，但是方法定义的实际顺序以及每个方法的原型都有可能存在一些细微的区别。然而，两个开发人员可能都会选择同一个逻辑名字 ICalculator。

在特定的最终用户的机器上，针对第一个开发人员制定的接口的客户程序，有可能会与第二个开发人员创建的对象一起运行（安装在同一台机器上）。因为两个接口共享同样的逻辑名字，所以如果客户只是简单地使用字符串“ICalculator”来询问对象是否支持这个接口，那么对象将会满足客户的请求，并返回给客户一个非空的接口指针。然而，客户眼中的 ICalculator 与对象眼中的 ICalculator 是有冲突的，客户得到的接口指针与它所期望的接口指针是不兼容的。尽管这两个接口共享了同一个逻辑名字，但是它们是完全不同的接口。

为了消除这种名字冲突，所有的 COM 接口在设计时候都被分配一个二进制名字，这就是接口的实质名字。这些实质名字被称为全局唯一标识符（Globally Unique Identifier，GUID），韵母读音与“squid”相同<sup>2</sup>。COM 大量使用 GUID 来命名各种静态的实体，比如接口或者实现。GUID 是 128 位的大数，可以保证在时间和空间两个方面都是唯一的。COM GUID 以 DCE RPC 中用到的 UUID（Universally Unique Identifier）为基础。当 GUID 用来命名 COM 接口时，它常常被称为接口 ID（interface ID，IID）。COM 中的实现也用 GUID 来命名，在这种情况下，GUID 被称为类 ID（class ID，CLSID）。如果用文本来表述，则 GUID 往往按照下面统一的格式来显示：

BDA4A270-A1BA-11D0-8C2C-0080C73925BA

这 32 个 16 进制数字代表了 GUID 的 128 位值。用 GUID 来命名接口和实现是非常重要的，这样可以避免多个组件之间产生名字冲突。

为了创建一个新的 GUID，COM 暴露了一个 API 函数，可以用来产生新的 128 位值。该函数使用了非集中式的唯一性算法，所以理论上可以保证结果值不会重复出现。函数原型如下：

---

<sup>2</sup> GUID 确切的发音曾经是 COM 开发人员中间热烈讨论的一个话题，尽管 COM 规范中指出 GUID 的韵母与“fluid”相同，而不是“squid”，但是我认为 COM 规范所说的是错误的，“languid”一词就是一个先例。

```
HRESULT CoCreateGuid(GUID *pguid);
```

`CoCreateGuid` 使用的算法用到了本地机器的网络接口地址、时钟的时间信息，以及一对用来“补偿时钟分辨率”和“非正常系统时钟变化”的永久计数器（例如日光节省时间、系统时钟的手工调整等）。如果当前机器并没有网络接口，那么它将产生一个统计意义上的唯一值，并且 `CoCreateGuid` 将返回一个不同的 HRESULT，以便指明这个值只是统计意义上的全局唯一值；如果这个值只是用在本地机器上的话，那么它也是真正唯一的。尽管有些时候直接调用 `CoCreateGuid` 会非常有用，但是大多数开发人员往往使用 `GUIDGEN.EXE` SDK 工具间接地调用这个函数。图 2.3 显示了 `GUIDGEN` 的使用界面。`GUIDGEN` 调用 `CoCreateGuid`，并且把结果得到的 GUID 放到四种不同的格式中，以便适用于 IDL 或者 C++源代码。当使用 IDL 的时候，第四种格式（标准的文本格式）正是我们所需要的。

为了把接口的实质名字和它在 IDL 中的定义联系起来，我们要使用第二个非可选接口属性`[uuid]`。`[uuid]`属性接受一个参数，即 GUID 的标准文本形式。示例如下：

```
[object, uuid(BDA4A270-A1BA-11d0-8C2C-0080C73925BA)]
interface ICalculator : IBaseInterface {
    HRESULT Clear(void);
    HRESULT Add([in] long n);
    HRESULT Sum([out, retval] long *pn);
}
```

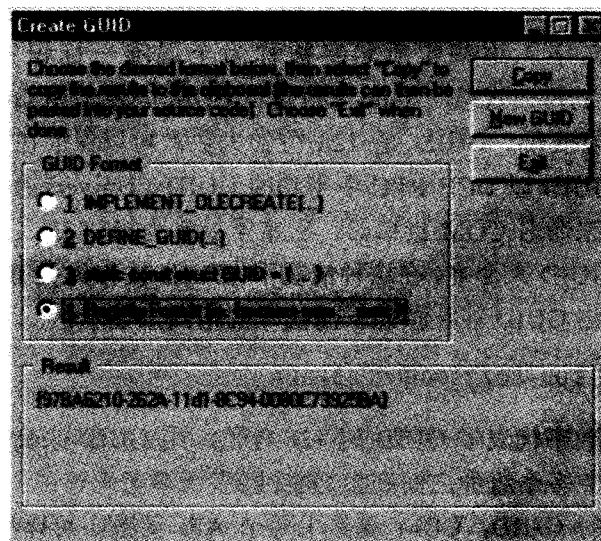


图 2.3 GUIDGEN

当我们在 C 或者 C++中编写程序涉及到接口的实质名字时，给定接口的 IID 只是简单地在接口的逻辑名字前面加上“`IID_`”前缀。例如，接口 `ICalculator` 也有一个 IID，通过 IDL 产生的常量 `IID_ICalculator`，我们可以在程序中访问这个 IID。C++的名字空

间也可以用来处理接口之间的符号名冲突。

因为很少有编译器支持 128 位整数，所以 COM 定义了一个 C 结构，用来表示 GUID 的 128 位值，同时也使用 `typedef` 为 IID 和 CLSID 类型提供了别称：

```
typedef struct _GUID {
    DWORD Data1;
    WORD Data2;
    WORD Data3;
    BYTE Data4[ 8 ];
} GUID;
typedef GUID IID; typedef GUID CLSID;
```

GUID 的内部结构与大多数的程序员没有关系，因为在 GUID 上执行的操作中，唯一有意义的操作是等价性测试（即判断两个 GUID 是否相等）。为了使传送 GUID 类型的函数参数更加高效，COM 还为每种 GUID 类型定义了常量引用别称：

```
#define REFGUID const GUID&
#define REFIID const IID&
#define REFCLSID const CLSID&
```

为了允许对 GUID 值进行比较，COM 提供了等价性测试函数，并为常量 GUID 引用类型重载了“==”操作符和“!=”操作符。

```
inline BOOL IsEqualGUID(REFGUID r1,REFGUID r2)
{ return !memcmp(&r1,&r2,sizeof(GUID)); }
#define IsEqualIID(r1,r2) IsEqualGUID((r1),(r2))
#define IsEqualCLSID(r1,r2) IsEqualGUID((r1),(r2))
inline BOOL operator == (REFGUID r1, REFGUID r2)
{ return !memcmp(&r1,&r2,sizeof(GUID)); }
inline BOOL operator != (REFGUID r1, REFGUID r2)
{ return !(r1 == r2); }
```

实际的 SDK 头文件包含了条件编译指令，可以编译出上述类型 (`typedef`)、宏和内联函数的 C 兼容版本来。

既然接口在运行时的名字是 GUID，而不是字符串，这就意味着前一章讲述的 `Dynamic_Cast` 方法需要重新修订。实际上，整个 `IExtensibleObject` 接口都需要重新考察，并且最终要转换到 COM 中等价的接口：`IUnknown` 上。

## 2.5 IUnknown

COM 接口 `IUnknown` 与上一章定义的 `IExtensibleObject` 接口用于同样的目的。下面

的代码是上一章结束时所得到的 **IExtensibleObject** 接口的最终版本：

```
class IExtensibleObject {
public:
    virtual void *Dynamic_Cast(const char* pszType) = 0;
    virtual void DuplicatePointer(void) = 0;
    virtual void DestroyPointer(void) = 0;
}
```

**Dynamic\_Cast** 方法用于运行时类型发现，它与 C++ 的 **dynamic\_cast** 操作符的功能类似。**DuplicatePointer** 方法用于通知对象“一个接口指针已经被复制了”。**DestroyPointer** 方法用于通知对象“一个接口指针已经被销毁了，它所拥有的资源可以被释放了”。下面是 **IUnknown** 的 C++ 定义：

```
extern "C" const IID IID_IUnknown;
interface IUnknown {
    virtual HRESULT STDMETHODCALLTYPE
        QueryInterface(REFIID riid, void **ppv) = 0;
    virtual ULONG STDMETHODCALLTYPE AddRef(void) = 0;
    virtual ULONG STDMETHODCALLTYPE Release(void) = 0;
};
```

SDK 头文件利用 C 预处理器把符号 **interface** 做成 C++ 的 **struct** 关键字。因为 COM 接口被定义为 **struct** 而不是 **class**，所以就不需要用 **public** 关键字来保证接口的方法是公开可见的。**STDMETHODCALLTYPE** 宏是必需的，以便针对目标平台产生 COM 兼容的堆栈结构。在 Microsoft C++ 编译器下面，当目标平台为 Win32 平台时，这个宏被展开为 **stdcall**。

**IUnknown** 在功能上与 **IExtensibleObject** 等价。**QueryInterface** 方法用于运行时类型发现，也等价于 C++ 的 **dynamic\_cast** 操作符。**AddRef** 方法用于通知对象“一个接口指针已经被复制了”。**Release** 方法用来通知对象“一个接口指针已经被销毁了，对象为该客户保留的所有资源都可以被释放了”。**IUnknown** 和上一章定义的接口之间的主要区别在于，**IUnknown** 使用 **GUID** 来标识运行时的接口类型，而不是字符串。

**IUnknown** 的 IDL 定义可以从 SDK 的 **include** 目录下的 **unknwn.idl** 文件中找到：

```
// unknwn.idl - 系统 IDL 文件

[
    local,
    object,
    uuid(00000000-0000-0000-C000-000000000046)
]
interface IUnknown {
    HRESULT QueryInterface([in] REFIID riid,
                           [out] void **ppv);
```

```

    ULONG AddRef(void);
    ULONG Release(void);
}

```

[local]属性禁止为该接口产生网络代码。这个属性可以放宽 COM 的要求，否则的话，所有的远程方法必须返回 HRESULT。正如后面章节将要讨论的，在处理远程对象的时候，IUnknown 是被特殊对待的。请注意，在 SDK 头文件中找到的实际 IDL 接口定义与上面给出的定义稍微有一点不同。实际的定义往往包含附加的属性，以便对产生的网络代码（与现在的讨论没有关系）进行优化。如果你表示怀疑的话，可以检查最新版本的 SDK 头文件找到完整的定义。

IUnknown 是所有 COM 接口的根源。IUnknown 是唯一一个不从其他 COM 接口派生的 COM 接口。所有其他合法的 COM 接口都必须直接从 IUnknown 派生，或者从其他合法 COM 接口派生，而这个接口本身也必须要么直接从 IUnknown 派生，要么从其他合法 COM 接口派生。这就意味着，所有 COM 接口的 vtbl 都从这三个入口 QueryInterface、AddRef 和 Release 开始。与接口相关的方法都有对应的 vtbl 入口，但是它们位于这三个公共的入口之后。

在 IDL 中，为了从一个接口派生另一个接口，我们要么在同一个文件中定义基接口，要么使用 import 指示符，以保证外部的基接口 IDL 定义在当前范围内是可见的。如下代码所示：

```

// calculator.idl
[object, uuid(BDA4A270-A1BA-11d0-8C2C-0080C73925BA) ]
interface ICalculator : IUnknown {
    import "unknwn.idl"; // 引入 IUnknown 的定义
    HRESULT Clear(void);
    HRESULT Add([in] long n);
    HRESULT Sum([out, retval] long *pn);
}

```

import 语句既可以出现在接口定义的内部（如上所示），也可以在接口定义之外的全局范围内。无论哪种情况，import 语句都是幂等的（**idempotent**），也就是说，一个 IDL 文件可以被引入多次而不会有任何伤害。为了完成继承操作，产生得到的 C/C++头文件将需要与“被引入的 IDL 文件”相对应的 C/C++版本，所以 IDL 文件中的 import 语句将被翻译为目标 C/C++头文件中的#include 语句：

```

// calculator.h - 由 MIDL 产生
// 引入 IUnknown 的定义
#include "unknwn.h"
extern "C" const IID IID_ICalculator;
interface ICalculator : public IUnknown {
    virtual HRESULT STDMETHODCALLTYPE Clear(void) = 0;
    virtual HRESULT STDMETHODCALLTYPE Add(long n) = 0;
}

```

```
    virtual HRESULT STDMETHODCALLTYPE Sum(long *pn) = 0;
}
```

MIDL 编译器也会产生 C 源文件，其中包含原 IDL 文件中所包含的 GUID 的实际定义，如下：

```
// calculator_i.c - 由 MIDL 产生
const IID IID_ICalculator =
{ 0xBDA4A270, 0xA1BA, 0x11d0, { 0x8C, 0x2C,
    0x00, 0x80, 0xC7, 0x39, 0x25, 0xBA } };
```

每个使用该接口的工程要么把 calculator\_i.c 加入到它的 makefile 文件中，要么使用 C 预处理器在某一个 C 或者 C++ 文件中包含 calculator\_i.c。如果不这样做的话，那么符号 IID\_ICalculator 将不会有存储空间存放它的 128 位值，工程将由于存在无法解析的外部符号（unresolved external symbol），而无法完成链接。

COM 并没有限制接口层次（继承关系）的深度，只是要求最终的基接口必须是 IUnknown。下面的 IDL 是完全合法的，也是合理的 COM：

```
import "unknwn.idl";

[object, uuid (DF12E151-A29A-11d0-8C2D-0080C73925BA)]
interface IAnimal : IUnknown {
    HRESULT Eat(void);           //动物——吃
}

[object, uuid(DF12E152-A29A-11d0-8C2D-0080C73925BA)]
interface ICat : IAnimal {
    HRESULT IgnoreMaster(void);   //猫——不认主人
}

[object, uuid(DF12E153-A29A-11d0-8C2D-0080C73925BA)]
interface IDog : IAnimal {
    HRESULT Bark(void);          //狗——吠
}

[object, uuid(DF12E154-A29A-11d0-8C2D-0080C73925BA)]
interface IPug : IDog {
    HRESULT Snore(void);         //哈巴狗——打鼾
}

[object, uuid(DF12E155-A29A-11d0-8C2D-0080C73925BA)]
interface IOldPug : IPug {
    HRESULT SnoreLoudly(void);    // 老哈巴狗——大声打鼾
}
```

但是 COM 限制了接口的继承性：COM 接口不能直接从多个（多于一个）接口继承

过来。例如，下面的代码在 COM 中是不合法的：

```
[object, uuid(DF12E156-A29A-11d0-8C2D-0080C73925BA)]
interface ICatDog : ICat, IDog { // 不合法的多重继承
    HRESULT Meowbark(void);
}
```

COM 禁止多重接口继承有多个不同的理由。一个理由是，多重继承得到的 C++ 抽象基类的二进制表示将不再是编译器无关的。这将使 COM 无法成为“与编译器厂商无关的二进制标准”。另一个理由则源于 COM 和 DCE RPC 之间的紧密关系。限制接口使它只能从一个基接口继承，于是 COM 接口和 DCE RPC 接口之间的映射就可以直接进行。本质上讲，不支持多个基接口并不算是一个限制，因为任何一个实现都会暴露多个接口（如它所希望的那样多）。这就意味着，在实现层次上，以 COM 为基础的 Cat/Dog（猫狗同体）仍然是有可能的：

```
class CatDog : public ICat, public IDog {
    // ...
};
```

如果一个客户要把一个对象看作 Cat/Dog，那么它就必须用 `QueryInterface` 把两种类型的指针绑定到对象上。只要其中任一个 `QueryInterface` 调用失败，那么这个对象就不是 Cat/Dog，客户可以按它的意愿来处置这种情况。因为所有的实现都可以暴露多个接口，所以 COM 在“禁止接口具有多个基接口”方面，还是有一些语义或者类型信息的损失。

COM 支持一种记号表示技术，利用这项技术可以表达“一个对象中哪些接口是可以使用的”。这项技术仍然坚持 COM 中“接口与实现分离”的哲学，并且也没有泄露对象的任何实现细节，而只是列出了它所暴露的接口表。

图 2.4 显示了 CatDog 类的标准表示法。注意，从这个图中我们可以得出的结论只有一条：除非发生大的灾难，CatDog 将暴露四个接口：`ICat`、`IDog`、`IAnimal` 和 `IUnknown`。

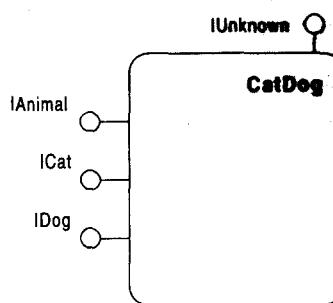


图 2.4 COM 标准表示法

## 2.6 资源管理和 IUnknown

就像上一章的 `DuplicatePointer` 和 `DestroyPointer` 一样, `IUnknown` 的 `AddRef` 和 `Release` 方法也有一个非常简单的协定, 接口指针的所有用户都必须遵守这个协定。当多个接口指针可能(也可能不是)指向同一个对象时, 这些规则使得客户可以从管理对象生命周期的重任中解脱出来。客户只需要针对它所碰到的每个接口指针, 都遵从简单的 `AddRef/Release` 规则, 对象将会管理它自己的生命周期。

COM 规范对于引用计数规则有非常明确的定义。理解这些定义背后的动机, 对于精通 C++ COM 程序设计是非常重要的。COM 的引用计数规则可以精简为以下三个简单的公理:

1. 当一个非空的接口指针从一个内存位置被拷贝到另一个内存位置时, 应该要调用 `AddRef`, 以便通知对象又有附加的引用发生了。
2. 对于已经包含非空接口指针的内存位置来说, 在重写该内存位置之前, 必须要先调用 `Release`, 以便通知对象“这个引用已经被销毁了”。
3. 如果你对两个或者多个内存位置之间的关系有特殊的理解的话, 那么多余的 `AddRef` 和 `Release` 调用可以被优化掉。

这里第三条关于特殊知识的公理之所以存在, 主要是为了将“可能会混淆的状态”映射到健全和敏感的代码片断(例如临时的堆栈变量和编译器产生的代码中的变量并不需要被引用计数)。我们可以花上几个月的时间, 在一个程序中找到显式接口指针变量之间的特殊关系, 然后优化掉冗余的 `AddRef` 和 `Release`, 但是这样做显然是很荒唐的。消除这些冗余调用的益处并不是非常重要的, 最差的情况下, 比如通过 14.4kbps 的传输介质在 8500 英里之外访问一个对象, 这些冗余调用并不会离开调用者的线程, 而且也不会要求太多的指令。

根据以上三条与引用计数和接口指针有关的公理, 我们可以把这些公理变成实际的编程指导, 以便确定什么时候应该调用 `AddRef` 和 `Release`, 什么时候不应该调用。下面是一些比较通用的、要求调用 `AddRef` 方法的情形:

- A1. 当把一个非空接口指针写到局部变量中时。
- A2. 当被调用方把一个非空接口指针写到方法或者函数的 `[out]` 或者 `[in,out]` 参数中时。
- A3. 当被调用方返回一个非空接口指针作为函数的实际结果时。
- A4. 当把一个非空接口指针写到对象的一个数据成员中时。

下面是一些比较通用的、要求调用 `Release` 方法的情形:

- R1. 在改写一个非空局部变量或者数据成员之前。

**R2.** 在离开非空局部变量的作用域（scope）之前。

**R3.** 当被调用方要改写方法或者函数的[in,out]参数，并且参数的初始值为非空时。

注意，[out]参数往往被假定“输入时为空值”，所以被调用方永远也不必释放[out]参数。

**R4.** 在改写一个对象的非空数据成员之前。

**R5.** 在离开一个对象的析构函数之前，并且这时还有一个非空接口指针作为数据成员。

还有一种很常见的特殊情况是，当把接口指针作为[in]参数传给函数时，可以适用前面给出的第3条关于特殊知识的规则：

**S1.** 当调用方把一个非空接口指针通过[in]参数传给一个函数或者方法时，既不需要调用 AddRef，也不需要调用 Release，因为在调用堆栈中，临时变量的生命周期只是“用于初始化形式参数”的表达式的生命周期的一个子集。

这十条指导规则几乎涵盖了 COM 编程过程中屡屡出现的各种情况，值得记下来。

为了使 COM 引用计数的规则更加直观，我们假定有一个全局函数，它返回一个指向某个对象的接口指针：

```
void GetObject([out] IUnknown **ppUnk);
```

而另一个全局函数针对某个对象执行一些很有用的工作：

```
void UseObject([in] IUnknown *pUnk);
```

下面的代码利用这两个函数来操纵某些对象，并返回一个接口指针给它的调用者。代码的注解部分给出了对应语句所适用的指导规则。代码如下：

```
void GetAndUse(/* [out] */ IUnknown ** ppUnkOut) {
    IUnknown *pUnk1 = 0, *pUnk2 = 0;
    *ppUnkOut = 0; // R3

    // 获取一个或两个对象的指针
    GetObject(&pUnk1); // A2
    GetObject(&pUnk2); // A2

    // 将 punk2 指向第一个对象
    if (pUnk2) pUnk2->Release(); // R1
    if (pUnk2 = pUnk1) pUnk2->AddRef(); // A1

    // 将 punk2 传给其他函数
    UseObject(pUnk2);

    // 将 punk2 返回给使用 ppUnkOut 参数的调用方
    if (*ppUnkOut = pUnk2) (*ppUnkOut)->AddRef(); // A2
```

```
// 超出作用域，因此清除
if (pUnk1) pUnk1->Release(); // R2
if (pUnk2) pUnk2->Release(); // R2
}
```

很重要的一点是，上述代码中 A2 指导规则用到了两次，但是从两个完全不同的角度来应用这条规则。当调用 `GetObject` 时，这部分代码作为调用方，而 `GetObject` 函数的实现作为被调用方。这意味着，`GetObject` 的实现有责任调用 [out] 参数的 `AddRef`。当改写 `ppUnkOut` 所指的内存时，这段代码作为被调用方，所以在把控制返回给调用方之前，必须要正确地调用接口指针的 `AddRef`。

关于 `AddRef` 和 `Release` 还有一些微妙之处值得讨论。`AddRef` 和 `Release` 的原型都返回一个 32 位无符号整数。这个整数反映出在执行了 `AddRef` 或者 `Release` 操作之后所有未完结计数的总数。然而，由于各种原因，比如多线程、远程访问和多处理器结构等，这个值并不能保证一定精确表达了所有未完结接口指针的总数，因此客户应该忽略这个整数，除非是在调试过程中出于诊断的目的。

唯一有意义的结果是 `Release` 返回 0 的时候。无论什么情况下，`Release` 返回 0 必定表示这个对象已经不再有效了。然而，反过来则不一定成立，也就是说，当 `Release` 返回非 0 值时，我们并不能假设该对象一定仍然是有效的。事实上，一旦在同一个接口指针上调用 `Release` 的次数与调用 `AddRef` 的次数相等的话，那么这个接口指针就会无效，它不再保证指向一个有效的对象。虽然有可能由于对象还存在其他的未完结指针，故对象仍然是有效的，但这只是一种意外情形，极有可能在最为要紧的时候情况发生变化了。要确保“接口指针在释放之后不再被使用”的一种策略是，在调用了 `Release` 方法之后立即将其设置为空（`null`）。如以下代码所示：

```
inline void SafeRelease(IUnknown * &rpUnk) {
    if (rpUnk) {
        rpUnk->Release();
        rpUnk = 0; // 按引用传递 rpUnk
    }
}
```

使用这项技术时，在接口被释放之后再使用接口指针就会立即引发“访问违例（access violation）”错误。并且，这样的错误一定可以再现，所以在开发过程中我们可以很可靠地捕捉到这样的错误。

与 `AddRef` 和 `Release` 有关的另一个微妙之处发生在退出一块代码时。前面给出的 `GetAndUse` 函数只有一个退出点，这就意味着，在函数尾部释放接口指针的操作总是在函数完成之前先执行。如果由于某些原因函数在到达这些语句之前先退出了，比如由于一个显式的 `return` 语句，或者更为严重的是，由于一个未被处理的 C++ 异常，那么这最后部分的语句就会被绕过去，于是未被释放的接口指针所拥有的任何资源都将被遗留在

系统中，在客户程序的生命周期范围内不可能再收回这些资源。这隐含着，COM 接口指针一定要小心处理，特别是在用到 C++ 异常的环境中。这和我们经常见到的其他系统资源，比如信号量（semaphore）或者动态分配的内存没有什么不同。本章后面将要讨论到 COM 智能指针，它可以确保在所有的情况下 Release 都会被调用到。

## 2.7 类型强制转换和 IUnknown

上一章讲述了在动态组合的系统中需要“运行时类型发现”的动机。C++ 语言通过 dynamic\_cast 操作符提供了合理的“动态类型发现”机制。虽然这个语言特性在每个编译器上都有自己私有的实现方式，但是上一章已经从另一个角度实现了这个概念，这就是在每个接口上加入一个显式方法，由它执行等价于 dynamic\_cast 的语义功能。COM 的 IUnknown 接口也有这样一个方法，被称为 QueryInterface。下面是 QueryInterface 的 IDL 描述：

```
HRESULT QueryInterface([in] REFIID riid,
                      [out] void **ppv);
```

第一个参数（riid）是被请求的接口的实质名字。第二个参数（ppv）指向一个接口指针变量，当函数成功返回时，它包含客户所请求的接口指针。

在响应 QueryInterface 请求时，如果对象并不支持被请求的接口类型，那么 QueryInterface 就必须先把 \*ppv 设置为 null，然后返回 E\_NOINTERFACE。如果对象支持被请求的接口，那么它必须用被请求类型的指针改写 \*ppv，然后返回 HRESULT 值 S\_OK。因为 ppv 是一个 [out] 参数，所以 QueryInterface 的实现要在控制返回给客户之前，必须要针对结果指针调用 AddRef（参考本章前面 A2 指导规则）。这个 AddRef 调用必须要与客户端的 Release 调用匹配。下面的代码显示了以 C++ 的 dynamic\_cast 操作符作为基本手段而实现的运行时类型发现功能，代码的内容以本章前面描述的 Dog/Cat 类型层次结构作为基础：

```
void TryToSnoreAndIgnore(/* [in] */ IUnknown *pUnk) {
    IPug *pPug = 0;
    pPug = dynamic_cast<IPug*>(pUnk);
    if (pPug) // 对象是 Pug 兼容的
        pPug->Snore();

    ICat *pCat = 0;
    pCat = dynamic_cast<ICat*>(pUnk);
```

```

if (pCat) // 对象是 Cat 兼容的
    pCat->IgnoreMaster ();
}

```

如果传递给这个函数的对象既是 ICat 兼容的，也是 IPug 兼容的，那么这两个功能就都可以使用。如果对象实际上不是 ICat 或者 IPug 兼容的，那么这个函数只是简单地忽略掉对象不支持的功能。下面的代码使用了 QueryInterface，但是在语义上与上面的代码等价：

```

void TryToSnoreAndIgnore(/* [in] */ IUnknown *pUnk) {
    HRESULT hr;
    IPug *pPug = 0;
    hr = pUnk->QueryInterface(IID_IPug, (void**)&pPug);
    if (SUCCEEDED(hr)) {// 对象是 Pug 兼容的
        pPug->Snore();
        pPug->Release(); // R2
    }

    ICat *pCat = 0;
    hr = pUnk->QueryInterface(IID_ICat, (void**)&pCat);
    if (SUCCEEDED(hr)) {// 对象是 Cat 兼容的
        pCat->IgnoreMaster ();
        pCat->Release(); // R2
    }
}

```

虽然两段代码有明显的语法区别，但是唯一重要的差别在于，基于 QueryInterface 的版本符合 COM 的引用计数规则。

有关 QueryInterface 和它的用法也存在一些微妙之处。QueryInterface 只能返回指向同一个 COM 对象的指针。第 4 章将专门讲述这个论点的每一个细微之处。然而，值得提前注意的是，客户一定不要把 AddRef 和 Release 看作是针对整个对象的操作。相反，这两个操作应该被看成是针对接口指针的操作。这意味着下面的代码是非法的：

```

void BadCOMCode(/* [in] */ IUnknown *pUnk) {
    ICat *pCat = 0;    IPug *pPug = 0;
    HRESULT hr;
    hr = pUnk->QueryInterface(IID_ICat, (void**)&pCat);
    if (FAILED(hr)) goto cleanup;
    hr = pUnk->QueryInterface(IID_IPug, (void**)&pPug);
    if (FAILED(hr)) goto cleanup;
    pPug->Bark(); pCat->IgnoreMaster();

cleanup:
    if (pCat) pUnk->Release(); // 在 QueryInterface 中对 pCat 调用了 AddRef
    if (pPug) pUnk->Release(); // 在 QueryInterface 中对 pDog 调用了 AddRef
}

```

尽管事实上 pCat、pPug 和 pUnk 都指向同一个对象，但是客户用 pUnk 上的 Release 调用来抵消发生在 pPug 和 pCat 上的 AddRef（当调用 QueryInterface 时），这是不合法的。正确的代码应该如下：

```
cleanup:
if (pCat) pCat->Release(); // 使用了 AddRef 的指针
if (pPug) pPug->Release(); // 使用了 AddRef 的指针
```

在这里，Release 调用作用在与接收 AddRef 调用完全一致的接口指针上（AddRef 是在 QueryInterface 函数返回之前被调用的）。这个要求使得开发人员在实现一个对象时有了更大的灵活性。例如，一个对象有可能选择基于每个接口的引用计数规则，以便对于“仅用于对象上某个特定接口的资源”能够更加积极有效地回收和使用。

与 QueryInterface 有关的另一个微妙之处在于它的第二个参数，它的类型为 void \*\*。具有讽刺意味的是，QueryInterface 作为 COM 类型系统的基础，在 C++ 中它本身却具有一个类型不安全 (type-unsafe) 的原型：

```
HRESULT __stdcall QueryInterface(REFIID riid,
                                void** ppv);
```

正如前面所讲述的，客户在调用 QueryInterface 时，要向对象提供一个指向接口指针的指针作为第二个参数，连同一个 IID 用来标识它所期望的接口指针的类型：

```
IPug *pPug = 0;
hr = punk->QueryInterface(IID_IPug, (void**)&pPug);
```

不幸的是，下面的代码对于 C++ 编译器来说看起来是正确的：

```
IPug *pPug = 0;
hr = punk->QueryInterface(IID_ICat, (void**)&pPug);
```

而且，下面这种更为微妙的情形也能正确地通过编译：

```
IPug *pPug = 0;
hr = punk->QueryInterface(IID_IPug, (void**)pPug);
```

我们知道，继承性规则并不适用于指针，所以下面这种 QueryInterface 定义形式也不能使问题有所好转：

```
HRESULT QueryInterface(REFIID riid, IUnknown** ppv);
```

因为隐式向上转换 (implicit upcasting) 只适用于实例，以及指向实例的指针，不适用于“指向实例的指针的指针”，例如：

```
IDerived **ppd;
IBase **ppb = ppd; // 不合法
```

这项限制也适用于指向指针的引用。下面的 `QueryInterface` 定义形式对于客户会更加方便：

```
HRESULT QueryInterface(const IID& riid, void* ppv);
```

因为它允许客户放弃类型转换。但是不幸的是，这个方案并不会减少错误的数量，前面所说的两种错误仍然存在，而且，由于消除了类型转换的需要之后，同时也消除了视觉上的指示，这有可能会带来更大的危险。对于 `QueryInterface` 所需要的语义来说，Microsoft 选择的参数类型是合理的，尽管它并不是类型安全的。要想避免以上讲述的与 `QueryInterface` 有关的错误，最简单的办法是确保 `IID` 与 `QueryInterface` 的第二个参数接口指针的类型一致。实际上，`QueryInterface` 的第一个参数描述了第二个参数的指针类型的“形状”。这个关系可以在编译时通过下面的 C 预处理器宏强迫实施：

```
#define IID_PPV_ARG(Type, Expr) IID_##Type, \
    reinterpret_cast<void**>(static_cast<Type **>(Expr))
```

有了这个宏之后，<sup>3</sup> 编译器将保证后续 `QueryInterface` 调用中用到的表达式总会具有正确的类型，同时也用到了适当的间接性：

```
IPug *pPug = 0;
hr = punk->QueryInterface(IID_PPV_ARG(IPug, &pPug));
```

这个宏关闭了因 `void **` 参数带来的漏洞，同时又没有增加任何运行时开销。

## 2.8 实现 IUnknown

在讲述了客户端的使用模式之后，`IUnknown` 各个方法的实现就非常直接了。假设仍然采用前面描述过的 Dog/Cat 类型层次。为了定义一个同时实现了 `IPug` 和 `ICat` 接口的 C++类，我们只要简单地在基类列表中加入每个接口即可：

```
class PugCat : public IPug, public ICat
```

在使用继承的时候，C++编译器可以保证派生类的二进制布局将与每个接口的二进制布局兼容。对于 `PugCat` 类来说，这意味着所有的 `PugCat` 对象都将包含一个 `vptr`，它指向一个与 `IPug` 兼容的 `vtbl`。`PugCat` 对象还将包含另一个 `vptr`，它指向第二个与 `ICat` 兼容的 `vtbl`。图 2.5 显示了作为基类的接口与对象布局之间的关系。

---

<sup>3</sup> 这要归因于我和 Tye McQueen 在一次 COM 研讨会上的讨论。

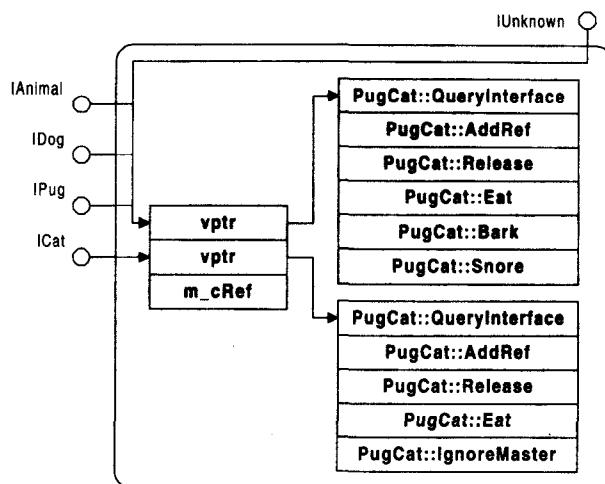


图 2.5 PugCat

因为 COM 接口定义中的所有成员函数都是纯虚函数，所以派生类必须为所有出现在它的接口中的方法都要提供相应的实现。两个或者多个接口中公共的方法（比如 `QueryInterface`、`AddRef` 等）只需实现一次，因为编译器将会初始化每个 `vtbl`，使它们指向同一个方法实现。这是在 C++ 中使用多重继承自然产生的副作用。

下面的类定义所产生的对象同时支持 `IPug` 和 `ICat` 接口：

```
class PugCat: public IPug, public ICat {
    LONG m_cRef;
protected:
    virtual ~PugCat(void);
public:
    PugCat (void);
// IUnknown 方法
    STDMETHODIMP QueryInterface(REFIID riid,
                                void **ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
// Ianimal 方法
    STDMETHODIMP Eat (void);
// Idog 方法
    STDMETHODIMP Bark(void);
// Ipug 方法
    STDMETHODIMP Snore(void);
// Icat 方法
    STDMETHODIMP IgnoreMaster(void);
};
```

注意，这个类必须实现它所继承的所有基类中定义的每一个方法，以及在隐含的基类（比如 IDog、IAnimal）中定义的方法。使用宏 STDMETHODIMP 和 STDMETHODIMP\_ 是必需的，以便产生与 COM 兼容的堆栈结构。当我们在 Win32 平台上使用 Microsoft C++ 编译器时，SDK 头文件定义这两个宏如下：

```
#define STDMETHODIMP HRESULT __stdcall
#define STDMETHODIMP_(type) type __stdcall
```

SDK 头文件也定义了宏 STDMETHOD 和 STDMETHOD\_，在没有 IDL 编译器的情况下定义接口就可以使用这两个宏。在主流的 COM 程序设计中，这两个宏并不需要。

AddRef 和 Release 的实现过程非常直接。m\_cRef 记录了该对象具有多少个未完结的接口指针。类的构造函数初始化引用计数，将其初始化为 0：

```
PugCat:: PugCat (void)
: m_cRef(0) // 将引用计数初始化为 0
{ }
}
```

AddRef 的实现把引用计数加 1，表示调用方已经复制了一个接口指针。更新之后的引用计数值被返回，以便于调试和诊断：

```
STDMETHODIMP_(ULONG) AddRef(void) {
    return ++m_cRef;
}
```

Release 的实现只是简单地把引用计数减 1，表示有一个接口指针已经被销毁了，并且当引用计数值到达 0 的时候，它要采取适当的行动。对于分配在堆（heap）中的对象，这意味着要调用 delete 操作符以便销毁这个对象：

```
STDMETHODIMP_(ULONG) Release(void) {
    LONG res = --m_cRef;
    if (res == 0)
        delete this;
    return res;
}
```

在上面的代码中，使用一个临时变量把修改后的引用计数值缓存起来是必要的，因为在对象已经被销毁之后再引用这个对象的数据成员将是非法的。

请注意，前面给出的 AddRef 和 Release 实现使用了 C++ 本身的增 1 和减 1 操作符。作为初始构想的实现方案，这样做完全是合理的，因为 COM 不会允许多个线程访问同一个对象，除非对象实现者显式地允许多个线程访问这个对象（为什么实现者要这样做以及如何做到这一点，将在第 5 章详细讲述）。对于将要在多线程环境中被访问的对象

而言，Win32 的 `InterlockedIncrement` 和 `InterlockedDecrement` 函数可以用来对引用计数以原子方式（atomically，不会被中断）进行调整：

```
STDMETHODIMP_(ULONG) AddRef(void) {
    return InterlockedIncrement(&m_cRef);
}
STDMETHODIMP_(ULONG) Release(void) {
    LONG res = InterlockedDecrement(&m_cRef);
    if (res == 0)
        delete this;
    return res;
}
```

这段代码比起前面直接使用 C++本身操作符的版本来说，效率要低一些。然而，通常情况下，使用稍微低效的 `InterlockedIncrement/InterlockedDecrement` 版本总是比较合理的，因为它们对于所有的环境都是很安全的，而且开发人员不必为不同的环境维护两套本质上几乎完全相同的代码版本。

上面给出的 `AddRef` 和 `Release` 实现假定这个对象是用 C++的 `new` 操作符被分配在堆中的。类定义把析构函数做成了保护（`protected`）操作，从而保证不可能用其他的方式来定义类的实例。然而，有时候我们希望对象不是在堆中被分配的。对于这些对象，在最后的 `Release` 调用中调用 `delete` 将会发生灾难性的后果。因为对象维护引用计数的唯一理由是要调用 `delete this`，所以针对不是从堆中分配得到的对象而言，可以将引用计数操作进行优化，所以下面的代码是合法的：

```
STDMETHODIMP_(ULONG) GlobalVar::AddRef(void) {
    return 2; // 任何非 0 值都合法
}

STDMETHODIMP_(ULONG) GlobalVar::Release(void) {
    return 1; // 任何非 0 值都合法
}
```

我们看到，这个实现用到了这样一个事实：`AddRef` 和 `Release` 的返回值只是一个参考值，并不是精确值。

有了 `AddRef` 和 `Release` 的实现之后，在 `IUnknown` 方法中，唯一还没有被实现的就是 `QueryInterface`。`QueryInterface` 的实现必须要在对象的类型层次结构上来回转换，并且使用静态的类型转换功能，为它所支持的接口返回正确的指针类型。针对前面显示的 `PugCat` 类定义，下面的代码可以作为 `QueryInterface` 的一个正确实现：

```
STDMETHODIMP PugCat::QueryInterface(REFIID riid,
                                     void **ppv) {
    assert(ppv != 0); // 或者在生成中返回 E_POINTER
    if (riid == IID_IPug)
```

```

        *ppv = static_cast<IPug*>(this),
    else if (riid == IID_IDog)
        *ppv = static_cast<IDog*>(this);
    else if (riid == IID_IAnimal) // cat 还是 pug?
        *ppv = static_cast<IDog*>(this);
    else if (riid == IID_IUnknown) // cat 还是 pug?
        *ppv = static_cast<IDog*>(this);
    else if (riid == IID_ICat)
        *ppv = static_cast<ICat*>(this);
    else { // 不支持的接口
        *ppv = 0;
        return E_NOINTERFACE;
    }
    // 如果到达此点*ppv 是非空的
    // 一定调用了 AddRef(规则 A2)
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}

```

建议使用 `static_cast` 进行类型转换，而尽量不要用下面传统的 C 风格的类型转换：

```
*ppv = (IPug*)this;
```

因为若转换的目标类型并不对应于实际的基类，那么 `static_cast` 版本将会引发一个编译时错误。(这当然比运行时错误强多了。)

注意，在上面给出的 `QueryInterface` 实现中，当某一个请求被多个基接口支持（比如 `IUnknown`、`IAnimal`）时，类型转换操作必须明确地选择一个更为精确的基类。对于 `PugCat` 类来说，下面的代码看上去正确，但实际上无法通过编译：

```

if (riid == IID_IUnknown)
    *ppv = static_cast<IUnknown*>(this);

```

正如上一章讲述 `FastString` 和 `IExtensibleObject` 时所显示的，上面代码之所以不能编译是因为转换操作存在二义性，有多个基类可以满足转换要求。相反，`QueryInterface` 实现必须选择一个更为精确的类型，然后实施转换，例如：

```

if (riid == IID_IDog)
    *ppv = static_cast<IDog*>(this);

```

或者

```

if (riid == IID_ICat)
    *ppv = static_cast<ICat*>(this);

```

对于 `PugCat` 的实现来说，这两段代码都是合法的。但是前者更为可取，因为当使

用最左边的基类时，大多数编译器产生的代码效率更高一些。<sup>4</sup>

## 2.9 使用 COM 接口指针

C++程序员必须显式地使用 `IUnknown` 的方法，因为 COM 所对应的 C++语言映射并没有在客户代码和对象代码之间提供一个运行时层(runtime layer)。因此，`IUnknown` 就是所有 COM 程序员之间相互进行通信的一组承诺。一般来说，这为 C++程序员带来了很大的好处，因为 C++产生的代码其性能比那些需要一个运行时层才能处理 COM 的语言更有效率。

与 C++程序员不同的是，Visual Basic 和 Java 程序员永远也不会看到 `QueryInterface`、`AddRef` 或者 `Release`。对于这两种语言来说，`IUnknown` 的细节完全被隐藏在每种语言所支持的虚拟机后面了。在 Java 中，`QueryInterface` 被简单地映射为一个类型转换：

```
public void TryToSnoreAndIgnore(Object obj) {
    IPug pug;
    try {
        pug = (IPug)obj; // 虚拟机调用 QueryInterface
        pug.Snore();
    } catch (Throwable ex) {
        // 忽略方法或 QueryInterface 失败
    }
    ICat cat;
    try {
        cat = (ICat)obj; // 虚拟机调用 QueryInterface
        cat.IgnoreMaster();
    } catch (Throwable ex) {
        // 忽略方法或 QueryInterface 失败
    }
}
```

Visual Basic 并不要求客户进行类型转换，相反，当一个接口指针被分配给一个类型不兼容的变量时，Visual Basic 虚拟机会在后台代表客户调用 `QueryInterface`。代码如下：

```
Sub TryToSnoreAndIgnore(obj as Object)
    On Error Resume Next '忽略错误
    Dim pug as Ipug
    Set pug = obj '虚拟机调用 QueryInterface
```

<sup>4</sup> 最左边基类的 vtbl 中的入口项，在进入到方法实现之前，并不需要 adjustor thunk 来调整 this 指针。这仅限于“使用 adjustor thunk”并且“把最左边基类放在对象布局结构顶部”的编译器。Microsoft C++编译器就满足这样的条件。

```

If Not (pug is Nothing) Then
    pug.Snore
End if
Dim cat as ICat
Set cat = obj '虚拟机调用 QueryInterface
If Not (cat is Nothing) Then
    cat.IgnoreMaster
End if
End Sub

```

QueryInterface 调用失败时, Visual Basic 和 Java 虚拟机都会出现特殊的异常(exception)。在这两个环境中, 虚拟机自动地把“该语言中变量的生命周期范围的概念”映射到显式地调用 AddRef 和 Release, 使客户程序员不必关心这些细节。

有一项技术可使在 C++ 中使用 COM 接口指针的操作得以简化, 这就是用智能指针类(smart pointer class)把接口指针隐藏起来, 从而无须直接调用 IUnknown 方法。理想情况下, COM 智能接口指针将可以完成以下工作:

1. 在赋值过程中, 正确地处理 AddRef/Release 调用。
2. 在析构函数中自动释放接口, 从而降低资源泄漏的可能性, 以及改进异常的安全性。
3. 通过 C++ 类型系统来简化对 QueryInterface 的调用。
4. 可以透明地将遗留代码中的纯接口指针(raw interface pointer)替换掉, 而不会牺牲程序的正确性。

最后一条使得一个非常困难的问题得以圆满解决。Internet 上充满了一大堆 COM 智能指针, 虽然它们支持对纯接口指针的透明替换, 但同时也引入了潜在的错误(如同他们当初声称要解决的问题一样多)。实际上, Visual C++ 5.0 发布了三个这样的指针(一个针对 MFC、一个针对 ATL, 还有一个支持 Direct-to-COM), 它们很容易被正确使用, 同时也很容易被误用。《C++ Report》<sup>5</sup> 1995 年 9 月期和 1996 年 2 月期各有一篇文章讲述使用智能指针可能导致的危险。本书随带的源代码包含一个 COM 智能指针, 它也是我写作这两篇文章的副产品。该智能指针企图解决发生在纯指针和智能 COM 指针之间常见的错误。这个智能指针类 SmartInterface 有两个模板参数: 接口的 C++ 类型和指向相应 IID 的指针。所有对 IUnknown 方法的访问都被隐藏在“被重载的操作符”中:

```

#include "smartif.h"

void TryToSnoreAndIgnore(/* [in] */ IUnknown *pUnk) {
    // 复制构造函数调用 QueryInterface
    SmartInterface<IPug, &IID_IPug> pPug = pUnk;
    if (pPug) // 类型转换操作符什么也不返回
        pPug->Snore(); // 操作符-> 返回安全纯指针
}

```

<sup>5</sup> 这些《C++ Report》文章可以通过 <http://www.develop.com/dbox/cxx/InterfacePtr.htm> 和 <http://www.develop.com/dbox/cxx/SmartPtr.htm> 获取。

```
// 复制构造函数调用 QueryInterface
SmartInterface<ICat, &IID_ICat> pCat = pUnk;
if (pCat) // 类型转换操作符什么也不返回
    pCat->IgnoreMaster();
// 操作符-> 返回安全纯指针
// 超出作用域时析构函数释放指针
}
```

猛一看，智能指针非常吸引人，但是它们也是非常危险的，因为它们可使程序员陷入一种梦幻状态，在这种状态下看似没有什么与 COM 有关的事情需要操心。智能指针确实可以解决实际问题，使程序员不再面对异常情况，然而一旦使用不当，智能指针能够带来的错误与它们要提防的错误一样多。例如，许多智能指针允许通过智能指针的操作符“->”来访问接口的方法。不幸的是，这也使得客户通过箭头操作符就可以调用 Release，而没有通知到底层的智能指针，于是在析构函数中自动调用 Release 的动作就变成多余的了，所以通过箭头操作符调用 Release 是不允许的。

## 2.10 优化 QueryInterface

实际上，本章前面给出的 QueryInterface 实现非常直接，任何人只要对 COM 和 C++ 有基本的理解就可以维护 QueryInterface 的代码。然而，许多产品环境和框架都喜欢采用数据驱动（data-driven）的实现方式，以便获得更大的灵活性和更好的性能，因为这样做可以降低代码尺寸。这样的实现方式假定每个 COM 兼容的类都提供了一个表格，通过固定的偏移或者其他的技术，把该类所支持的 IID 映射到对象的某个地方。从本质上讲，本章前面给出的 QueryInterface 实现以编译之后的机器代码为基础，为依次出现的每个 if 建立了一个表格，并且使用 static\_cast 操作符来计算固定的偏移（static\_cast 只是简单地加上基类的偏移，以便找到类型兼容的 vptr）。

为了实现表格驱动（table-driven）的 QueryInterface，我们首先需要定义表格将包含什么样的信息。最起码每个表项要包含一个指向 IID 的指针，以及某些其他的信息以便 QueryInterface 能够找到对象中与被请求的接口相对应的 vptr。为了具有最大的灵活性，在每个表项中保存一个函数指针，这样做将可以支持新的“查找接口”技术，而不仅仅限于在转换到基类时采用的偏移计算法。本书随带的源代码包含一个头文件 inttable.h，其中定义的接口表项如下：

```
// inttable.h (本书专用的头文件) ///////
// 扩展性函数的 typedef
typedef HRESULT (*INTERFACE_FINDER)
```

```

(void *pThis, DWORD dwData, REFIID riid, void **ppv);

// 伪函数表明表项就是偏移
#define ENTRY_IS_OFFSET INTERFACE_FINDER(-1)

// 表的基本布局
typedef struct _INTERFACE_ENTRY
{
    const IID * pIID;           // 匹配 IID
    INTERFACE_FINDER pfnFinder; // 寻找函数
    DWORD dwData;              // 偏移值/其他数据
} INTERFACE_ENTRY;

```

头文件也包含下面的宏，以便在类定义的内部组成接口表格：

```

// inttable.h (本书专用的头文件) //////

#define BASE_OFFSET(ClassName, BaseName) \
(DWORD (static_cast<BaseName*> (reinterpret_cast<
<ClassName*> (0x1000000)) - 0x1000000)
#define BEGIN_INTERFACE_TABLE(ClassName) \
typedef ClassName_ITCls; \
const INTERFACE_ENTRY *GetInterfaceTable(void) { \
    static const INTERFACE_ENTRY table[] = { \
#define IMPLEMENTS_INTERFACE(Itf) \
{&IID_##Itf, ENTRY_IS_OFFSET, BASE_OFFSET(_ITCls, Itf)}, \
#define IMPLEMENTS_INTERFACE_AS(req, Itf) \
{&IID_##req, ENTRY_IS_OFFSET, BASE_OFFSET(_ITCls, Itf)}, \
#define END_INTERFACE_TABLE() \
{ 0, 0, 0 } }; return table; }

```

我们所需要的是一个函数过程，它能够解析一个接口表格以便响应客户的 `QueryInterface` 请求。文件 `inttable.cpp` 包含了这样一个函数：

```

// inttable.cpp (本书专用的源文件) //////

HRESULT InterfaceTableQueryInterface(void *pThis,
                                      const INTERFACE_ENTRY *pTable,
                                      REFIID riid, void **ppv) {
    if (InlineIsEqualGUID(riid, IID_IUnknown)) {
        // 第一项肯定是一个偏移值
        *ppv = (char*)pThis + pTable->dwData;
        ((IUnknown*)(*ppv))->AddRef(); // A2
        return S_OK;
    }
    else {

```

```

HRESULT hr = E_NOINTERFACE;
while (pTable->pfnFinder) { // null fn ptr == EOT
    if (!pTable->pIID || InlineIsEqualGUID(riid,
                                              *pTabl e->pIID)) {
        if (pTable->pfnFinder == ENTRY_IS_OFFSET) {
            *ppv = (char*)pThis + pTable->dwData;
            (IUnknown*) (*ppv)->AddRef(); // A2
            hr = S_OK;
            break;
        }
        else {
            hr = pTable->pfnFinder(pThis,
                                      pTable->dwData, riid, ppv);
            if (hr == S_OK) break;
        }
    }
    pTabl e++;
}
if (hr != S_OK) *ppv = 0;
return hr;
}
}

```

给定了被查询对象的一个接口指针之后，InterfaceTableQueryInterface 遍历整个表格，查找与指定 IID 匹配的表项，找到之后，或者加上相应的偏移，或者调用适当的函数。前面给出的代码使用了 IsEqualGUID 的一个优化版本，虽然产生的程序代码要大一些，但是与非表格驱动的实现做法相比，其速度可以提高约 20%~30%。由于目标代码中 InterfaceTableQueryInterface 只出现一次，所以这样做是很值得的。

对于任何 C++，要提供基于这种表格驱动的 COM 支持是很简单的，我们只需使用 C 预处理器即可。下面的代码摘自 impunk.h，它利用接口表格为对象定义了 QueryInterface、AddRef 和 Release，同时它假定对象被分配在堆中。代码如下：

```

// impunk.h (本书专用的头文件) //////////

// AUTO_LONG 是一个构造为 0 的长整型
struct AUTO_LONG {
    LONG value;
    AUTO_LONG(void) : value(0) {}
};

#define IMPLEMENT_UNKNOWN(ClassName) \
AUTO_LONG m_cRef; \
STDMETHODIMP QueryInterface (REFIID riid,void **ppv) { \
    return InterfaceTableQueryInterface (this, \
                                         GetInterfaceTable(), riid, ppv); \
} \
STDMETHODIMP_(ULONG) AddRef(void) { \
    return InterlockedIncrement(&m_cRef.value); \
} \

```

```

STDMETHODIMP_(ULONG) Release(void) {
    ULONG res = InterlockedDecrement(&m_cRef.value);
    if (res == 0) \
        delete this;
    return res;
}

```

实际的头文件也包含其他一些宏，以便支持不在堆中的对象（non-heap-based object）。

为了实现上一章中的 PugCat 例子，我们只需去掉原有的 QueryInterface、AddRef 和 Release 实现，然后插入适当的宏，代码如下：

```

class PugCat : public IPug, public ICat {
protected:
    virtual ~PugCat(void);
public:
    PugCat(void);
    // IUnknown 方法
    IMPLEMENTS_UNKNOWN(PugCat)
    BEGIN_INTERFACE_TABLE(PugCat)
        IMPLEMENTS_INTERFACE(IPug)
        IMPLEMENTS_INTERFACE(IDog)
        IMPLEMENTS_INTERFACE_AS(IAnimal, IDog)
        IMPLEMENTS_INTERFACE(ICat)
    END_INTERFACE_TABLE()
    // IAnimal 方法
    STDMETHODIMP Eat(void);
    // IDog 方法
    STDMETHODIMP Bark(void);
    // IPug 方法
    STDMETHODIMP Snore(void);
    // ICat 方法
    STDMETHODIMP IgnoreMaster(void);
};

```

在使用了这些预处理器宏之后，就不再需要其他的代码来支持 IUnknown 了。剩下的事情就是实现所有接口中其他实际的方法，这些方法正是这个类区别于其他类的独特之处。

## 2.11 数据类型

所有的 COM 接口都必须被定义在 IDL 中。IDL 允许我们可以用一种“语言独立”

和“平台独立”的方式来描述非常复杂的数据类型。表 2.1 为 IDL 所支持的基本类型，以及对应到 C、Java 和 Visual Basic 语言中的数据类型。整数和浮点类型不言自明，无须多说。在 COM 程序设计中首先值得“感兴趣”的数据类型是字符类型和字符串类型。

表 2.1 COM 基本类型

语 言	IDL	Microsoft C++	Visual Basic	Microsoft Java
基本类型	boolean	Unsigned char	不支持	char
	byte	Unsigned char	不支持	char
	small	char	不支持	char
	short	short	Integer	short
	long	long	Long	int
	hyper	_int64	不支持	long
	float	float	Single	float
	double	double	Double	double
	char	unsigned char	不支持	char
	wchar_t	wchar_t	Integer	char
扩展类型	enum	enum	Enum	int
	接口指针	接口指针	接口引用	接口引用
	VARIANT	VARIANT	Variant	ms.com.Variant
	BSTR	BSTR	String	java.lang.String
	VARIANT_BOOL	Short [-1/0]	Boolean [True/False]	boolean [true/false]

COM 中所有的字符都用 OLECHAR 数据类型来表示。在 Windows NT (译注 1)、Windows 95、Win32 和 Solaris 环境中，OLECHAR 只是简单地被定义 (typedef) 为 C 数据类型 wchar\_t。其他的平台请参照有关的文档。Win32 平台使用 wchar\_t 数据类型来表示 16 位 Unicode 字符。<sup>6</sup>因为指针类型在 IDL 中被假定为指向某一单个实例，而不是数组，所以 IDL 引入了 [string] 属性，表明这个指针指向一个以“null”作为结束符的字符串数组。

```
HRESULT Method([in, string] const OLECHAR *pwsz);
```

为了允许我们在代码中定义与 OLECHAR 兼容的字符串和字符，COM 提供了 OLESTR 宏，它把字母 L 加在字符串或者字符常量之前，告诉编译器该常量的类型为

译注 1：5.0 版本已被改名为 Windows 2000。

<sup>6</sup> OLECHAR 类型比 Win32 API 中习惯使用的 TCHAR 更为便利，因为它可以避免“为每个接口维护两个版本(CHAR 和 WCHAR)”。由于对象只需要支持一种字符类型，所以对象的开发人员不必考虑客户程序是否使用了 UNICODE 预处理器符号。

wchar\_t。例如，下面的代码可以正确地初始化一个 OLECHAR 指针：

```
const OLECHAR *pwsz = OLESTR("Hello");
```

在 Win32 或者 Solaris 下，上面的代码等价于：

```
const wchar_t *pwsz = L"Hello";
```

前者更为可取，因为在任何平台下它都可以通过编译。

因为我们往往有必要把基于 wchar\_t 的字符串拷贝到普通的、基于 char 的缓冲区中，所以 C 运行时库提供了两个转换函数：

```
size_t mbstowcs(wchar_t *pwsz, const char *psz, int cch);
size_t wcstombs(char *psz, const wchar_t *pwsz, int cch);
```

这两个函数的工作方式类似于 C 的运行时函数 strcpy，只不过这两个函数在拷贝过程中会扩大或者缩减字符串。下面的代码说明了如何把基于 OLECHAR 的方法参数拷贝到基于 char 的数据成员中：

```
class BigDog : public ILabrador {
    char m_szName[1024];
public:
    STDMETHODIMP SetName(/* [in, string] */ const OLECHAR *pwsz) {
        HRESULT hr = S_OK;
        size_t cb = wcstombs(m_szName, pwsz, 1024);
        // 检测缓冲是否溢出，或是否有不良转换
        if (cb == sizeof(m_szName) || cb == (size_t)-1) {
            m_szName[0] = 0;
            hr = E_INVALIDARG;
        }
        return hr;
    }
};
```

上面的代码非常简单，但是程序员必须要清楚地知道他所使用的这两种字符类型的区别。比较复杂的（也是很常见的）情形是在 OLECHAR 和 Win32 TCHAR 数据类型之间的转换，由于 TCHAR 在不同的情况下会被编译为 char 或者 wchar\_t，所以方法的实现必须要适当地处理这两种情形：

```
class BigDog : public ILabrador {
    TCHAR m_szName[1024]; // 注意：是 TCHAR 字符串
public:
    STDMETHODIMP SetName(/* [in, string] */ const OLECHAR *pwsz) {
        HRESULT hr = S_OK;
#ifdef UNICODE
```

```

// Unicode 已内置(TCHAR == wchar_t)
    wcsncpy(m_szName, pwsz, 1024);
// 检测缓冲是否溢出
    if (m_szName[1023] != 0) {
        m_szName[0] = 0;
        hr = E_INVALIDARG;
    }
#else
// Unicode 未内置(TCHAR == char)
    size_t cb = wcstombs(m_szName, pwsz, 1024);
// 检测缓冲是否溢出, 或是否有不良转换
    if (cb := sizeof(m_szName) || cb == (size_t)-1) {
        m_szName[0] = 0;
        hr = E_INVALIDARG;
    }
#endif
    return hr;
}
};


```

很显然, 处理 OLECHAR 到 TCHAR 之间的转换要更加复杂一些。不幸的是, 这恰好是 Win32 平台上 COM 编程过程中最常见的情形。

简化文本转换的一种办法是利用 C++ 类型系统, 使用函数重载的方法, 并且根据参数的类型来选择正确的字符函数。本书随带的源代码 `cstring.h` 头文件包含一组字符串库函数, 它们与 C 运行库中 (`string.h`) 的标准函数类似。例如, `strncpy` 有四个对应的版本, 其参数类型涵盖了两种可能的字符类型 (`wchar_t` 或者 `char`) 的组合, 代码如下:

```

// 摘自 cstring.h (本书专用头文件) //////
inline bool ustrncpy(char *p1, const wchar_t *p2, size_t c){
    size_t cb = wcstombs(p1, p2, c);
    return cb != c && cb != (size_t)-1;
}
inline bool ustrncpy(wchar_t *p1, const wchar_t *p2, size_t c){
    wcsncpy(p1, p2, c);
    return p1[c - 1] == 0;
}
inline bool ustrncpy(char *p1, const char *p2, size_t c) {
    strncpy(p1, p2, c);
    return p1[c - 1] == 0;
}
inline bool ustrncpy(wchar_t *p1, const char *p2, size_t c) {
    size_t cch = mbstowcs(p1, p2, c);
    return cch != c && cch != (size_t)-1;
}

```

注意, 对于任意一种字符类型组合, 都可以找到对应的 `ustrncpy` 重载版本, 其结果指示完整的字符串是否被拷贝或者被转换。因为这些函数都被声明为内联 (`inline`) 函数。

数，所以使用这些函数不会导致性能损失。有了这些函数之后，前面的代码就可以被简化，不再需要条件编译指令，如下：

```
class BigDog : public ILabrador {
    TCHAR m_szName[1024]; // 注意：是 TCHAR 字符串
public:
    STDMETHODIMP SetName(/* [in, string] */ const OLECHAR *pwsz) {
        HRESULT hr = S_OK;
        // 使用本书专用的 ustrncpy 重载版本拷贝或转换
        if (!ustrncpy(m_szName, pwsz, 1024)) {
            m_szName[0] = 0,
            hr = E_INVALIDARG;
        }
        return hr;
    }
};
```

头文件 `cstring.h` 中也包括 `strlen`、`strcpy` 和 `strcat` 函数的重载版本。

正如前面所示，把字符串从一个缓冲区拷贝到另一个缓冲区，这种基于重载库函数的方法可以获得最佳的性能，也可以导致最小的代码，以及程序员最轻的负担。然而，当同时使用 COM 和 Win32 API 函数时，由于它们不能采用这种技术，因而会引起一种很普遍的现象。考虑下面的代码片断，从编辑框控制（edit control）中读入一个字符串，并把字符串转换为 IID，代码如下：

```
HRESULT IIDFromHWND(HWND hwnd, IID& riid) {
    TCHAR szEditText [1024];
    // 调用一个基于 TCHAR 的 Win32 例程
    GetWindowText(hwnd, szEditText, 1024);
    // 调用一个基于 OLECHAR 的 COM 例程
    return IIDFromString(szEditText, &riid);
}
```

假定在编译代码时已经定义了 C 预编译器符号 `UNICODE`，那么这段代码就会工作得很好，因为 `TCHAR` 和 `OLECHAR` 都只不过是 `wchar_t` 的别称，所以实际上并没有转换发生。然而，如果这段代码是在 Win32 API 的非 `UNICODE` 版本下被编译的，那么 `TCHAR` 是 `char` 的别名，所以 `IIDFromString` 的第一个参数是不正确的类型。解决这个问题的一个办法是依赖条件编译指令：

```
HRESULT IIDFromHWND(HWND hwnd, IID& riid) {
    TCHAR szEditText [1024];
    GetWindowText(hwnd, szEditText, 1024);
    #ifdef UNICODE
    return IIDFromString(szEditText, &riid);
    #else
    OLECHAR wszEditText [1024];
```

```

    ustrncpy(wszEditText, szEditText, 1024);
    return IIDFromString(wszEditText, &riid);
#endif
}

```

虽然这段代码可以产生最优的代码，但是若每次有错误的字符类型参数存在时，我们都使用这项技术，那将是非常繁琐和沉闷的。处理这种问题的另一项技术是使用一个“垫片类（shim class）”，它的构造函数可以接受任一种字符类型作为参数。这个垫片类也需要类型转换操作符，以便它可以用在“应该出现 `const char *` 或者 `const wchar_t *`”的场合。在这些类型转换操作符内部，垫片类或者分配一个重复的缓冲区，并执行必要的转换操作，或者直接返回原来的字符串而不做任何转换。然后垫片类的析构函数会释放所有这些缓冲区。`cstring.h` 头文件包含两个垫片类：`_U` 和 `_UNCC`。前者针对通常的用法；后者针对非常量正确（non-const-correct）的函数和方法<sup>7</sup>（比如 `IIDFromString`）。有了这两个垫片类之后，前面的代码就可以考虑进一步简化：

```

HRESULT IIDFromHWND(HWND hwnd, IID& riid) {
    TCHAR szEditText [1024];
    GetWindowText(hwnd, szEditText, 1024);
    // 如果必要使用_UNCC 垫片类转换
    return IIDFromString(_UNCC(szEditText), &riid);
}

```

注意，这里并不需要条件编译指令。如果这段代码是在 Win32 的 Unicode 版本下被编译的，那么 `_UNCC` 类将通过它的类型转换操作符，简单地把原来的缓冲区传递出来。如果这段代码是在 Win32 的非 Unicode 版本下被编译的，那么 `_UNCC` 类将分配一个缓冲区，然后把字符串转换成 Unicode 码。当这条语句被完全执行完毕之后，`_UNCC` 的析构函数将会释放该缓冲区。<sup>8</sup>

另外一个必须要讨论的、与文本有关的数据类型是 `BSTR`。凡是要在 Visual Basic 或者 Java 中用到的接口，都必须使用 `BSTR` 字符串类型。`BSTR` 包含长度前缀（length-prefixed），是以 `null` 作为结束符的 `OLECHAR` 字符串。长度前缀指示该字符串所包含的字节数（不包括结束符 `null`），它作为一个 4 字节整数被保存在字符串的首字符之前的 4 个字节中。图 2.6 显示了字符串“Hi”作为 `BSTR` 的情形。为了使 `BSTR` 能够自由地在“那些无须考虑内存分配问题的方法”中被返回给调用方，所有的 `BSTR` 都必须由“COM 管理的内存分配器”来分配。COM 提供了几个 API 函数用来管理 `BSTR`：

---

<sup>7</sup> `_UNCC` 只是 `_U` 的一个版本而已，它也提供 `wchar_t *` 和 `char *` 的类型转换操作符。虽然这个扩展版本可以用在任何地方，但是我只选择在处理非常量正确接口时才使用它，以此强调类型系统的一些折衷。而且，大部分的 COM API 并不是常量正确（const-correct）的，所以 `_UNCC` 垫片类非常有用。

<sup>8</sup> 虽然我发现 `cstring.h` 中的字符串工具类足够用来管理基于 COM 的文本处理工作，但是 ATL 和 MFC 库使用了不同的做法，它们以 `alloca` 和宏为基础。请参考有关这些技术的文档以获得更多的信息。

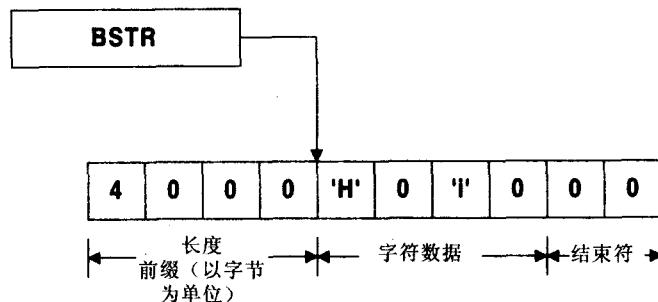


图 2.6 “Hi” 作为 BSTR 的情形

```
// 摘自 oleauto.h
// 分配和初始化一个 BSTR
BSTR SysAllocString(const OLECHAR *psz);
BSTR SysAllocStringLen(const OLECHAR *psz, UINT cch);

// 重分配和初始化一个 BSTR
INT SysReAllocString(BSTR *pbstr, const OLECHAR *psz);
INT SysReAllocStringLen(BSTR *pbstr, const OLECHAR *psz,
                        UINT cch);

// 释放一个 BSTR
void SysFreeString(BSTR bstr);

// 将“长度前缀”转换为字符或 byte
UINT SysStringLen(BSTR bstr);
UINT SysStringByteLen(BSTR bstr);
```

当把字符串作为 [in] 参数传给一个方法时，调用方有责任在调用方法之前先调用 `SysAllocString`，然后在方法完成（返回）之后调用 `SysFreeString`。考虑下面的方法定义：

```
HRESULT SetString([in] BSTR bstr);
```

假定调用方已经有了一个 OLECHAR 兼容的字符串，那么在调用方法之前，有必要先把字符串转换为 BSTR，如下代码所示：

```
// 将纯 OLECHAR 字符串转换为 BSTR
BSTR bstr = SysAllocString(OLESTR("Hello"));
// 调用方法
HRESULT hr = p->SetString(bstr);
// 释放 BSTR
SysFreeString(bstr);
```

`cstring.h` 头文件中也包含一个专门处理 BSTR 的垫片类 `_UBSTR`，定义如下：

```
// 摘自 ustring.h (本书专用的头文件) /////
class _UBSTR {
    BSTR m_bstr;
public:
    _UBSTR(const char *psz)
        : m_bstr(SysAllocStringLen(0, strlen(psz))) {
        mbstowcs(m_bstr, psz, INT_MAX);
    }

    _UBSTR(const wchar_t *pwsz)
        : m_bstr(SysAllocString(pwsz)) {

        operator BSTR (void) const { return m_bstr; }

        ~_UBSTR(void) { SysFreeString(m_bstr); }
    };
};
```

有了这个垫片类之后，前面的代码片断就可以被简化为：

```
// 调用方法
HRESULT hr = p->SetString(_UBSTR(OLESTR("Hello")));
```

注意，基于 `char` 或者 `wchar_t` 的字符串也可以与 `_UBSTR` 垫片类一起使用。

当把字符串作为 `[out]` 参数从方法中传出来时，对象有责任为结果字符串调用 `SysAllocString` 分配一个缓冲区。然后，释放缓冲区则是调用方的责任，它在用完字符串之后可以调用 `SysFreeString` 以释放缓冲区。考虑下面的方法定义：

```
HRESULT GetString([out, retval] BSTR *pbstr);
```

方法实现将需要新建一个 `BSTR`，并返回给调用方，代码如下：

```
STDMETHODIMP MyClass::GetString(BSTR *pbstr) {
    *pbstr = SysAllocString(OLESTR("Goodbye!"));
    return S_OK;
}
```

然后，当调用方把 `BSTR` 字符串拷贝到应用自身管理的字符串缓冲区之后，它应该释放该字符串，如下：

```
extern OLECHAR g_wsz[]; BSTR bstr = 0;
HRESULT hr = p->GetString(&bstr);
if (SUCCEEDED(hr)) {
    wcscpy(g_wsz, bstr);
    SysFreeString(bstr);
}
```

`BSTR` 有一个很重要的方面我还没有提到。把 `null` 指针作为 `BSTR` 来传递也是合法

的，它可以表示一个空字符串。这意味着前面的代码片断不是非常合适的。对 `wcscpy` 的调用：

```
wcscpy(g_wsz, bstr);
```

需要进行保护处理，以防止可能出现 null 指针：

```
wcscpy(g_wsz, bstr ? bstr : OLESTR(""));
```

为了简化对 BSTR 的使用，`wstring.h` 头文件包含一个简单的内联函数，如下：

```
inline OLECHAR *SAFEBSTR(BSTR b){return b ? b : OLESTR("");}
```

从内存使用角度来讲，允许 null 指针作为 BSTR 来使用，确实使数据类型更为有效，但是却因为以上这些简单的测试而使代码比较零乱。

表 2.1 所列的基本数据类型可以利用 C 风格的结构组合起来。IDL 在标记名字空间（tag namespace）中遵循 C 语言的规则，这意味着大多数的 IDL 接口定义可以使用 `typedef` 语句：

```
typedef struct tagCOLOR {
    double red;
    double green;
    double blue;
} COLOR;
HRESULT SetColor([in] const COLOR *pColor);
```

或者必须直接使用 `struct` 关键字来修饰类型名：

```
struct COLOR {
    double red;
    double green;
    double blue;
};
HRESULT SetColor([in] const struct COLOR *pColor);
```

前者是大家比较喜欢的格式。像这样比较简单的结构既可以在 Visual Basic 中使用，也可以在 Java 中使用；然而，当本书写作时，Visual Basic 当前的版本可以访问使用结构的接口，但是不能实现“以结构作为方法参数”的接口。

IDL 和 COM 也支持联合（union）。为了保证对联合的解释不存在二义性，IDL 期望有一个鉴别器（discriminator）能与联合（union）一起提供出来，以便表明哪个联合成员是真正被使用的。鉴别器必须是整数类型，而且必须出现在与联合相同的逻辑层次中。如果联合被声明在结构体的外面，那么可以考虑使用未经封装的联合：

```
union NUMBER {
```

```
[case(1)] long i;
[case(2)] float f;
};
```

[case]属性用于匹配鉴别器与实际使用的联合成员。为了把鉴别器与未经封装的联合类型联系起来，我们可以使用[switch\_is]属性：

```
HRESULT Add([in, switch_is(t)] union NUMBER *pn,
            [in] short t);
```

当联合与其鉴别器一起被包装在一个结构中时，此聚合类型被称为“经过封装的（encapsulated）”或者“经过鉴别的（discriminated）”联合：

```
struct UNUMBER {
    short t;
    [switch_is(t)] union VALUE {
        [case(1)] long i;
        [case(2)] float f;
    };
};
```

COM 预先定义了一个通用的、经过鉴别的联合供 Visual Basic 使用。这个联合被称为 VARIANT，<sup>9</sup> 它可以容纳“IDL 所支持的基本类型的一个子集”中类型的实例或者引用。每个被支持的类型都有一个对应的鉴别器值：

VT_EMPTY	nothing
VT_NULL	SQL 网格的 Null
VT_I2	short
VT_I4	long
VT_R4	float
VT_R8	double
VT_CY	CY (64-位 currency 类型)
VT_DATE	DATE (double)
VT_BSTR	BSTR
VT_DISPATCH	IDispatch *
VT_ERROR	HRESULT
VT_BOOL	VARIANT_BOOL (True=-1, False=0)
VT_VARIANT	VARIANT *
VT_UNKNOWN	IUnknown *
VT_DECIMAL	16 位定点类型
VT_UI1	opaque byte

<sup>9</sup> 也被称为 VARIANTARG。术语 VARIANTARG 表示所有合法参数类型的变体类型；而术语 VARIANT 是指合法的方法结果的变体类型。其实，数据类型 VARIANTARG 只是 VARIANT 的一个别称，两者可以互换使用。

下面两个标志可以与以上这些标记组合起来，说明该变体（variant）所包含的是指定类型的引用还是数组。

VT\_ARRAY 包含一个 SAFEARRAY  
 VT\_BYREF 表示变体是一个引用

COM 提供了几个 API 函数用于管理 VARIANT：

```
// 将变体初始化为空
void VariantInit (VARIANTARG * pvarg);

// 释放变体中的资源
HRESULT VariantClear(VARIANTARG * pvarg);

// 将深拷贝变体
HRESULT VariantCopy(VARIANTARG * plhs, VARIANTARG * prhs);

// 间接引用并深拷贝变体
HRESULT VariantCopyInd(VARIANT * plhs, VARIANTARG * prhs);

// 将变体转换为指定类型
HRESULT VariantChangeType(VARIANTARG * plhs,
                           VARIANTARG * prhs, USHORT wFlags, VARTYPE vtlhs);

// 将变体转换为指定类型(explicit locale)
HRESULT VariantChangeTypeEx(VARIANTARG * plhs,
                            VARIANTARG * prhs, LCID lcid, USHORT wFlags,
                            VARTYPE vtlhs);
```

这些函数极大地简化了对 VARIANT 变量的操作。

为了理解这些 API 函数是如何被使用的，请考虑以下的方法，它接受一个[in]参数，其类型为 VARIANT：

```
HRESULT UseIt([in] VARIANT var);
```

下面的代码片断演示了如何向这个方法传递一个整数：

```
VARIANT var;
VariantInit(&var); // 初始化 VARIANT
V_VT(&var) = VT_I4; // 设置鉴别器
V_I4(&var) = 100; // 设置联合
HRESULT hr = pltf->Uselt(var); // 使用 VARIANT
VariantClear(&var); // 释放 VARIANT 中的资源
```

注意，该代码片断使用了标准访问器宏（standard accessor macro）（译注 2）来访问 VARIANT 的数据成员。其中这两行代码：

```
V_VT(&var) = VT_I4;
V_I4(&var) = 100;
```

---

译注 2：即上述代码的 V\_xx。

等价于下面直接访问数据成员的代码：

```
var.vt = VT_I4;
var.lVal = 100;
```

不过前者比较受欢迎，因为它也可以在不支持匿名联合(anonymous union)的 C 编译器下通过编译。

仍然是前面给出的方法 UseIt，下面的方法实现使用 VARIANT 参数作为字符串：

```
STDMETHODIMP MyClass::UseIt (/* [in] */ VARIANT var) {
    // 声明并初始化另一个 VARIANT
    VARIANT var2; VariantInit(&var2);
    // 将 var 转换为 BSTR 并存在 var2 中
    HRESULT hr = VariantChangeType(&var2, &var, 0, VT_BSTR);
    // 使用字符串
    if (SUCCEEDED(hr)) {
        ustrcpy(m_szSomeDataMember, SAFEBCSTR (V_BSTR (&var2)));
    }
    // 释放 var2 所拥有的资源
    VariantClear(&var2);
}
return hr;
}
```

注意，VariantChangeType API 函数处理“客户传递过来的任何一种类型的 VARIANT”和“对象所期望的类型”(本例子中为 BSTR)之间可能存在的复杂转换。

最后一种值得讨论的数据类型是 COM 接口。COM 接口也可以作为方法参数进行传递，可以有两种方式。如果在设计时接口指针的类型是已知的话，那么我们可以以静态的方式来声明接口类型，例如：

```
HRESULT GetObject([out] IDog **ppDog);
```

如果在设计时接口指针的类型是未知的，那么接口设计者可以给用户一个机会，让他在运行时指定接口的类型。为了支持动态类型的接口，IDL 支持[iid\_is]属性：

```
HRESULT GetObject([in] REFIID riid,
                  [out, iid_is(riid)] IUnknown **ppUnk);
```

虽然这种形式的描述能够很好地工作，但是下面的代码形式更受欢迎，因为它与 QueryInterface 非常类似：

```
HRESULT GetObject([in] REFIID riid,
                  [out, iid_is(riid)] void **ppv);
```

[iid\_is]属性可以与 IUnknown \* 或者 void \* 类型的[in]参数或者[out]参数一起使用。为了使用动态类型的接口参数，我们只需简单地指定“所希望的指针类型”的 IID 即可：

```
IDog *pDog = 0;
HRESULT hr = pItf->GetObject(IID_IDog, (void**)&pDog);
```

对应的实现也只要简单地使用目标对象的 `QueryInterface` 方法，来初始化该参数：

```
STDMETHODIMP Class::GetObject(REFIID riid, void **ppv) {
    extern IUnknown * g_pUnkTheDog;
    return g_pUnkTheDog->QueryInterface(riid, ppv);
}
```

动态类型的接口指针应该总是优先于“以 `IUnknown` 为类型”的静态类型接口指针，从而避免额外的“客户与对象之间的方法调用”。

## 2.12 IDL 属性和 COM 属性

指明一个对象具有某些公开可见的属性，并且可通过 COM 接口来访问或者修改这些属性，这种做法在有些情况下是非常有用的。COM IDL 允许接口的方法可以被加上一些注释，以便反映出该方法可以被用来修改或者读取对象中一个已经被命名的属性。考虑下面的接口定义：

```
[ object, uuid(0BB3DAE1-11F4-11d1-8C84-0080C73925BA) ]
interface ICollie : IDog {
    // Age 是只读属性
    [propget] HRESULT Age([out, retval] long *pVal);
    // HairCount 是读/写属性
    [propget] HRESULT HairCount([out, retval] long *pVal);
    [propput] HRESULT HairCount([in] long val);
    // CurrentThought 是只写属性
    [propput] HRESULT CurrentThought([in] BSTR val);
}
```

接口定义中使用了 `[propput]` 和 `[propget]` 属性，这相当于告诉编译器，它们所对应的方法应该被映射到那些“显式支持属性的语言”中的属性访问器（property accessor）。在 Visual Basic 中，这意味着成员 `Age`、`HairCount` 和 `CurrentThought` 可以使用类似“访问结构成员”同样的语法来操纵：

```
Sub UseCollie(fido as ICollie)
    fido.HairCount = fido.HairCount - (fido.Age * 1000)
    fido.CurrentThought = "I wish I had a bone"
End Sub
```

C++ 对该接口的映射是简单地在方法名前面加上 `put_` 或 `get_`，使程序员知道他正在访问一个属性：

```
void UseCollie(ICollie *pFido) {
    long n1, n2;
    HRESULT hr = pFido->get_HairCount(&n1)
    assert(SUCCEEDED(hr));
    hr = pFido->get_Age(&n2);
    assert(SUCCEEDED(hr));
    hr = pFido->put_HairCount(n1 - (n2 * 1000));
    assert(SUCCEEDED(hr));
    BSTR bstr = SysAllocString(OLESTR("I wish I had a bone"));
    hr = pFido->put_CurrentThought(bstr);
    assert(SUCCEEDED(hr));
    SysFreeString(bstr);
}
```

虽然属性并没有显式地带来任何开发技术，但是它们对于那些支持它们的编程语言来说，提供了一种比较自然的映射关系。<sup>10</sup>

## 2.13 异常

COM 对于“在方法实现中抛出异常”有特殊支持。因为 C++ 语言的异常并没有二进制标准，所以 COM 提供了显式的 API 函数，来抛出或者捕捉 COM 异常对象：

```
// 抛出异常
HRESULT SetErrorInfo([in] ULONG reserved, // m.b.z.
                     [in] IErrorInfo *pei);

// 捕捉异常
HRESULT GetErrorInfo([in] ULONG reserved, // m.b.z.
                     [out] IErrorInfo **ppei);
```

`SetErrorInfo` 函数在方法实现的内部被调用，以便把一个异常对象与当前的逻辑线程关联起来。<sup>11</sup> `GetErrorInfo` 从当前逻辑线程中获取异常对象，并清除异常，从而后续的 `GetErrorInfo` 调用将返回 `S_FALSE`，说明不再有未决的异常。正如这些函数所暗示的，COM 异常对象必须至少支持 `IErrorInfo` 接口：

<sup>10</sup> Microsoft 的 Direct-to-COM 支持功能允许客户可以把属性当作接口的公开数据成员(通过某种非常曲折古怪的机制)。

<sup>11</sup> COM 规范使用术语逻辑线程(logical thread)，来表示一个方法调用链，它有可能跨越多个实际的操作系统线程。

```
[ object, uuid(1CF2B120-547D-101B-8E65-08002B2BD119) ]
interface IErrorInfo: IUnknown {
    // 获得抛出异常的接口的 IID
    HRESULT GetGUID([out] GUID * pGUID);
    // 获得抛出异常的对象的类名
    HRESULT GetSource([out] BSTR * pBstrSource);
    // 获得异常的可读描述
    HRESULT GetDescription ([out] BSTR * pBstrDescription);
    // 获得错误文档的 WinHelp 文件名
    HRESULT GetHelpFile([out] BSTR * pBstrHelpFile);
    // 获得错误文档的 WinHelp 上下文 ID
    HRESULT GetHelpContext([out] DWORD * pdwHelpContext);
}
```

自定义的异常对象可以在实现 `IErrorInfo` 接口的基础上，有选择地实现其他的、与异常有关的接口。

COM 提供了 `IErrorInfo` 接口的一个缺省实现，我们可以使用 COM API 函数 `CreateErrorInfo` 创建它的对象实例：

```
HRESULT CreateErrorInfo([out] ICreateErrorInfo **ppcei);
```

除了 `IErrorInfo` 之外，缺省异常对象还暴露了 `ICreateErrorInfo` 接口，允许用户对新异常的状态进行初始化。下面是 `ICreateErrorInfo` 接口的定义：

```
[ object, uuid(22F03340-547D-101B-8E65-08002B2BD119) ]
interface ICreateErrorInfo: IUnknown {
    // 设置抛出异常的接口的 IID
    HRESULT SetGUID([in] REFGUID rGUID);
    // 设置抛出异常的对象的类名
    HRESULT SetSource([in, string] OLECHAR* pwszSource);
    // 设置异常的可读描述
    HRESULT SetDescription([in, string] OLECHAR* pwszDesc);
    // 设置错误文档的 WinHelp 文件名
    HRESULT SetHelpFile([in, string] OLECHAR* pwszHelpFile);
    // 设置错误文档的 WinHelp 上下文 ID
    HRESULT SetHelpContext([in] DWORD dwHelpContext);
}
```

注意，这个接口只是简单地允许用户设置异常对象的 5 个基本属性（通过 `IErrorInfo` 接口可以访问这些属性）。

下面的方法实现使用缺省的异常对象，向它的调用者抛出一个 COM 异常：

```
STDMETHODIMP PugCat::Snore(void) {
    if (this->IsAsleep())           //可以操作吗？
        return this->DoSnore();    //操作并返回
    //否则产生一个异常对象
    ICreateErrorInfo *pcei = 0;
```

```

HRESULT hr = CreateErrorInfo(&pcei);
assert (SUCCEEDED(hr));
// 初始化异常对象
hr = pcei->SetGUID(IID_IPug);
assert (SUCCEEDED(hr));
hr = pcei->SetSource(OLESTR("PugCat")),
assert (SUCCEEDED(hr));
hr = pcei->SetDescription(OLESTR("I am not asleep!"));
assert (SUCCEEDED(hr));
hr = pcei->SetHelpFile(OLESTR("C:\PugCat.hlp"));
assert (SUCCEEDED(hr));
hr = pcei->SetHelpContext(5221);
assert (SUCCEEDED(hr));
// 抛出异常
IErrorInfo *pei = 0;
hr = pcei->QueryInterface(IID_IErrorInfo, (void**)&pei);
assert (SUCCEEDED(hr));
hr = SetErrorInfo(0, pei);
// 释放资源并返回一个SEVERITY_ERROR
pei->Release();
pcei->Release();
return PUG_E_PUGNOTASLEEP;
}

```

注意，一旦异常对象已经被传递给 SetErrorInfo 函数了，那么 COM 将拥有该异常对象的一个引用，一直到“调用者使用 GetErrorInfo 捕捉到该异常”为止。

抛出异常的对象必须实现 ISupportErrorInfo 接口，以便指定哪个接口支持异常。客户使用这个接口来确定 GetErrorInfo 的结果是否可靠。<sup>12</sup> 这个接口非常简单：

```

[ object, uuid(DF0B3D60-548F-101B-8E65-08002B2BD119) ]
interface ISupportErrorInfo: IUnknown {
    HRESULT InterfaceSupportsErrorInfo([in] REFIID riid);
}

```

假定上一章的 PugCat 类从每个它所支持的接口都会抛出异常，那么 ISupportErrorInfo 接口的实现应该如下：

```

STDMETHODIMP PugCat:: InterfaceSupportsErrorInfo(REFIID riid)
{
    if (riid == IID_IAnimal || riid == IID_ICat ||
        riid == IID_IDog || riid == IID_IPug)
        return S_OK;
    else
        return S_FALSE;
}

```

<sup>12</sup> 实现 ISupportErrorInfo 接口的对象等于是告知外界，它显式支持 COM 异常，下级对象偶尔抛出的异常不会影响到它。

下面是一个客户的例子，它使用 `ISupportErrorInfo` 和 `GetErrorInfo` 来安全地处理异常：

```

void Tel_l_PugToSnore(/* [in] */ IPug *pPug) {
    // 调用方法
    HRESULT hr = pPug->Snore();
    if (FAILED(hr)) {
        // 检测对象是否支持 COM 异常
        ISupportErrorInfo *psei = 0;
        HRESULT hr2 = pPug->QueryInterface( IID_ISupportErrorInfo,
                                              (void**)&psei);
        if (SUCCEEDED(hr2)) {
            // 检测对象是否通过 IPug 方法支持 COM 异常
            hr2 = psei->InterfaceSupportsErrorInfo(IID_IPug);
            if (hr2 == S_OK) {
                // 读取逻辑线程的异常对象
                IErrorInfo *pei = 0;
                hr2 = GetErrorInfo(0, &pei);
                if (hr2 == S_OK) {
                    // 收集 source 和 description 字符串
                    BSTR bstrSource = 0, bstrDesc = 0;
                    hr2 = pei->GetDescription(&bstrDesc);
                    assert (SUCCEEDED(hr2));
                    hr2 = pei->GetSource(&bstrSource);
                    assert (SUCCEEDED(hr2));
                    // 显示错误信息
                    MessageBoxW(0, bstrDesc ? bstrDesc : L"",
                               bstrSource ? bstrSource : L"", MB_OK);
                }
                // 释放资源
                SysFreeString(bstrSource);
                SysFreeString(bstrDesc);
                pei->Release();
            }
            psei->Release();
        }
    }
    if (hr2 != S_OK) // 异常中有错误发生
        MessageBoxA(0, "Snore Failed", "IPug", MB_OK);
}

```

要把 COM 异常映射到 C++ 异常也是非常简单的。假设下面的类把 COM 异常和 `HRESULT` 结果绑定到同一个 C++ 类中：

```

struct COMException {
    HRESULT m_hresult; // 要返回的 HRESULT
    IErrorInfo *m_pei; // 要抛出的异常
    COMException(HRESULT hresult, REFIID riid,

```

```

        const OLECHAR *pszSource,
        const OLECHAR *pszDesc,
        const OLECHAR *pszHelpFile = 0,
        DWORD dwHelpContext = 0)  {
// 创建并初始化一个错误信息对象
    ICreateErrorInfo *pcei = 0;
    HRESULT hr = CreateErrorInfo(&pcei);
    assert (SUCCEEDED(hr));
    hr = pcei->SetGUID(riid);
    assert (SUCCEEDED(hr));
    if (pszSource)
        hr=pcei->SetSource (const_cast<OLECHAR*> (pszSource)),
    assert (SUCCEEDED(hr));
    if (pszDesc)
        hr=pcei->SetDescription (const_cast<OLECHAR*> (pszDesc));
    assert (SUCCEEDED(hr));
    if(pszHelpFile)
        hr=pcei->SetHelpFile (const_cast<OLECHAR*> (pszHelpFile)),
    assert (SUCCEEDED(hr));
    hr = pcei->SetHelpContext (dwHelpContext);
    assert (SUCCEEDED(hr));
// 将HRESULT 和 IerrorInfo 指针作为数据成员
    m_hresult = hr;
    hr=pcei->QueryInterface(IID_IErrorInfo, (void**)&m_pei);
    assert (SUCCEEDED(hr));
    pcei->Release ();
}
};


```

给定了上面所示的 COMException C++类之后，以前给出的 Snore 方法可以作进一步修改，把任意的 C++ 异常映射为 COM 异常，代码如下：

```

STDMETHODIMP PugCat:: Snore(void)
{
    HRESULT hrex = S_OK;
    try {
        if (this->IsAsleep())
            return this->DoSnore();
        else
            throw COMException(PUG_E_PUGNOTASLEEP, IID_IPug,
                               OLESTR("PugCat"), OLESTR("I am not asleep!"));
    }
    catch (COMException& ce) {
// 抛出一个 C++ COMException
        HRESULT hr = SetErrorInfo(0, ce.m_pei);
        assert (SUCCEEDED(hr));
        ce.m_pei->Release ();
        hrex = ce.m_hresult;
    }
    catch (...) {

```

```

// 抛出了未辨识的 C++ 异常
COMException ex(E_FAIL, IID_IPug, OLESTR("PugCat"),
                 OLESTR("A C++ exception was thrown"));
HRESULT hr = SetErrorInfo(0, ex.m_pei);
assert(SUCCEEDED(hr));
ex.m_pei->Release();
hrex = ex.m_hresult;
}
return hrex;
}

```

注意，此方法实现非常小心地禁止“纯 C++ 异常”被传播到方法边界之外，这是 COM 的一项绝对要求。

## 2.14 我们走到哪儿了？

这一章展示了 COM 接口的概念。COM 接口具有非常简单的二进制结构特征，通过此二进制结构可允许任何客户访问一个对象，而无须考虑客户或者对象实现使用什么样的编程语言。为了提供多种语言支持，COM 接口用 IDL 语言来定义。这些 IDL 接口定义也可以用来产生通信代码，允许对象（通过网络）远程访问。

本章大部分内容都在讨论 IUnknown，COM 的一切特性都建立在 IUnknown 的基础上。所有的 COM 接口都必须从 IUnknown 派生，因此，所有的 COM 对象都必须实现 IUnknown。IUnknown 提供了三个方法，客户通过这些方法，可以安全地查询对象的类型层次结构，从而可以访问对象引出的所有扩展功能。从这个意义上讲，QueryInterface 可以被认为是 COM 的类型转换操作符。按照同样的逻辑，IUnknown 也可以被看作是接口指针的“void \*”类型，因为直到 IUnknown 被通过 QueryInterface “转换”到某个更有意义的接口指针时，它才会真正有用。

值得一提的是，当访问或者实现 IUnknown 时，并没有重要的系统调用发生。从这个意义上讲，IUnknown 只不过是一个简单的协议，或者说是一组“所有程序必须要遵循的”承诺。这使得 COM 对象非常轻量，非常高效。在 C++ 中实现 IUnknown 只要求少量的样板代码。为了使“在 C++ 中实现 IUnknown”这项工作能够自动化，本章展示了一系列预处理宏，它们以表格驱动的方式来实现 QueryInterface。虽然这些宏并不是必需的，但是它们可以从每个类定义中消除大多数的公共样板代码，并且也不会使代码实现增加不适当的复杂性。

# 第3章

## 类

```
int cGorillas = Gorilla::GetCount();
IApe *pApe = new Gorilla();
pApe->GetYourStrokingPawsOffMeYouDamnDirtyApe();
Charleton Heston, 1968
```

上一章从一般意义上讨论了 COM 接口的基础，并特别讨论了 `IUnknown` 接口。同时也展示了在 C++ 中管理接口指针的技术，并细致讨论了 `IUnknown` 的实现技术。但是上一章并没有谈到客户如何获取指向对象的第一个接口指针，或者说，对象的实现者如何使它们的对象能够被外界所发现。这一章将讲述 COM 对象如何被集成到 COM 运行时环境中，以使客户可以找到或者创建给定类型的对象。

### 3.1 再谈接口与实现

上一章把 COM 接口定义为一组操作的抽象集合，这些操作表达了对象可以引出的一种功能特性。COM 接口是以 IDL 的形式定义的，它们有一个逻辑名字说明它们所针对的功能。下面是 COM 接口 `IApe` 的 IDL 定义：

```
[object, uuid(753A8A7C-A7FF-11d0-8C30-0080C73925BA) ]
interface IApe : IUnknown {
    import "unknwn.idl";
    HRESULT EatBanana(void);
    HRESULT SwingFromTree (void);
    [propget] HRESULT Weight([out, retval] long *plbs);
}
```

IApe 对应的文档应该粗略地说明这三个操作 EatBanana、SwingFromTree 和 Weight 的语义。所有通过 `QueryInterface` 暴露 IApe 的对象必须保证，它们对这些函数的实现完全符合 IApe 的语义约定。然而，接口定义总是故意留下一些空间让对象自己来解释。这意味着客户永远也不可能完全确定任一个给定方法的精确行为方式，只能确定它的行为将遵从接口文档中所描述的粗略指导。这种“受控制的不确定程度”正是多态性的基本特征，也是面向对象软件开发的基础之一。

考虑刚才给出的 IApe 接口，极有可能（事实上也往往是这样）IApe 接口将存在多个实现。因为 IApe 的定义对于所有的实现来说很具一般性，所以“客户要求做出 EatBanana 方法的行为”这一假设必须具有足够的模糊性，以便允许大猩猩、黑猩猩和猩猩（gorilla、chimpanzee、orangutan，它们都有可能实现 IApe 接口）对这个操作都有合法（但是有一些细微不同）的解释。没有这种灵活性，则不可能有多态性。

COM 清晰地把接口（interface）、实现（implementation）和类（class）看作三个不同的概念。接口是与对象进行通信的抽象协议。实现是支持一个或者多个接口的具体数据类型，它为接口的每个抽象操作都提供了精确的语义解释。类是被命名的实现，它代表了具体的、可实例化的类型，其正式名称为 COM 类或者 coclass。

从封装的意义上来看，有关 COM 类的所有信息就是它的名字，以及它可能暴露的一组接口列表。与 COM 接口类似，COM 类也使用 GUID 来命名，只不过当 GUID 被用来命名 COM 类时，它被称为 CLSID。如同接口名字一样，客户在使用这些类名之前，一定要先知道它们。由于 COM 接口为了允许多态性而保留了语义上的模糊性，所以 COM 不允许客户简单地请求某个接口的“任一个可供使用的”实现。相反，客户必须指定它所期望的精确实现。这进一步加强了“COM 接口只是简单的抽象通信协议”这一事实，接口的唯一目标就是允许客户可以与具体的对象进行通信（这些对象属于具有实际意义的实现类）。<sup>1</sup>

除了用 CLSID 来命名实现之外，COM 也支持文本方式的别称，被称为 programmatic ID 或者 ProgID。ProgID 采用库名.类名.版本的形式，而且，与 CLSID 不同的是，我们只是为了方便而要求 ProgID 是唯一的。通过 COM API 函数 CLSIDFromProgID 和

<sup>1</sup> 虽然对给定的接口请求“任一可供使用的实现”这种做法并没有意义，但是有时候，如果把一组共享某些高级特征（比如“是一种动物”或者“提供了日志记录服务”）的实现组织起来，则又是很有意义的。为了支持这种类型的组件发现，COM 通过组件类别(component category)支持组件分类法。虽然通常情况下，属于同一组件类别的所有类都将实现同一组共同的接口，但是这并不是“属于同一个组件类别”的充分条件。

ProgIDFromCLSID，客户可以在 ProgID 和 CLSID 两者之间进行转换，这两个函数的原型如下：

```
HRESULT CLSIDFromProgID(
    [in, string] const OLECHAR *pwszProgID,
    [out]        CLSID *pcsid);
HRESULT ProgIDFromCLSID(
    [in] REFCLSID rclsid,
    [out, string] OLECHAR **ppwszProgID);
```

为了把一个 ProgID 转换为 **CLSID**，我们只需简单地调用 CLSIDFromProgID：

```
HRESULT GetGorillaCLSID(CLSID& rclsid) {
    const OLECHAR wszProgID[] = OLESTR("Apes.Gorilla.I");
    return CLSIDFromProgID(wszProgID, &rclsid);
}
```

COM 运行库将会检查它的配置数据库，以便把 ProgID Apes.Gorilla.I 映射到对应的 COM 实现类的 CLSID。

## 3.2 类对象

所有 COM 类的一个**基本要求**是，它们必须有一个类对象（译注1）。对于每个类来说，类对象是独一无二的，它实现了该类的创建功能。对于给定的实现，类对象就像是它的元类，并且它所实现的方法扮演了 C++ 中静态成员函数的角色。逻辑上讲，每一个类只有一个类对象；然而，由于 COM 的分布式本质，对于给定的类，有可能每台机器上有一个类对象，或者每个用户账号有一个类对象，或者每个进程有一个类对象，这要取决于该类是如何被配置的。类实现的第一个进入点就是通过它的类对象而实现的。

类对象是非常有用的**编程抽象**。类对象可以担当起知名对象（它们的 CLSID 可以作为对象的名字）的角色，允许多个客户根据指定的 CLSID 绑定到同一个对象上。虽然整个系统也有可能使用类对象建立起来，但是类对象往往被用作中介代理（broker），用来创建一个类的新实例，或者根据某些知名的对象名字找到现有的对象实例。当类对象被这样使用时，它通常只暴露一个或者两个中间接口，允许客户创建或者找到对象的实例，最终由这些实例来完成客户感兴趣的工作。例如，考虑前面描述的接口 IApe，对于类对象来说，它也可以暴露 IApe 接口，这样做并没有违背 COM 的规则：

```
class GorillaClass : public IApe {
```

---

译注 1：class object，在其他一些书籍中被称为 class factory，即类厂。

```

public:
    // 类对象是独一无二的, 不要删除
    IMPLEMENT_UNKNOWN_NO_DELETE(GorillaClass)
    BEGIN_INTERFACE_TABLE(GorillaClass)
        IMPLEMENTS_INTERFACE(IApe)
    END_INTERFACE_TABLE()
    // IApe 方法
    STDMETHODIMP EatBanana(void),
    STDMETHODIMP SwingFromTree(void);
    STDMETHODIMP get_Weight(long *plbs);
};

```

既然这个 C++类同时也被当作一个 COM 类对象，于是在任何给定的时刻都只有一个大猩猩（gorilla）。对于某些情况，单一实例也是非常需要的。然而，在上面的大猩猩例子中，客户极有可能想要建立多个应用，而且这些应用各自使用不同的大猩猩实例。为了支持这种用法，大猩猩类对象不应该暴露引出 IApe 接口，相反，它应该引出一个新的接口，允许客户创建新的大猩猩实例，或者通过名字找到现有的大猩猩实例。这就要求实现者必须定义两个 C++类：一个用来实现类对象，另一个实现实际的类实例。对于大猩猩实现来说，定义大猩猩实例的 C++类将实现 IApe：

```

class Gorilla : public IApe{
public:
    // 实例是基于堆的, 完成后删除
    IMPLEMENT_UNKNOWN(Gorilla)
    BEGIN_INTERFACE_TABLE(Gorilla )
        IMPLEMENTS_INTERFACE(IApe)
    END_INTERFACE_TABLE()
    // IApe 方法
    STDMETHODIMP EatBanana(void),
    STDMETHODIMP SwingFromTree(void);
    STDMETHODIMP get_Weight(long *plbs);
};

```

我们有必要再定义一个接口，其中包含 Gorilla 类对象将要实现的操作：

```

[object,uuid(753A8AAC-A7FF-11d0-8C30-0080C73925BA) ]
interface IApeClass : IUnknown {
    HRESULT CreateApe([out, retval] IApe **ppApe);
    HRESULT GetApe([in] long nApeID,
                  [out, retval] IApe **ppApe);
    [propget] HRESULT AverageWeight([out, retval] long *plbs);
}

```

有了这个接口定义之后，类对象将实现 IApeClass 的方法，CreateApe 方法创建 C++ 类 Gorilla 的新实例，而 GetApe 方法把某个对象名字（这里是一个整数）映射到特定的实例上。代码如下：

```

class GorillaClass : public IApeClass {
public:
    IMPLEMENT_UNKNOWN_NO_DELETE(GorillaClass)
    BEGIN_INTERFACE_TABLE(GorillaClass)
        IMPLEMENTS_INTERFACE(IApeClass)
    END_INTERFACE_TABLE()

    STDMETHODIMP CreateApe(IApe **ppApe) {
        if ((*ppApe = new Gorilla) == 0)
            return E_OUTOFMEMORY;
        (*ppApe)->AddRef();
        return S_OK;
    }

    STDMETHODIMP GetApe(long nApeID, IApe **ppApe) {
        // 假定其他地方有一个包含知名大猩猩的表格
        extern Gorilla *g_rgWellKnownGorillas[];
        extern int g_nMaxGorillas;
        // 确定 nApeID 是合法索引
        *ppApe = 0;
        if (nApeID > g_nMaxGorillas || nApeID < 0)
            return E_INVALIDARG;
        // 假定 ID 就是表格索引
        if ((*ppApe = g_rgWellKnownGorillas [nApeID]) == 0)
            return E_INVALIDARG;
        (*ppApe)->AddRef();
        return S_OK;
    }

    STDMETHODIMP get_AverageWeight(long *plbs) {
        extern Gorilla *g_rgWellKnownGorillas[];
        extern int g_nMaxGorillas;
        *plbs = 0; long lbs;
        for (int i = 0; i < g_nMaxGorillas; i++) {
            g_rgWellKnownGorillas[i] ->get_Weight(&lbs);
            *plbs += lbs;
        }
        // 假定 g_nMaxGorillas 非零
        *plbs /= g_nMaxGorillas;
        return S_OK;
    }
};

```

注意，以上代码假定已经存在一个包含现有大猩猩实例的外部表，这个表可以是由 Gorilla 实例本身来维护的，也可以是由其他某个代理来维护的。

### 3.3 激活

客户需要有一种机制来找到类对象。因为 COM 的动态特性，这将会涉及到装载 DLL 或者启动服务进程。“把一个对象带入到活动状态”的动作被称为对象激活 (activation)。

COM 有三种激活模型，可以用来把对象带入到内存中，并允许客户调用它的方法。客户可以要求 COM 绑定到给定类的类对象。客户也可以要求 COM 根据 CLSID 所代表的类创建一个新实例。最后，客户还可以要求 COM 根据对象的永久状态把它带入到活动状态。在这三种模型中，只有第一种模型（绑定到一个类对象）是绝对必需的。其他两种模型只是简单地将通常使用的激活概念进行了优化。用户自定义的其他激活模型可以建立在这三种基本模型的基础之上。

COM 的三种激活模型都用到了 COM 服务控制管理器 (SCM)<sup>2</sup> 所提供的服务。SCM 是一台机器上所有激活请求的中心控制点。凡是支持 COM 的每台主机都有它自己本地的 SCM，由它把远程激活请求转发给远程机器上的 SCM，在远程机器上这个激活请求将被当作本地激活请求来对待。SCM 只被用于激活对象和绑定初始接口指针。一旦一个对象已经被激活了，以后 SCM 就不再介入到客户与对象之间的方法调用过程中。如图 3.1 所示，在 Windows NT 平台上，SCM 是由 RPCSS 服务实现的。SCM 的服务将以高层名字对象类型 (moniker types)<sup>3</sup> 或者底层 API 函数的形式暴露给应用程序，所有这些功能都是在 COM 库 (COM library, 这是 COM 规范中的叫法) 中实现的。在 Windows NT 下，COM 库的绝大多数功能是在 OLE32.DLL 中实现的。为了提高效率，COM 库可以使用本地的或者被缓存的状态，以避免应用进程与 RPCSS 服务之间进行不必要的进程间通信 (IPC)。

回忆一下，我们以前提到过，COM 的第一个原则就是接口与实现的分离。对客户所隐藏的实现细节之一就是“对象实现驻留在哪里”。我们不但不能检测到一个对象是被激活在哪台机器上，而且也不可能检测到本地对象是被激活在客户的进程中，还是被激活在本地机器上另外一个进程中。这使对象实现者拥有极大的灵活性，他可以决定如何配置对象实现，把对象实现配置在哪里，可以考虑可靠性、安全性、负载平衡以及性能等因素。客户在激活时刻还有机会可以指定对象将在哪里被激活。然而，许多客户其实并不在乎对象的位置，所以这种情况下，SCM 将根据目标类的当前配置情况来作出决定。

---

<sup>2</sup> Windows NT 也有一个被称为服务控制管理器的子系统，它被用来启动与登录过程独立的进程，即 Windows NT Services。在本书的剩余部分，我将把 Windows NT 服务控制管理器称为 NT SCM，以区别于 COM SCM。

<sup>3</sup> 名字对象 (moniker) 也是定位器对象 (locator object)，它把激活 (activation) 或者绑定 (binding) 算法的细节隐藏起来了。本章后文将详细讨论名字对象。

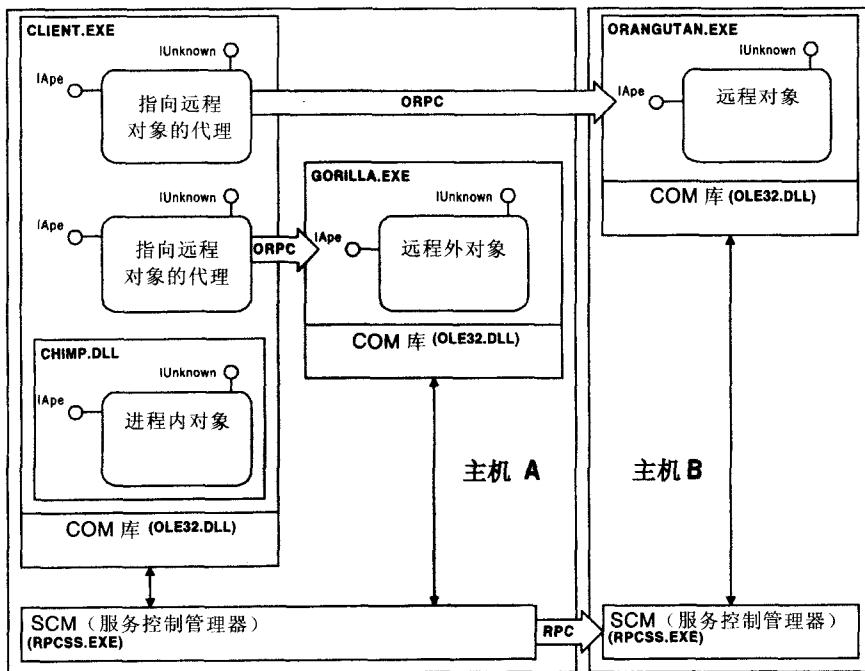


图 3.1 COM 库和 SCM

当一个对象被激活在客户进程内部时，实现该对象方法的 DLL 将被装载到客户进程中，对象的所有数据成员都驻留在客户的地址空间中。这使方法调用非常高效，因为不需要进程切换。而且，如果客户的线程符合对象的线程要求，那么客户的线程有可能直接被用来执行方法代码。也就是说，如果客户和对象有兼容的线程要求，两者之间就不需要线程切换。当方法调用在客户线程上执行时，在对象被激活之后，客户和对象之间并没有中间运行库介入进来，所以方法调用的开销就如同简单的虚函数调用一样。这使得进程内 COM 特别适合于性能敏感的应用，因为方法调用并不比 DLL 内部的普通全局函数调用代价更高。<sup>4</sup>

当一个对象在另一个进程中（即在本地机器上不同的进程中，或者在远程机器的进程中）被激活时，实现对象方法的代码将在服务器进程中执行，对象的所有数据成员都驻留在服务器进程的地址空间中。为了允许客户可以与进程外对象进行通信，COM 在激活时刻透明地给客户返回一个代理对象。正如第 5 章将要详细讨论的那样，这个代理对象运行在客户的线程中，它把方法调用翻译成服务器执行环境中的 RPC 请求，然后在服务器执行环境中，这些 RPC 请求又被翻译回实际对象上的方法调用。这使得方法调用的效率要低一些，因为每次访问对象都需要线程切换和进程切换。进程外激活方式的好处包括错误隔离、分布式以及安全性的增强。第 6 章将讨论进程外激活的细节。

<sup>4</sup> “调用到外部 DLL 的函数”所需要的间接层次，大致上与“通过 vtbl 入口调用一个函数”的间接层次相当。

### 3.4 使用 SCM

上一节提到了 SCM 支持三种基本激活方式（绑定到类对象、绑定到类实例、绑定到来自文件的永久实例）。如图 3.2 所示，这些基本激活方式相互之间构成了一个逻辑层次结构。<sup>5</sup> 位于最底层的激活方式是绑定到类对象，这种方式也是最容易理解的。

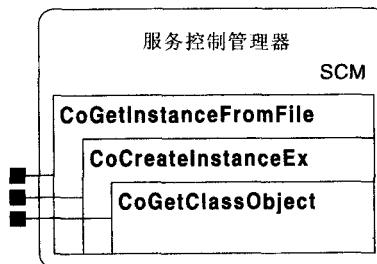


图 3.2 激活方式层次结构

客户不再手工装载类的代码，而是通过底层的 COM API 函数 CoGetClassObject 使用 SCM 的服务。这个函数请 SCM 把一个指针绑定到客户所请求的类对象上：

```
HRESULT CoGetClassObject(
    [in] REFCLSID rclsid,           // 哪个类对象?
    [in] DWORD dwClsCtx,           // 局部特征
    [in] COSERVERINFO *pcsi,        // 主机/安全信息
    [in] REFIID riid,              // 哪个接口?
    [out, iid_is(riid)] void **ppv); // 在此存放结果!
```

CoGetClassObject 的第一个参数说明了客户希望获得哪一个实现类。最后一个参数是指向要被绑定的接口指针的一个引用，第四个参数只是一个简单的 IID，说明了最后一个参数所指的接口指针的类型。最值得感兴趣的参数是第二个和第三个参数，它们决定了类对象应该在哪里被激活。

CoGetClassObject 把它的第二个参数当作一个位掩码，允许客户指定待创建对象的可能特征以及可靠性特征（也即对象应该运行在进程内部、进程外部还是在另外一台机器上）。位掩码的有效值定义在标准的 CLSCTX 枚举类型中：

```
enum tagCLSCXTX {
    CLSCTX_INPROC_SERVER      = 0x1,          // 进程内运行
```

<sup>5</sup> 这个层次划分只是概念性的，因为 COM 库和有线协议(wire-protocol)把每种激活方式都按照独立的代码路径和包格式来实现。

```

CLSTX_INPROC_HANDLER = 0x2, // *
CLSTX_LOCAL_SERVER = 0x4, // 进程外运行
CLSTX_REMOTE_SERVER = 0x10 // 主机外运行
} CLSTX;

```

这些标志可以通过按位或 (OR) 组合在一起，当客户请求的多个 CLSTX 都可以使用时，COM 将会选择最有效率的服务器类型（这意味着 COM 将尽可能地使用掩码的最低位）。SDK 头文件也包含几个快捷方式的宏，它们代表了许多常见情况下所使用的 CLSTX 标志组合：

```

#define CLSTX_INPROC (CLSTX_INPROC_SERVER| \
CLSTX_INPROC_HANDLER)
#define CLSTX_SERVER (CLSTX_INPROC_SERVER| \
CLSTX_LOCAL_SERVER| \
CLSTX_REMOTE_SERVER)
#define CLSTX_ALL (CLSTX_INPROC_SERVER| \
CLSTX_INPROC_HANDLER| \
CLSTX_LOCAL_SERVER| \
CLSTX_REMOTE_SERVER)

```

注意，诸如 Visual Basic 和 Java 这样的环境总是使用 CLSTX\_ALL，这表示任何可供使用的实现都可以满足需要。

**CoGetClassObject** 的第三个参数是一个结构指针，它所指向的结构包含远程信息和安全信息。该结构的类型为 **COSERVERINFO**，它允许客户显式地指定在哪台机器上激活对象，以及如何配置“用于发出对象激活请求”的安全设置：

```

typedef struct _COSERVERINFO {
    DWORD     dwReserved1;      // 保留，必须为 0
    LPWSTR    pwszName;        // 主机名或 null
    COAUTHINFO *pAuthInfo;     // 安全设定
    DWORD     dwReserved2;      // 保留，必须为 0
} COSERVERINFO;

```

如果客户并没有指定主机名字，而是只使用 CLSTX\_REMOTE\_SERVER 标志，那么 COM 将使用针对每个 **CLSID** 的配置信息，来决定应该在哪台机器上激活对象。如果客户显式地传送了一个主机名字，那么这个主机名字将会替换任何其他的预配置主机名。如果客户不希望给 **CoGetClassObject** 传递显式的安全信息或者主机名字，那么它可以传递一个 null (**COSERVERINFO**) 指针。

讨论了 **CoGetClassObject** 的可能性之后，客户可以请 SCM 把一个接口指针绑定到

---

<sup>6</sup> 进程内处理器(handler)基本上是从 OLE 文档的沿用下来的。进程内处理器是一些作为对象的客户端代表的进程内组件，该组件实际上处于不同的进程中。OLE 文档中，处理器用于缓存客户端的绘图，以减少屏幕重绘时的 IPC 流量。虽然处理器通常是有意义的，但它并未在 OLE 文档之外得到广泛使用。Windows 2000 为实现处理器将提供附加的设施，但写作时细节尚不完全清楚。

相应的类对象上：

```

HRESULT GetGorillaClass(IApeClass * &rpgc) {
    // 声明 Gorilla 的 CLSID 为 GUID
    const CLSID CLSID_Gorilla = { 0x571F1680, 0xCC83, 0x11d0,
        { 0x8C, 0x48, 0x00, 0x80, 0xC7, 0x39, 0x25, 0xBA } };
    // 直接调用 CoGetClassObject
    return CoGetClassObject (CLSID_Gorilla, CLSCTX_ALL, 0,
        IID_IApeClass, (void**)&rpgc);
}

```

注意，如果被请求的类可以被作为进程内服务器来使用，那么 COM 将会自动地装载相应的 DLL，并调用一个知名引出函数，该函数会返回一个指向客户所请求的类对象的指针。然后，COM 把这个指针返回给客户，让客户直接访问它所期望的类对象。<sup>7</sup>一旦 CoGetClassObject 调用完成之后，COM 库和 SCM 将不再介入到客户与对象（类对象或者目标对象）之间的通信过程中。如果只有进程外或者远程服务器对象可供使用，那么 COM 将返回一个指向代理的指针，客户通过这个代理访问进程外或者远程类对象。

我们来回忆一下，IApeClass 接口的设计目的是使客户能够找到或者创建给定类的实例，考虑下面的例子：

```

HRESULT FindAGorillaAndEatBanana(long nGorillaID) {
    IApeClass *pgc = 0;
    // 通过 CoGetClassObject 寻找类对象
    HRESULT hr = CoGetClassObject(CLSID_Gorilla, CLSCTX_ALL,
        0, IID_IApeClass, (void**)&pgc);
    if (SUCCEEDED(hr)) {
        IApe *pApe = 0;
        // 使用类对象寻找现存的大猩猩
        hr = pgc->GetApe(nGorillaID, &pApe);
        if (SUCCEEDED(hr)) {
            // 让特定大猩猩吃香蕉
            hr = pApe->EatBanana();
            pApe->Release ();
        }
        pgc->Release();
    }
    return hr;
}

```

这个例子使用 Gorilla 的类对象找到一个已经被命名的对象，并通知它去吃香蕉。要使这个例子能够工作，某个外部过程必须向调用者提供知名大猩猩的名字。如果匿名的大猩猩也能够满足客户请求的话，那么下面的例子可以做到这一点：

---

<sup>7</sup> 技术上讲，类的并发性要求必须与调用线程一致。

```

HRESULT CreateAGorillaAndEatBanana(void) {
    IApeClass *pgc = 0;
    // 通过 CoGetClassObject 寻找类对象
    HRESULT hr = CoGetClassObject (CLSID_Gorilla, CLSCTX_ALL,
        0, IID_IApeClass, (void**)&pgc);
    if (SUCCEEDED(hr)) {
        IApe *pApe = 0;
        // 使用类对象创建新的大猩猩
        hr = pgc->CreateApe(&pApe);
        if (SUCCEEDED(hr)) {
            // 让大猩猩吃香蕉
            hr = pApe->EatBanana();
            pApe->Release ();
        }
        pgc->Release();
    }
    return hr;
}

```

注意，除了各自使用 IApeClass 不同的方法之外，这两个例子完全一样。因为类对象可以引出任意复杂的接口，它们可以被用来构建复杂的对象激活机制、初始化机制和定位策略。

## 3.5 类和服务器

COM 服务器是一个二进制文件，它包含一个或者多个 COM 类的方法代码。服务器既可以被包装成一个动态链接库，也可以被包装成一个普通的可执行文件。无论哪种情况下，SCM 都会负责自动装载服务器程序。

如果一个对象的激活请求指明“进程内激活”方式，那么必须存在服务器程序的 DLL 版本，以便能够把它装载到客户的地址空间中。相反，如果对象的激活请求指明了“进程外激活”或者“远程激活”方式，那么在指定的目标主机上（有可能与客户在同一台主机上）必须要有一个可执行文件以便能够启动服务器进程。COM 也支持把基于 DLL 的服务器装载到代理进程中，从而使遗留下来的进程内服务器也能够实现进程外激活或者远程激活。第 6 章讨论了代理进程如何与进程外或者远程激活联系起来的细节信息。

为了允许客户在激活对象时无需考虑组件包的类型或者文件被安装在哪里，COM 记录了一个配置数据库，将 CLSID 映射到实现该类的服务器程序。在 Windows 2000 或者更高的版本下，此配置数据库的主要存储位置是 NT 目录（译注 1）。NT 目录是一个

---

译注 1：现已改名为活动目录(active directory)。

分布式的安全数据库，记录了用户帐号、主机信息，以及其他管理信息。NT 目录也可以包含有关 COM 类的信息。这些信息被保存在目录中被称为 COM 类存储（COM Class Store）的部分。COM 使用类存储来把 CLSID 解析到实现文件名（在本地激活请求的情况下）或者远程主机名字（在远程激活请求的情况下）。当针对某个 CLSID 的激活请求作用到给定的机器上时，COM 首先要在本地一个缓冲区中进行查找。如果在本地缓冲区中没有配置信息可供利用，于是 COM 给类存储发送一个请求，要求本地机器准备好相应的实现文件以供使用。这有可能意味着只是简单地在本地缓冲区中加入一些信息，然后把请求重定向到其他的机器上；或者可能会导致把类的实现文件下载到本地机器上，然后运行安装程序。无论在哪种情况下，一旦一个类已经被注册到类存储中，那么客户的激活请求就可以得到满足，除非有安全限制。

在前面讨论类存储（class store）时提到的本地缓冲区，其正式名称为注册表（Registry）。注册表是每台机器上一个以文件形式存在的层次数据库，COM 利用注册表把 CLSID 映射到文件名（如果是本地激活的话）或者远程机器名字（如果是远程激活的话）。在 Windows NT 5.0（即 Windows 2000）之前，注册表也是保存 COM 配置信息的唯一位置。基于层次结构形式的键（key），我们可以迅速有效地搜索注册表中的信息，而键的名字则是以反斜杠为分割符的字符串。注册表中每个键都可以有一个或者多个值（value），这些值可以包含字符串，也可以包含整数值，或者二进制数据。在 Windows NT 4.0 的 COM 实现中，大多数 COM 配置信息都被保存在这个键的下面：

HKEY\_LOCAL\_MACHINE\Software\Classes

而大多数程序使用更为方便的别名：

HKEY\_CLASSES\_ROOT

在 Windows NT 5.0 的 COM 实现中，它继续使用 HKEY\_CLASSES\_ROOT 作为机器范围内的设置信息，但是也允许针对每个用户的 CLSID 配置信息，以便提供更大的灵活性和安全性。在 Windows NT 5.0 下面，COM 首先查找：

HKEY\_CURRENT\_USER\Software\Classes

然后检查 HKEY\_CLASSES\_ROOT。为了符号表达方便起见，我们通常使用缩写 HKLM、HKCR 和 HKCU 分别代表 HKEY\_LOCAL\_MACHINE、HKEY\_CLASSES\_ROOT 和 HKEY\_CURRENT\_USER。<sup>8</sup>

COM 把机器范围内与 CLSID 相关的信息都保存在注册表键 HKCR\CLSID 的下面。在 Windows NT 5.0 或者更高的版本下，COM 在下面的键下查找每个用户的类信息：

HKCU\Software\Classes\CLSID

---

<sup>8</sup> 在源代码或者配置文件中这些缩写不是合法的。它们的主要用意是使很长的键名可以出现在文档（或者其他 COM 资料）的一行内而不会被打断。但是你在讲述这些缩写词或者在源代码中输入时，应该把它们展开。

在这两个键（HKCR\CLSID 和 HKCU\Software\Classes\CLSID）下面，本地已知的 CLSID 都将被保存起来，每个 CLSID 有一个对应的子键。例如，本章前面用到的 Gorilla 类可能有一个机器范围的表项：

```
[HKCR\CLSID\{571F1680-CC83-11d0-8C48-0080C73925BA}]9  
@="Gorilla"
```

为了支持 Gorilla 对象的本地激活功能，注册表中 Gorilla 的 CLSID 表项必须有一个子键用于指示哪个文件包含该类方法的可执行代码。如果服务器程序被包装成 DLL，那么下面的表项将是必需的：

```
[HKCR\CLSID\{571F1680-CC83-11d0-8C48-0080C73925BA}\InprocServer32]  
@="C:\ServerOfTheApes.dll"
```

为了指示方法代码被包含在一个可执行文件中，那么下面的表项将是必需的：

```
[HKCR\CLSID\{571F1680-CC83-11d0-8C48-0080C73925BA}\LocalServer32]  
@="C:\ServerOfTheApes.exe"
```

同时提供上面两个表项也是合法的，这样客户可以根据各种可能的要求（比如可靠性要求）选择合适的位置信息。为了支持 ProgIDFromCLSID 函数，下面的子键是必需的：

```
[HKCR\CLSID\{571F1680-CC83-11d0-8C48-0080C73925BA}\ProgID]  
@="Apes.Gorilla.1"
```

反之，为了支持 CLSIDFromProgID 函数，则下面的子键是必需的：

```
[HKCR\Apes.Gorilla.1]  
@="Gorilla"  
[HKCR\Apes.Gorilla.1\CLSID]  
@="{571F1680-CC83-11d0-8C48-0080C73925BA}"
```

ProgID 是可选的，但是建议最好支持 ProgID，这样可使那些不能很方便地处理 CLSID 的环境也能够实现激活调用。

所有实现良好（well-implemented）的 COM 服务器都支持自注册（self-registration）。对于进程内服务器而言，这意味着 DLL 必须要引出下面两个知名函数：

```
STDAPI DllRegisterServer(void);  
STDAPI DllUnregisterServer(void);
```

---

<sup>9</sup> 此处使用了标准 REGEDIT4 语法。括号内的字符串与键名对应。该键下面的“名=值”对表示该键中保存的值。符号“@”说明该键的缺省值。

注意，STDAPI 只是一个简单的宏，它指示相应的函数返回一个 HRESULT，并且对于全局函数使用 COM 的标准调用习惯。这些函数必须要被显式地引出来，或者使用模块定义文件和链接器开关，或者使用编译指示符。类存储（Class Store）在把类的实现文件下载到客户机器上之后，利用这些函数来配置本地缓冲区。除了类存储要用到这些函数之外，其他各种环境（比如 MTS、ActiveX Code Download，以及其他各种安装工具）也利用这些知名函数在机器上安装或者删除服务器程序。Win32 SDK 包含一个工具 REGSVR32.EXE，它可以利用这些知名的引出函数来安装或者删除 COM 进程内服务器。

进程内服务器在实现 DllRegisterServer 和 DllUnregisterServer 时，必须要调用到注册表函数，以便插入或者删除适当的键（这些键把服务器的 CLSID 和 ProgID 映射到服务器的文件名）。尽管可以有各种各样的技术来实现这样的功能，但是最灵活也最有效的技术是创建一个字符串表格，这个表格包含适当的键（key）、值（value）的名字和值本身，然后简单地对表格中的项目分别调用 RegSetValueEx（安装时）或者 RegDeleteKey（删除时）。为了用这项技术来实现注册操作，服务器只需简单地定义一个  $N \times 3$  的字符串数组，数组中每一行包含键字符串、值的名字字符串以及值本身的字符串表示，举例如下：

```
const char *g_RegTable[][3] = {
    // 格式为{ 键, 值的名字, 值 }
    { "CLSID\\{\ 571F1680-CC83-11d0-8C48-0080C73925BA\}", 0,
        "Gorilla" },
    { "CLSID\\{\ 571F1680-CC83-11d0-8C48-0080C73925BA\}\InprocServer32",
        0, (const char*)-1 // 除去表示文件名的值
    },
    { "CLSID\\{\571F1680-CC83-11d0-8C48-0080C73925BA\}\ProgID",
        0, "Apes.Gorilla.1"
    },
    { "Apes.Gorilla.1", 0, "Gorilla" },
    { "Apes.Gorilla.1\\CLSID",
        0, "{571F1680-CC83-11d0-SC48-00S0C73925BA}" },
};
```

有了这张表之后，DllRegisterServer 的实现过程就非常直截了当：

```
STDAPI DllRegisterServer(void) {
    HRESULT hr = S_OK;
    // 寻找服务器的文件名
    char szFileName [MAX_PATH];
    GetModuleFileNameA(g_hinstDll, szFileName, MAX_PATH);
    // 注册表中的项
    int nEntries = sizeof(g_RegTable)/sizeof(*g_RegTable);
    for (int i = 0; SUCCEEDED(hr) && i < nEntries; i++) {
        const char *pszKeyName = g_RegTable[i][0];
```

```

const char *pszValueName = g_RegTable[i][1];
const char *pszValue = g_RegTable[i][2];
// 将值映射到模块文件名
if (pszValue == (const char*)-1)
    pszValue = szFileName;
HKEY hkey;
// 创建键
long err = RegCreateKeyA(HKEY_CLASSES_ROOT,
                         pszKeyName, &hkey);
if (err == ERROR_SUCCESS) {
// 设定键
    err = RegSetValueExA(hkey, pszValueName, 0,
                         REG_SZ, (const BYTE*)pszValue,
                         (strlen(pszValue) + 1));
    RegCloseKey(hkey);
}
if (err != ERROR_SUCCESS) {
// 如果不能添加键或值, 返回并表示失败
    DllUnregisterServer();
    hr = SELFREG_E_CLASS;
}
}
return hr;
}

```

对应的 `DllUnregisterServer` 应该像这样:

```

STDAPI DllUnregisterServer(void) {
    HRESULT hr = S_OK;
    int nEntries = sizeof(g_RegTable)/sizeof(*g_RegTable);
    for (int i = nEntries - 1; i >= 0; i--) {
        const char *pszKeyName = g_RegTable[i][0];
        long err = RegDeleteKeyA(HKEY_CLASSES_ROOT, pszKeyName);
        if (err != ERROR_SUCCESS)
            hr = S_FALSE;
    }
    return hr;
}

```

注意, `DllUnregisterServer` 的实现过程是倒着遍历整个表格, 从最后一个表项开始向前删除。这样可以克服 `RegDeleteKey` 的限制, 因为 `RegDeleteKey` 只允许没有子键的键才能够被删除。`DllUnregisterServer` 的实现过程假定表格中的项目是这样安排的: 一个键的所有子键一定出现在父键的项目之后。

一旦 COM 已经把 `CLSID` 映射到指定的实现文件了, 我们还需要有一些标准的技术, 用来把服务器的类对象暴露给 COM。对于以可执行文件为基础的服务器程序来说, COM 提供了显式的 API 函数可以把类对象与它们的 `CLSID` 联系起来。这些 API 函数的细节

将在第 6 章讨论。对于以 DLL 为基础的服务器来说，DLL 必须要引出一个知名函数，在外界需要类对象时 CoGetClassObject 将调用这个知名函数。这个函数必须使用模块定义文件来引出，并且具有如下的原型：

```
HRESULT DllGetClassObject(
    [in] REFCLSID rclsid, // 哪个类对象?
    [in] REFIID riid,    // 哪个接口?
    [out, iid_is(riid)] void **ppv); // 结果存放在这里!
```

考虑到效率，也考虑到方便起见，给定的服务器可以包含多个类的代码。DllGetClassObject 的第一个参数指示：被请求的是哪个类。第二和第三个参数只是简单地让函数向 COM 返回一个指定类型的接口指针。

考虑一个服务器实现了三个类：Gorilla、Chimp 和 Orangutan。服务器程序可能包含六个不同的类：其中三个构造每个类的实例，另外三个构造每个类的类对象。在这种情形下，服务器的 DllGetClassObject 实现看起来可能像这样：

```
STDAPI DllGetClassObject (REFCLSID rclsid,
                           REFIID riid, void **ppv) {
    // 为每个类定义一个单独类对象
    static GorillaClass s_gorillaClass;
    static OrangutanClass s_orangutanClass;
    static ChimpClass s_chimpClass;
    // 返回已知类的接口指针
    if (rclsid == CLSID_Gorilla)
        return s_gorillaClass.QueryInterface(riid, ppv);
    else if (rclsid == CLSID_Orangutan)
        return s_orangutanClass.QueryInterface(riid, ppv);
    else if (rclsid == CLSID_Chimp)
        return s_chimpClass.QueryInterface(riid, ppv);
    // 如到这里，表示 rclsid 是一个未实现的类
    // 以知名错误代码表示失败
    *ppv = 0;
    return CLASS_E_CLASSNOTAVAILABLE;
}
```

注意，这段代码并不关心每个类对象暴露哪个接口。它只是简单地把 QueryInterface 请求转发给适当的类对象。

下面的伪代码显示了 API 函数 CoGetClassObject 如何与服务器的 DllGetClassObject 发生关系：

```
// 摘自 OLE32.DLL 中的伪代码
HRESULT CoGetClassObject(REFCLSID rclsid, DWORD dwClsCtx,
                        COSERVERINFO *pcsi, REFIID riid, void **ppv) {
    HRESULT hr = REGDB_E_CLASSNOTREG;
    *ppv = 0;
```

```

if (dwClsCtx & CLSCTX_INPROC) {
    // 试图执行进程内激活
    HRESULT (*pfnGCO)(REFCLSID, REFIID, void**) = 0;
    // 在进程中已加载服务器表中查找
    pfnGCO = LookupInClassTable(rclsid, dwClsCtx);
    if (pfnGCO == 0) { // 还未加载!
        // 在类库或注册表中查找 DLL 名
        char szFileName [MAX_PATH];
        hr=GetFileFromClassStoreOrRegistry(rclsid,dwClsCtx,szFil eName);
        if (SUCCEEDED(hr)) {
            // 试图加载 DLL 并挖出 DllGetClassobject
            HINSTANCE hInst = LoadLibrary(szFileName);
            if (hInst == 0)
                return CO_E_DLLNOTFOUND;
            pfnGCO = GetProcAddress (hInst, "DllCetClassObject");
            if (pfnGCO == 0)
                return CO_E_ERRORINDLL;
        }
        // 缓存 DLL 以备后用
        InsertInClassTable(rclsid, dwClsCtx, hInst, pfnGCO);
    }
}
// 调用函数获取类对象指针
hr = (*pfnGCO)(rclsid, riid, ppv);
}
if ((dwClsCtx&(CLSCTX_LOCAL_SERVER|CLSCTX_REMOTE_SERVER))
    && hr == REGDB_E_CLASSNOTREG) {
}
return hr;
}

```

注意，**CoGetClassObject** 函数是唯一调用 **DllGetClassObject** 的实体。为了进一步强化这个事实，如果模块定义文件中引出函数 **DllGetClassObject** 项目上没有使用 **private** 关键字的话，那么 COM 可感知的链接器将会产生一个警告。因此，模块文件应该如下定义：

```

// 摘自 APELIB.DEF
LIBRARY APELIB
EXPORTS
    DllGetClassObject private

```

事实上，COM 可感知的链接器希望所有与 COM 有关的入口项都使用 **private** 关键字。

## 3.6 一般化 (generalization)

前一个例子把 IApeClass 接口当作类层次上的接口，也就是说，它只针对那些暴露出 IApe 接口的类。IApeClass 接口允许客户创建新的对象，或者找到现有的对象，但是无论哪种情况下，客户得到的结果对象必须实现 IApe 接口。如果一个新的类要想允许客户创建或者找到非 IApe 兼容的对象，那么它的类对象将需要实现其他的接口。因为创建和找到对象是一种很常见的需求，大多数的类都需要提供这样的支持，所以 COM 定义了一些标准接口，用来构造出一般化的“对象发现和创建”模型。一个用于对象发现 (object discovery) 的标准接口是 IOleItemContainer:

```
// from oleidl.idl
[ object, uuid(0000011c-0000-0000-C000-000000000046) ]
interface IOleItemContainer : IOleContainer {

    // 寻找名为 pszItem 的对象

    HRESULT GetObject(
        [in] LPOLESTR pszItem,                      // 哪个对象?
        [in] DWORD dwSpeedNeeded,                    // 时限
        [in, unique] IBindCtx *pbc,                 // 绑定信息
        [in] REFIID riid,                          // 哪个接口?
        [out, iid_is(riid)] void **ppv);           // 存在此处!

    // 为了清楚起见, 保留了已删除的方法
}
```

注意，GetObject 方法允许客户指定结果接口指针的类型。结果对象的实际类型是与环境相关的，取决于 IOleItemContainer 的具体实现。下面的例子请 Gorilla 类对象找到一个名为 “Ursus” 的对象：

```
HRESULT FindUrsus(IApe * &rpApe) {
    // 绑定 Gorilla 类对象的一个引用
    rpApe = 0;
    IOleItemContainer *poic = 0;
    HRESULT hr = CoGetClassObject(CLSID_Gorilla, CLSCTX_ALL, 0,
                                   IID_IOleItemContainer, (void**)&poic);
    if (SUCCEEDED(hr)) {
        // 寻找 Ursus
        hr = poic->GetObject(OLESTR("Ursus"),
                              BINDSPEED_INDEFINITE, 0,
                              IID_IApe, (void**)&rpApe);
```

```

    poic->Release();
}
return hr;
}

```

尽管这样的用法是完全合法的，但是 IOleItemContainer 接口的设计目的是与单项名字对象（Item Moniker）联合使用，本章后文将会讨论到单项名字对象。

COM 也为“对象创建（object creation）”定义了一个标准接口。这个接口被称为 IClassFactory：

```

// 摘自 unknwn.idl
[object, uuid d (00000001-0000-0000-C000-000000000046) ]
interface IClassFactory : IUnknown {
    HRESULT CreateInstance([in] IUnknown *pUnkOuter,
                          [in] REFIID riid,
                          [out, iid_is(riid)] void **ppv);
    HRESULT LockServer( [in] BOOL block);
}

```

尽管任何一个类的实例都可能会引出 IClassFactory 接口，但是通常只有类对象会引出这个接口。类对象并不必须要实现 IClassFactory，但是，为了统一起见，它们往往都会引出这个接口。在本书写作的时候，凡是需要集成到 MTS（Microsoft Transaction Server）环境中的类都必须实现 IClassFactory（实际上，MTS 并不能识别其他的类对象接口）。

IClassFactory 接口有两个方法：LockServer 和 CreateInstance。LockServer 是在进程外激活请求过程中，被 COM 内部调用的，第 6 章将对它进行细节讨论。CreateInstance 方法则被用来请求类对象创建该类的一个新实例。如同 IApeClass::CreateApe 的情形一样，即将被创建的对象的类型是由类对象来决定的，即由客户发送 CreateInstance 请求的类对象来决定的。CreateInstance 的第一个参数被用于聚合（aggregation）的情形，将在第 4 章进一步讨论。在本章的讨论过程中，这个参数一定是 null（空）。CreateInstance 的第二和第三个参数使得它可以向客户返回一个动态类型的接口指针（即运行时确定类类型的接口指针）。

假定 Gorilla 类对象引出了 IClassFactory 接口，而不是 IApeClass，于是客户现在可以使用 IClassFactory::CreateInstance 方法来创建新的 Gorilla 实例：

```

HRESULT CreateAGorillaAndEatBanana() {
    IClassFactory *pcf = 0;
    // 寻找类对象
    HRESULT hr = CoGetClassObject(CLSID_Gorilla, CLSCTX_ALL, 0,
                                  IID_IClassFactory, (void**)&pcf);
    if (SUCCEEDED(hr)) {
        IApe *pApe = 0;
        // 使用类对象创建一个大猩猩
        hr = pcf->CreateInstance(0, IID_IApe, (void**)&pApe);
    }
}

```

```

// 不再需要类对象, 释放它
pcf->Release();
if (SUCCEEDED(hr)) {
    // 让新的大猩猩吃香蕉
    hr = pApe->EatBanana(),
    pApe->Release ();
}
}
return hr,
}

```

这段代码在语义上等价于先前使用 `IApeClass` 接口的函数版本，只是使用 `IClassFactory` 接口代替 `IApeClass` 接口而已。

为了让上面的例子能够正常工作，`Gorilla` 类对象有必要实现 `IClassFactory`：

```

class GorillaClass : public IClassFactory {}
public:
    IMPLEMENT_UNKNOWN_NO_DELETE (GorillaClass)
    BEGIN_INTERFACE_TABLE (GorillaClass)
        IMPLEMENTS_INTERFACE (IClassFactory)
    END_INTERFACE_TABLE ()

    STDMETHODIMP CreateInstance(IUnknown *pUnkOuter,
                               REFIID riid, void **ppv) {
        *ppv = 0;
        if (pUnkOuter != 0) // 还不支持聚合
            return CLASS_E_NOAGGREGATION;
        // 创建 C++ 类 Gorilla 的一个新实例
        Gorilla *p = new Gorilla;
        if (p == 0)
            return E_OUTOFMEMORY;
        // 引用计数加 1
        p->AddRef();
        // 将结果接口指针存入 *ppv
        HRESULT hr = p->QueryInterface(riid, ppv);
        // 引用计数减 1, 从而
        // 在 QI 失败时删除对象
        p->Release();
        // 返回 Gorilla::QueryInterface 的结果
        return hr;
    }
    STDMETHODIMP LockServer(BOOL block);
};

```

`LockServer` 的实现过程将在本章后面进行讨论。注意 `CreateInstance` 的实现过程，它首先创建一个新的 `Gorilla` 对象，然后询问对象是否支持客户所请求的接口。如果对象确实支持被请求的接口，那么 `QueryInterface` 调用就会触发一次 `AddRef` 调用，客户

最终会调用与之相对应的 `Release`。如果 `QueryInterface` 失败的话，需要某种机制来删除刚才新建的对象。上面的例子使用了一种标准技术，即把 `QueryInterface` 调用放在 `AddRef` 和 `Release` 之间。如果 `QueryInterface` 调用失败的话，则 `Release` 调用将导致引用计数为 0，从而触发对象的自我删除动作。如果 `QueryInterface` 调用成功的话，则 `Release` 调用将导致引用计数为 1。剩下的一次计数属于客户，当客户不再需要这个对象的时候，它将会发出最后的 `Release` 调用。

### 3.7 优化 (Optimization)

由一个标准的接口来完成实例化工作，好处之一是 COM 可以为实例化过程提供更为有效的技术。考虑下面的代码，它创建了类 Chimp 的一个新实例：

```
HRESULT CreateChimp(/*~out*/ IAp * &rpApe) {
    extern const CLSID CLSID_Chimp;
    rpApe = 0;
    IClassFactory *pcf = 0;
    HRESULT hr = CoGetClassObject(CLSID_Chimp, CLSCTX_ALL, 0
        IID_IClassFactory, (void**)&pcf);
    if (SUCCEEDED(hr)) {}
    hr = pcf->CreateInstance(0, IID_IAp, (void**)&rpApe),
    pcf->Release();
}
return hr;
```

这段代码完成了一个操作：创建一个 Chimp 对象。注意，为了执行这一个操作，要求三个子操作（`CoGetClassObject`、`CreateInstance`、`Release`）。如果服务器程序是以进程内服务器的形式被装载到内存中的，那么这三个子操作将不会有很昂贵的开销。然而，如果服务器是一个进程外或者远程服务器，那么每个子操作都将要求在客户和服务器进程之间做一次来回数据传输。虽然 COM 使用了一种非常有效的 IPC/RPC 传输机制，但是每个子操作都将会有一个不可忽视的性能代价。理想的情况下，COM 到达服务器进程只需一次即可，然后在服务器上代表客户调用类对象的 `CreateInstance` 方法。如果类对象只是被用来实现 `IClassFactory` 的话，那么这项技术将比前面显示的“三步骤”技术要有效得多。COM 提供了一个 API 函数 `CoCreateInstanceEx`，它包含了 `CoGetClassObject` 和 `IClassFactory::CreateInstance` 的功能，这使创建一个新的对象只需一次来回数据传输。

`CoCreateInstanceEx` 允许客户指定将要被实例化的对象的 CLSID。在成功完成之后，`CoCreateInstanceEx` 返回一个或者多个接口指针，它们都指向新创建的对象实例。在使用 `CoCreateInstanceEx` 时，客户永远也不会看到中间“用来实例化该对象”的类对象。然而，服务器程序的实现者并不需要实现任何附加的功能，客户就可以调用

`CoCreateInstanceEx`。从服务器的角度来看，它所需要做的就是暴露出类对象，这也正是 `CoGetClassObject` 的要求。`CoCreateInstanceEx` 将使用与 `CoGetClassObject` 同样的技术，来找到相应的类对象。两者之间主要的区别在于，一旦类对象已经被找到了，`CoCreateInstanceEx` 在调用了 `IClassFactory::CreateInstance` 之后，可能会调用一个或者多个 `QueryInterface`，并且所有这些 `QueryInterface` 调用都在类对象的进程内部执行。如果激活请求是由另一个进程发出的，那么这将会很可观的提高性能。

与 `CoGetClassObject` 一样，`CoCreateInstanceEx` 也允许客户指定它所希望的 `CLSID` 和 `COSERVERINFO` 参数。而且，`CoCreateInstanceEx` 允许客户请求多个指向新建对象的接口指针。为了做到这一点，它让客户传送一个 `MULTI_QI` 结构数组，当服务器进程在执行时，它要使用这些结构的信息来调用新实例的 `QueryInterface`，`MULTI_QI` 结构如下：

```
typedef struct tagMULTI_QI {
    // 需要哪个接口?
    const IID *piid;
    // 输入 null, 输出包含指针
    [iid_is(piid)] IUnknown *pItf;
    // 输出包含 QueryInterface 的 HRESULT
    HRESULT hr;
} MULTI_QI;
```

由于 `CoCreateInstanceEx` 允许客户请求多个指向新对象的接口指针，所以即使客户希望获得多个类型的接口指针，它也不需要显式地调用 `QueryInterface`。因为这些 `QueryInterface` 是在类对象进程的内部，由 COM 代表客户发出的，所以这里并不需要附加的“客户\_对象”之间的来回数据传输过程。注意，`CoCreateInstanceEx` 返回的每个接口指针都指向同样的对象。COM 并没有提供直接的操作，可以在一次来回数据传输过程中创建多个实例。

理解了 `MULTI_QI` 的结构之后，`CoCreateInstanceEx` 函数的定义就非常简单、非常易于理解了，如下：

```
HRESULT CoCreateInstanceEx(
    [in] REFCLSID rclsid,           // 哪一种对象?
    [in] IUnknown *pUnkOuter,        // 用于聚合
    [in] DWORD dwClscxt,           // 局部特征
    [in] COSERVERINFO *pcsi,         // 主机/安全信息
    [in] ULONG cmqi,                // 接口数目?
    [out, size_is(cmqi)] MULTI_QI *prgmq); // 存放位置
```

如果新对象能够提供客户在 `CoCreateInstanceEx` 中请求的所有接口，那么 `CoCreateInstanceEx` 返回 `S_OK`。如果至少有一个接口可以提供，但是又不能提供所有的接口，那么 `CoCreateInstanceEx` 返回 `CO_S_NOTALLINTERFACES`，表示部分成功。然后调用

者可以分别检查每个 MULTI\_QI 结构的 HRESULT 值，以便确定哪些接口可以使用，哪些接口不能使用。如果 CoCreateInstanceEx 不能够创建对象，或者客户请求的所有接口都不能提供，那么 CoCreateInstanceEx 返回一个以 SERVERITY\_ERROR 为基础的 HRESULT，说明为什么操作失败。

当客户需要多个类型的接口时，CoCreateInstanceEx 是非常高效的。考虑下面新加的接口定义：

```
[object, uuid(753A8F7C-A7FF-11d0-8C30-0080C73925BA)]
interface IEgghead : IUnknown {
    import "unknwn.idl";
    HRESULT ContemplateNavel (void);
}
```

有了这个接口定义之后，客户现在可以在一次“客户\_服务器”来回数据传输过程中，同时把两种类型的指针绑定到同一个新的黑猩猩（chimp）对象上。代码如下：

```
void CreateChimpEatBananaAndThinkAboutIt(void) {
    // 声明并初始化一个 MULTI_QI 的数组
    MULTI_QI rgmqi[2] =
        { { &IID_IApe, 0, 0 }, { &IID_IEgghead, 0, 0 } };

    HRESULT hr = CoCreateInstanceEx(
        CLSID_Chimp, // 创建一个新的黑猩猩
        0,           // 无聚合
        CLSCTX_ALL, // 任何局部
        0,           // 无显示主机/安全信息
        2,           // 两个接口
        rgmqi);      // MULTI_QI 结构数组

    if (SUCCEEDED(hr)) {
        // hr 可能是 CO_S_NOTALLINTERFACES，因此检测每个结果
        if (hr == S_OK || SUCCEEDED(rgmqi[0].hr)) {
            // 随便将结果指针转换为
            // 对应用于请求接口的 IID 的类型，都是安全的
            IApe *pApe = reinterpret_cast<IApe*>(rgmqi[0].pItf);
            assert(pApe);
            HRESULT hr2 = pApe->EatBanana();
            assert(SUCCEEDED(hr2));
            pApe->Release();
        }
        if (hr == S_OK || SUCCEEDED(rgmqi[1].hr)) {
            IEgghead *peh =
                reinterpret_cast<IEgghead*>(rgmqi[1].pItf);
        }
    }
}
```

```

        assert (peh);
        HRESULT hr2 = peh->ContemplateNavel();
        assert(SUCCEEDED (hr2));
        peh->Release ();
    }
}
}
}

```

图 3.3 说明了 CoCreateInstanceEx 创建一个新对象时所执行的步骤。很重要的一点是，两个结果指针都指向同一个对象。如果客户要求两个不同的对象，那么它必须调用 CoCreateInstanceEx 两次。

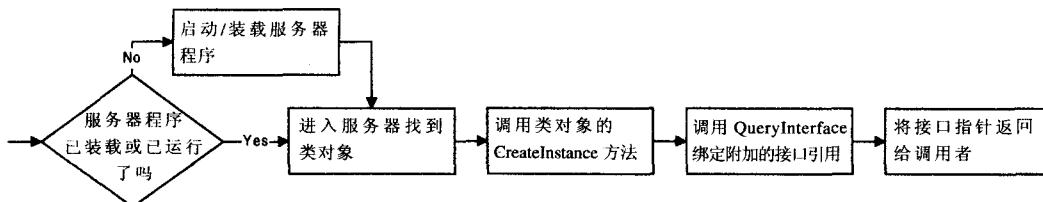


图 3.3 CoCreateInstanceEx 执行过程示意图

如果客户只希望得到一个接口指针的话，那么 CoCreateInstanceEx 的用法就非常直截了当：

```

HRESULT CreateChimpAndEatBanana(void) {
    // 声明并初始化一个 MULTI_QI
    MULTI_QI mqi = { &IID_IApe, 0, 0 };
    HRESULT hr = CoCreateInstanceEx(
        CLSID_Chimp, // 创建一个新的黑猩猩
        0,           // 无聚合
        CLSCTX_ALL, // 任何局部
        0,           // 无显示主机/安全信息
        1,           // 一个接口
        &mqi);       // MULTI_QI 结构数组

    if (SUCCEEDED(hr)) {
        IApe *pApe = reinterpret_cast<IApe*>(mqi, pItf);
        assert(pApe);
        // 使用新对象
        hr = pApe->EatBanana();
        // 释放接口指针
        pApe->Release ();
    }
    return hr;
}

```

如果客户只需要一个接口，并且不需要传递 COSERVERINFO 信息，COM 提供了另一个更为方便的 CoCreateInstanceEx 版本，被称为 CoCreateInstance：<sup>10</sup>

```
HRESULT CoCreateInstance (
    [in] REFCLSID rclsid,           // 哪种对象?
    [in] IUnknown *pUnkOuter,       // 用于聚合
    [in] DWORD dwClsCtx,           // 局部特征
    [in] REFIID riid,              // 哪种接口
    [out, iid_is(riid)] void **ppv); // 存放位置
```

如果使用 CoCreateInstance 的话，前面的例子就会变得非常简单，如下：

```
HRESULT CreateChimpAndEatBanana(void) {
    IApe *pApe = 0;
    HRESULT hr = CoCreateInstance(
        CLSID_Chimp,           // 创建一个新的黑猩猩
        0,                      // 无聚合
        CLSCTX_ALL,             // 任何局部
        IID_IApe,               // 何种 itf
        (void**)&pApe);      // 存放位置

    if (SUCCEEDED(hr)) {
        assert(pApe);
        // 使用新对象
        hr = pApe->EatBanana();
        // 释放接口指针
        pApe->Release();
    }
    return hr;
}
```

上面两个例子的功能完全等价。实际上，CoCreateInstance 的内部实现只是简单地调用 CoCreateInstanceEx：

```
// CoCreateInstance API 实现的伪代码
HRESULT CoCreateInstance(REFCLSID rclsid,
    IUnknown *pUnkOuter, DWORD dwClsCtx,
    REFIID riid, void **ppv) {
    MULTI_QI rgmqi[] = { &riid, 0, 0 };
}
```

<sup>10</sup> 从技术上讲，其实 CoCreateInstance 先出现。CoCreateInstanceEx 是在 WindowsNT4.0 中才被加入进来的，因为很显然，有些开发人员希望在 COM 的激活 API 函数中传递一些安全信息和主机信息。在原先的 CoGetClassObject 原型中，第三个参数是被保留的，所以 NT4.0 把这个保留参数用作 COSERVERINFO。不幸的是，CoCreateInstance 没有未使用的参数，所以便有了 CoCreateInstanceEx。有人可能会说，增加 CoGetClassObject 函数的另一个版本，让它也使用 MULTI\_QI，从而允许一次绑定到多个接口上，这样做将会非常有用，但是到本书写作为止，实际上并不存在这样的 CoGetClassObjectEx 函数。类似的情况也出现在 Imoniker::BindToObject 和 MULTI\_QI 之间。

```

HRESULT hr = CoCreateInstanceEx(rclsid, pUnkOuter,
                                dwClCtx, 0, 1, rgmqi);
*ppv = rgmqi[0].pItf;
return hr;
}

```

尽管使用 `CoCreateInstance` 发出远程激活请求也是有可能的，但是由于 `CoCreateInstance` 不存在 `COSERVERINFO` 参数，所以这使得调用者无法显式地指定远程主机名字。相反，在调用 `CoCreateInstance` 的同时指定 `CLCTX_REMOTE_SERVER` 标志（不再指定其他的标志），这相当于告诉 SCM，使用每个 `CLSID` 的配置信息，以便选择“将被用于激活对象”的主机名字。

图 3.4 显示了 `CoCreateInstanceEx` 的参数如何被映射到 `CoGetClassObject` 和 `IClassFactory::CreateInstance` 的参数。与一般人的想像相反，`CoCreateInstanceEx` 内部并没有调用 `CoGetClassObject`。虽然这两项技术之间并没有逻辑上的不同，但是，在只需创建一个实例的情况下，`CoCreateInstanceEx` 的实现将更有效率，因为它避免了客户与服务器之间额外的来回传递过程，而如果使用 `CoGetClassObject` 的话，这些额外的传递过程不可避免。然而，如果客户需要创建大量的实例的话，那么它可以把类对象指针缓存起来，然后简单地多次调用 `IClassFactory::CreateInstance` 即可。由于 `IClassFactory::CreateInstance` 只是一个方法调用，它并没有涉及到 SCM，所以它多多少少会比调用 `CoCreateInstanceEx` 要快一些。把类对象缓存起来从而避开 `CoCreateInstanceEx`，这样做虽然能更加有效地创建多个对象，但是到底创建多少个对象以上才会更有效率，其阈值取决于主机和网络上的 IPC 和 RPC 性能。

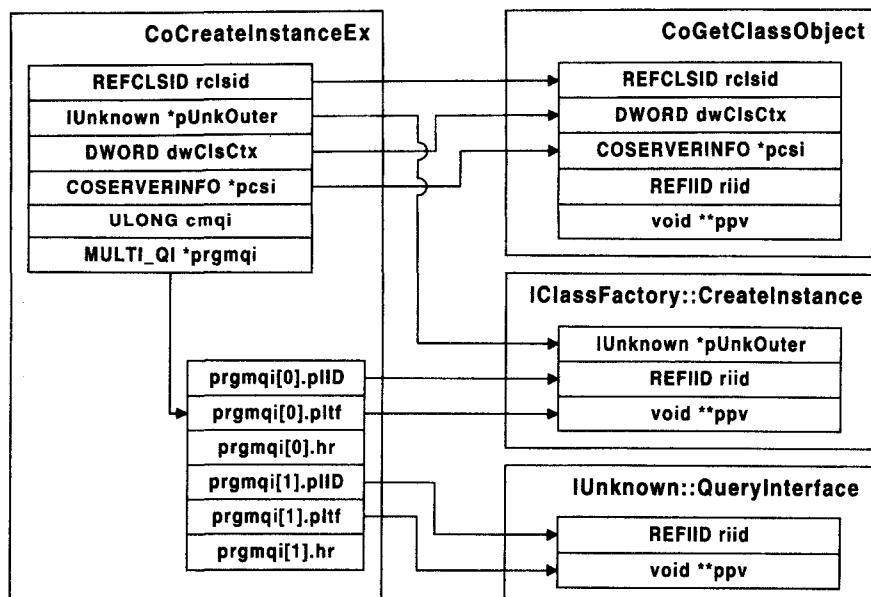


图 3.4 `CoCreateInstanceEx` 和 `CoGetClassObject`

## 3.8 再论接口与实现

前面的客户端激活例子都显式地调用 COM 的激活 API 函数。通常情况下，为了绑定到所期望的对象，客户必须要执行多个步骤（比如，客户先创建某一个类型的对象，然后利用某些查询信息，向它请求指向另一个对象的引用）。为了解除客户和“用来查找或者创建对象”的实际算法之间的依赖关系，COM 提供了一种标准机制，可以把任意的对象名字解析到这些名字所引用的对象上。这种机制用到了定位器对象（locator object），它把绑定算法封装到一个标准的、统一的接口中。这些定位器对象正式名称为名字对象（moniker），它们只是引出了 IMoniker 接口的普通 COM 对象。IMoniker 接口是最复杂的 COM 接口之一；然而，它所暴露的一个方法 BindToObject 对这部分的讨论非常重要：

```
interface IMoniker : IPersistStream {
    HRESULT BindToObject([in] IBindCtx *pbc,
                         [in, unique] IMoniker *pmkToLeft,
                         [in] REFIID riid
                         [out, iid_is(riid)] void **ppv);
    // 为清楚起见其他方法没有列出
}
```

前面章节曾经提到过，接口定义必须非常抽象，并且要足够模糊，以便允许对每个方法的语义有多种解释。BindToObject 的抽象语义是“运行你的查找或者创建对象的算法，一旦对象被创建或者找到之后就返回一个指定类型的接口指针”。当客户在一个名字对象上调用了 BindToObject 之后，它并不清楚名字对象是如何把它的内部状态解析到一个指向特定对象的接口指针。给出三个不同的名字对象，它们可能会使用三个完全不同的算法。这种多态行为使得名字对象的功能非常强大。

客户有多种途径可以获得名字对象。客户可以通过外部的“代理机构”获得名字对象，比如，对某些正在使用中的对象调用方法的结果。客户也可以显式地调用某些 API 函数，来创建特定类型的名字对象。客户只要有一个简单的文本串，把它作为名字对象的“stringified”（字符串化）状态。这后一种情形是最值得感兴趣的，因为它允许应用使用外部的配置文件或者系统注册表，来装入或者保存“基于文本的对象名字(text-based object name)”。如果这项技术已经被作为应用配置的一部分写到公开的文档中，那么系统管理员或者有经验的用户就可以重新配置这个应用，让它使用其他的策略来找到“原始开发者可能（也可能没有）期望存在的对象”。例如，假如有一个支持负载平衡的名字对象，我们只要改变保存在应用配置文件中的名字对象的文本信息，这个名字对象就有可能被重新配置，从而使用不同的策略来选择目标主机。

名字对象文本表述的正式名称是显示名（display name）。IMoniker 接口暴露了一个名为 `GetDisplayName` 的方法，它允许客户向名字对象询问它的显示名。更让人感兴趣的问题是：我们如何把任意一个显示名解析到名字对象上。这项任务还是问题重重，因为客户不能很容易地说出一个显示名对应于哪种名字对象。这正是 `MkParseDisplayName` 的工作，无可非议，它是 COM 中最重要的 API 函数。

`MkParseDisplayName` 可接受任意的显示名，然后把它解析为一个名字对象：

```
HRESULT MkParseDisplayName(
    [in] IBindCtx *pbc,           // 绑定信息
    [in, string] const OLECHAR *pwszName, // 对象名
    [out] ULONG *pcchEaten,        // 错误过程
    [out] IMoniker **ppmk);       // 结果名字对象
```

名字对象的名字空间是可以扩充的，因此可以支持新的名字对象类型。`MkParseDisplayName` 使用的最上层解析器检查显示名的前缀，然后试着把前缀与已经被注册的 ProgID 进行匹配，这个 ProgID 指出了该显示名与哪种类型的名字对象相对应。如果找到了这样的匹配，那么这种类型的名字对象新实例就会被创建，然后把显示名交给这个名字对象，由它作进一步的解析。因为名字对象是层次状的，而且支持组合，所以结果得到的名字对象有可能是两个或者更多个名字对象的组合。这些都是实现细节，与客户没有关系。客户只是简单地使用结果名字对象的 `IMoniker` 接口指针（可能指向复合名字对象，也可能不是），来找到问题中的目标对象。

前面曾经提到过，进入到 COM 类的初始进入点需要经过它的类对象。为了与一个类对象连接起来，我们需要一个类型为 Class Moniker（类名字对象）的名字对象。类名字对象是内置的名字对象类型，由 COM 直接提供。类名字对象记录一个 CLSID 作为它的内部状态，我们可以利用显式的 COM API 函数 `CreateClassMoniker` 来创建类名字对象，函数原型如下：

```
HRESULT CreateClassMoniker([in] REFCLSID rclsid,
                           [out] IMoniker **ppmk);
```

或者把类名字对象的显示名传给 `MkParseDisplayName`:<sup>11</sup>

`clsid:571F1680-CC83-11d0-8C48-0080C73925BA:`

注意，前缀“`clsid`”正是类名字对象的 ProgID。

下面的代码说明了使用 `MkParseDisplayName` 来创建一个类名字对象，然后用它来

<sup>11</sup> 虽然使用 `MkParseDisplayName` 将会损失一些效率，但是这样做具有更大的灵活性。正如前面所说明的，显示名可以从文件中读出来，或者从用户界面中获取。Microsoft 的 Internet Explorer 是一个非常好的应用例子，它允许用户输入任意的对象名字（URL），然后它使用扩展的 API 函数 `MkParseDisplayNameEx`，把该名字解析为名字对象。

连接到 Gorilla 类对象：

```

HRESULT GetGorillaClass(IApeClass * &rpgc) {
    rpgc = 0;
    // 声明 Gorilla 的 CLSID 为一个显示名
    const OLECHAR pwsz[] =
        OLESTR("clsid:571F1680-CC83-11d0-8C48-0080C73925BA");
    // 为绑定和解析名字对象创建
    // 一个新的绑定环境
    IBindCtx *pbc = 0;
    HRESULT hr = CreateBindCtx(0, &pbc);
    if (SUCCEEDED(hr)) {
        ULONG cchEaten;
        IMoniker *pmk = 0;
        // 请求 COM 将显示名转换为名字对象
        hr = MkParseDisplayName(pbc, pwsz, &cchEaten, &pmk);
        if (SUCCEEDED(hr)) {
            // 请求名字对象寻找或创建
            // 它指向的对象
            hr = pmk->BindToObject(pbc, 0, IID_IApeClass,
                (void**)&rpgc);
        }
        // 现在我们有了指向所需对象的指针，因此释放
        // 名字对象和绑定环境
        pmk->Release ();
    }
    pbc->Release ();
}
return hr;
}

```

绑定环境（binding context）同时被传给 `MkParseDisplayName` 和 `IMoniker::BindToObject`，它实际上只是一个辅助对象，允许我们将一些辅助参数传递给名字对象的解析和绑定机制。对于这个简单的例子而言，我们所需要的只是一个新的绑定环境对象，用来当作一个存储空间（placeholder），我们只要调用 COM API 函数 `CreateBindCtx` 就可以得到这样的对象。<sup>12</sup>

Windows NT 4.0 引入了另一个 API 函数，从而可以简化对 `MkParseDisplayName` 和 `IMoniker::BindToObject` 的调用：

```

HRESULT CoGetObject (
    [in, string] const OLECHAR *pszName,
    [in, unique] BIND_OPTS *pBindOptions,
    [in] REFIID riid,
    [out, iid_is(riid)] void **ppv);

```

<sup>12</sup> 绑定环境被复合名字对象（composite moniker）用来对解析和绑定操作进行优化。绑定环境也允许客户指定 CLSCTX 标记和 COSERVERINFO，但是当前类名字对象的实现版本忽略这两个属性。相反，类名字对象假定它将与另一个“指向 IclassActivator 接口的实现”的名字对象组合起来，如此一来便可以提供更大的灵活性。

这个 API 函数的实现过程如下：

```
// 摘自 OLE32.DLL 的伪代码
HRESULT CoGetObject(const OLECHAR *pszName, BIND_OPTS *pOpt,
                     REFIID riid, void **ppv) {
    // 准备失败
    *ppv = 0;
    // 创建绑定环境
    IBindCtx *pbc = 0;
    HRESULT hr = CreateBindCtx(0, &pbc);
    if (SUCCEEDED(hr)) {
        // 如已提供, 设置绑定选项
        if (pOpt)
            hr = pbc->SetBindOptions(pOpt);
        if (SUCCEEDED(hr)) {
            // 将显示名转换为名字对象
            ULONG cch;
            IMoniker *pmk = 0;
            hr = MkParseDisplayName(pbc, pszName, &cch, &pmk);
            if (SUCCEEDED(hr)) {
                // 要求名字对象绑定已命名的对象
                hr = pmk->BindToObject(pbc, 0, riid, ppv);
                pmk->Release();
            }
        }
        pbc->Release();
    }
    return hr;
}
```

有了这个函数之后，现在创建新的 Gorilla 对象将非常简单，只需找到类对象，然后调用 CreateInstance 方法即可，示例代码如下：

```
HRESULT CreateAGorillaAndEatBanana() {
    IClassFactory *pcf = 0;
    // 声明 Gorilla 的 CLSID 为显示名
    const OLECHAR pwsz[] =
        OLESTR("clsid: 571F1680-CC83-11d0-8C48-0080C73925BA:");
    // 通过 Gorilla 的类名字对象寻找类对象
    HRESULT hr = CoGetObject(pwsz, 0, IID_IClassFactory,
                           (void**) &pcf),
    if (SUCCEEDED(hr)) {
        IApe *pApe = 0;
        // 使用类对象创建 Gorilla
        hr = pcf->CreateInstance(0, IID_IApe, (void**) &pApe);
        if (SUCCEEDED(hr)) {
            // 让新的 Gorilla 吃香蕉
            hr = pApe->EatBanana();
            pApe->Release();
        }
    }
}
```

```

    }
    pcf->Release ();
    returnhr
}
}

```

图 3.5 显示了每个操作创建或者找到哪个对象。

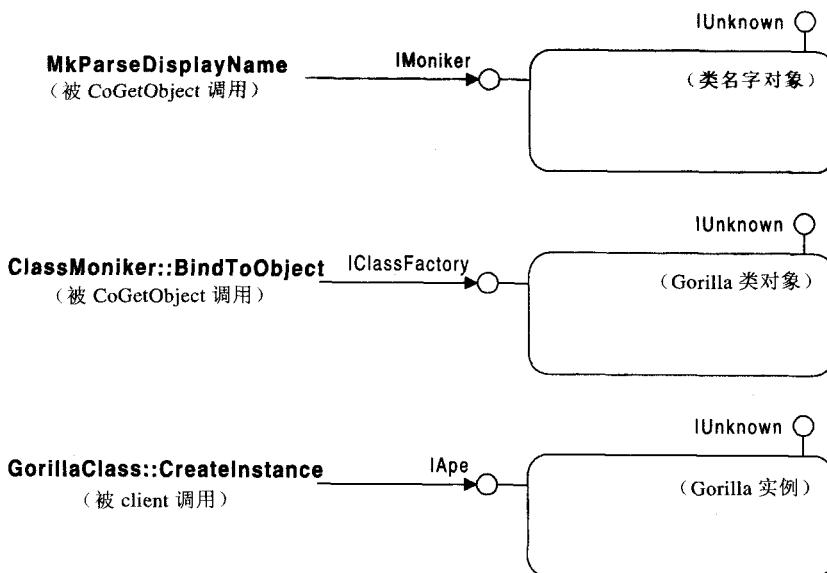


图 3.5 使用类名字对象来激活对象

Visual Basic 通过它的内置函数 GetObject 暴露了 CoGetObject API 函数的功能。下面的 Visual Basic 代码也可以创建一个新的 Gorilla 对象，然后请它吃香蕉：

```

Sub CreateGorillaAndEatBanana()
    Dim gc as IapeClass
    Dim ape as Iape
    Dim sz as String
    sz = "clsid:571F1680-CC83-11d0-8C48-0080C73925BA:"

    ' 获取类对象
    Set gc = GetObject(sz)
    ' 要求类对象创建新的 Gorilla
    Set ape = gc.CreateApe()
    ' 让 Gorilla 吃香蕉
    ape.EatBanana
End Sub

```

注意，这个函数的 Visual Basic 版本使用 IApeClass 接口来实例化对象。原因在于，由于 Visual Basic 语言的限制，它无法使用 IClassFactory 接口。

## 3.9 名字对象和组合

名字对象常常可以由其他的名字对象组合得到，可以在路径的文本描述的基础上遍历操纵整个对象层次结构。为了使这种遍历操纵访问更为容易，COM 提供了一个标准的名字对象实现，当它被组合到另一个名字对象的右边时，它就请对象（对应于左边的名字对象）绑定一个指向层次结构中另一个对象的引用。这个名字对象被称为单项名字对象，它使用对象的 `IOleItemContainer` 接口，把对象名字解析为接口指针。

下面的显示名说明了单项名字对象与类名字对象组合在一起的用法：

```
clsid:571F1680-CC83-11d0-8C48-0080C73925BA:!Ursus
```

注意，上面的显示名利用“！”把类名字对象的显示名与单项名“Ursus”分割开来。在解析的时候，`MkParseDisplayName` 将首先把前缀“`clsid`”当作一个 `ProgID`，与类名字对象的实现进行联系。然后 `MkParseDisplayName` 请类名字对象尽可能多地解析剩余的字符串。这意味着，在类名字对象把 `GUID` 从字符串中解析出来之后，剩下的字符串断仍需要进一步被解析：

```
!Ursus
```

因为只有在该名字左边的名字对象所命名的对象的范围内，这个名字才有意义，所以 `MkParseDisplayName` 将首先绑定最左边的名字对象（类名字对象），然后请它所命名的对象（`Gorilla` 类对象）来解析剩余的字符串。为了提供这种解析显示名的支持，COM 定义了标准的接口 `IParseDisplayName`：

```
[ object, uuid(0000011a-0000-0000-C000-00000000046) ]
interface IParseDisplayName : IUnknown {
    // 将显示名转化为名字对象
    HRESULT ParseDisplayName(
        [in, unique] IBindCtx *pbc,
        [in] LPOLESTR pszDisplayName,
        [out] ULONG *pchEaten,
        [out] IMoniker **ppmkOut
    );
}
```

对于这个例子中用到的显示名来说，`Gorilla` 类对象有必要实现 `IParseDisplayName`，

并且把字符串“!Ursus”转换为一个名字对象，`MkParseDisplayName` 将把它组合到类名字对象的右边。因为通常期望的是一个单项名字对象，所有下面的实现可以满足要求：

```
STDMETHODIMP GorillaClass::ParseDisplayName(IBindCtx *pbc,
                                             LPOLESTR pszDisplayName, ULONG *pchEaten,
                                             IMoniker **ppmkOut) {
    // 使用显式 API 函数创建一个单项名字对象
    HRESULT hr = CreateItemMoniker(OLESTR("!"),
                                    pszDisplayName + 1,
                                    ppmkOut);

    // 指明解析了多少个字符
    if (SUCCEEDED(hr))
        *pchEaten = wcslen(pszDisplayName);
    else
        *pchEaten = 0;
    return hr;
}
```

注意，这个例子并没有试图检验它当前正在解析的名字是否有效。它只是简单地剔除字符串头上的“!”，并且根据剩余的显示名创建一个新的单项名字对象。

一旦这两个名字对象已经被解析完成，`MkParseDisplayName` 将利用通用复合名字对象（generic composite moniker）把这两个名字对象组合起来。通用复合名字对象只是简单地把两个名字对象放在一起。通用复合名字对象的 `BindToObject` 实现非常简单，只是绑定右边的名字对象，同时把左边名字对象的指针作为 `pmkToLeft` 参数传给它。下面的伪代码说明了这一点：

```
// 摘自 OLE32.DLL 的伪代码
STDMETHODIMP GenericComposite::BindToObject(IBindCtx *pbc,
                                              IMoniker *pmkToLeft,
                                              REFIID riid, void **ppv) {
    return m_pmkRight->BindToObject(pbc, m_pmkLeft, riid, ppv);
}
```

上面的伪实现代码说明：右边的名字对象只有在它左边的名字对象的范围内才有意义。在本例子中“类名字对象！单项名字对象”形式的情况下，单项名字对象在绑定时，将会接收到作为 `pmkToLeft` 参数的类名字对象。

正如前面已经说明的那样，单项名字对象使用 `IOleItemContainer` 接口来绑定接口指针。下面是伪代码形式的单项名字对象的 `BindToObject` 实现：

```
// 摘自 OLE32.DLL 的伪代码
STDMETHODIMP ItemMoniker::BindToObject(
    IBindCtx *pbc, IMoniker *pmkToLeft,
    REFIID riid, void **ppv) {
```

```

// 假定失败
*ppv = 0;
if (pmkToLeft == 0) // 需要一个作用域
    return E_INVALIDARG;
// 首先绑定右边
IOleItemContainer *poic = 0;
HRESULT hr = pmkToLeft->BindToObject(pbc, 0,
                                         IID_IOleItemContainer, (void**)&poic);
if (SUCCEEDED(hr)) {
// 在绑定环境中缓存已绑定对象
    pbc->RegisterObjectBound (poic);
// 从绑定环境获取绑定速度
    DWORD dwBindSpeed = this->MyGetSpeedFromCtx(pbc);
// 要求对象给出已命名的子对象
    hr = poic->GetObject(m_pszItem, dwBindSpeed, pbc,
                           riid, ppv);
    poic->Release();
}
}

```

这段实现代码隐含着下面的代码：

```

HRESULT GetUrsus(IApe **&rpApe) {
    const OLECHAR pwsz[] =
OLESTR("clsid:571F1680-CC83-11d0-8C48-0080C73925BA:!Ursus");
    return CoGetObject(pwsz, 0, IID_IApe, (void**)&rpApe);
}

```

等同于以下的代码：

```

HRESULT GetUrsus(IApe **&rpApe) {
    IOleItemContainer *poic = 0;
    HRESULT hr = CoGetClassObject (CLSID_Gorilla, CLSCTX_ALL,
                                   0, IID_IOleItemContainer, (void**)&poic);
    if (SUCCEEDED(hr)) {
        hr = poic->GetObject(OLESTR("Ursus"), BINDSPEED_INFINITE,
                               0, IID_IApe, (void**)&rpApe);
        poic->Release ();
    }
    return hr;
}

```

注意，由于使用 `CoGetObject` 而带来的间接性，因此客户只要从配置文件或者从注册表的键中读入不同的显示名，就可以改变绑定策略。

## 3.10 名字对象和永久性

如果不讨论文件名字对象 (File Moniker)，那么对名字对象的讨论就不会是完整的。前面曾经提到过，COM 支持三种激活方式：绑定到类对象、绑定到新的类实例，以及绑定到保存在文件中的永久对象。本章已经详细地探讨了前面两种激活方式。第三种激活方式以 COM API 函数 `CoGetInstanceFromFile` 为基础，函数原型如下：

```
HRESULT CoGetInstanceFromFile(
    [in, unique] COSERVERINFO *pcsi,           // 主机/安全信息
    [in, unique] CLSID      *pClSID,           // 显式的 CLSID
    [in, unique] IUnknown  *punkOuter,          // 用于聚合
    [in]        DWORD     dwClsCtx,            // 局部特征
    [in]        DWORD     grfMode,              // 文件打开模式
    [in]        OLECHAR   *pwszName,             // 对象文件名
    [in]        DWORD     cmqi,                 // 多少个接口？
    [out, size_is(cmqi)] MULTI_QI  *prgmq,       // 存的位置
);
```

这个函数接受一个文件名作为输入，而该文件名指向对象的永久状态。<sup>13</sup> `CoGetInstanceFromFile` 首先确保这个对象运行起来，然后返回一个或者多个接口指针，指向这个被[重新]激活的对象。为了完成这项任务，`CoGetInstanceFromFile` 首先需要确定对象的 CLSID。有两个理由需要获得这个 CLSID。第一，如果对象还没有运行起来，那么 COM 将需要用这个 CLSID 来创建一个新的实例，然后利用永久映像 (persistent image) 对它进行初始化。第二，如果调用者没有显式地指定“该激活调用将被转发出去”的机器名字，那么 COM 使用 CLSID 来决定将在哪台机器上激活对象。<sup>14</sup>

如果调用者没有显式地传递 CLSID，那么 `CoGetInstanceFromFile` 调用 COM API 函数 `GetClassFile`，利用文件信息获得 CLSID。`GetClassFile` 函数原型如下：

```
HRESULT GetClassFile([in, string] OLECHAR *pwszFileName,
                     [out] CLSID  *pclsid);
```

`GetClassFile` 使用文件的头信息和注册表信息，来决定文件中包含的是哪种对象。

<sup>13</sup> 这个 API 函数的中一个版本 `CoGetInstanceFromIStorage`，接受一个指向层次状存储介质的指针作为输入，而不是文件名。

<sup>14</sup> 除了通常使用 `CoGetClassObject/CoCreateInstanceEx` 来把 CLSID 对激活请求转发到其他的机器上之外，`CoGetInstanceFromFile` 也可以使用文件的 UNC 主机名字，把激活请求转发到文件所在的机器上。在 COM 规范中，这种激活模式被称为“*AtBits*”激活，我们可以使用第 6 章描述的“*ActivateAtStorage*”（在存储中激活）注册表设置来指定这种模式。

一旦类 (CLSID) 和机器名字都已经被确定下来了，COM 检查目标主机上的 ROT (Running Object Table，运行对象表)，以便确定对象是否已经被激活了。ROT 是 SCM 的一个设施，它把任意的名字对象映射到本地机器上正在运行的对象实例上。永久对象应该在装载的时候把自己注册到本地 ROT 中。为了把永久对象的文件名表现为一个名字对象，COM 提供了一种标准的名字对象类型，称为文件名字对象，它把文件名包装到 IMoniker 接口后面。创建文件名字对象可以有两种途径，一种是把文件名传递给 MkParseDisplayName 调用，另一种则是调用专用的 API 函数 CreateFileMoniker：

```
HRESULT CreateFileMoniker(
    [in, string] const OLECHAR *pszFileName,
    [out] IMoniker **ppmk);
```

如果永久对象已经把它的文件名字对象注册到 ROT 中了，那么 CoGetInstanceFromFile 只是简单地返回一个指针，指向 ROT 中这个已经在运行的对象。如果在 ROT 中没有找到永久对象，COM 将创建该文件的类 (CLSID) 的新实例，然后利用对象实例的 IPersistFile::Load 方法，利用永久映像对它进行初始化。IPersistFile::Load 方法的声明如下：

```
[ object, uuid(0000010b-0000-0000-C000-000000000046) ]
interface IPersistFile : IPersist {
// CoGetInstanceFromFile 调用初始化对象
    HRESULT Load (
        [in, string] const OLECHAR * pszFileName,
        [in] DWORD grfMode
    );
// 为清楚起见，其他方法未列出
}
```

从文件中装载永久状态，以及把对象本身注册到本地 ROT 中，以确保同一时刻每个文件只有一个实例正在运行，这都是对象实现的责任。举例如下：

```
STDMETHODIMP Gorilla::Load(const OLECHAR *pszFileName,
                           DWORD grfMode) {
// 读入对象状态
    HRESULT hr = this->MyReadStateFromFile(pszFile, grfMode);
    if (FAILED(hr)) return hr;
// 从 SCM 获取 ROT 指针
    IRunningObjectTable *prot=0;
    hr = GetRunningObjectTable(0, &prot);
    if (SUCCEEDED(hr)) {
// 创建一个在 ROT 中注册的文件名字对象
        IMoniker *pmk = 0;
        hr = CreateFileMoniker (pszFileName, &pmk),
        if (SUCCEEDED(hr)) {
// 在 ROT 中注册
        hr = prot->Register(0, this, pmk, &m_dwReg);
```

```

    pmk->Release ();
}
prot->Release ();
}
return hr,
}
}

```

在 CoGetInstanceFromFile 执行过程中, SCM 将会调用新创建实例的 IPersistFile::Load 方法。上面的例子使用 COM API 函数 GetRunningObjectTable, 来获得进入 SCM 的 IRunningObjectTable 接口指针。然后它用这个接口, 把它的名字对象注册到 ROT 中, 因此, 以后再用同样的文件名调用 CoGetInstanceFromFile 将不会创建新的对象, 而是返回指向这个对象的引用。<sup>15</sup>

文件名字对象之所以存在有两个理由。一个理由是, 允许对象把自身注册到 ROT 中, 以便 CoGetInstanceFromFile 能够找到它们。第二个理由是, 对客户而言, 可把 CoGetInstanceFromFile 隐藏到 IMoniker 接口后面。文件名字对象的 BindToObject 实现只是简单地调用 CoGetInstanceFromFile:

```

// 摘自 OLE32.DLL 的伪代码
STDMETHODIMP FileMoniker:: BindToObject (IBindCtx *pbc,
                                             IMoniker *pmkToLeft,
                                             REFIID riid, void **ppv) {
//假定失败
*ppv = 0;
HRESULT hr = E_FAIL;
if (pmkToLeft == 0) { // 左边没有名字对象
    MULTI_QI mqi = { &riid, 0, 0 };
    COSERVERINFO *pcsi;
    DWORD grfMode;
    DWORD dwClsCtx;
//这三个参数是 BindCtx 的属性
    this->MyGetFromBindCtx(pbc, &pcsi, &grfMode, &dwClsCtx);
    hr = CoGetInstanceFromFile(pcsi, 0, 0, dwClsCtx,
                               grfMode, this->m_pszFileName,
                               1, &mqi);
    if (SUCCEEDED(hr))
        *ppv = mqi.pItf;
}
else { //左边有一个名字对象
    // 要求 IClassActivator 或
    // IClassFactory 的对象
}
return hr;
}

```

<sup>15</sup> 技术上讲, ROT 不是机器范围内(machine-wide)的表, 而是窗口站范围内(Winstation-wide)的表, 这意味着, 在缺省情况下, 并不是所有登录会话都可以访问这个对象。为了保证这个对象对于所有可能的客户都是可见的, 对象的调用 IRunningObjectTable::Register 时应该指定 ROTFLAGS-ALLOWANYCLIENT 标志。

有了文件名字对象的行为之后，下面调用 `CoGetInstanceFromFile` 的函数：

```
HRESULT GetCornelJus(IApe * &rpApe) {
    OLECHAR *pwszObject =
        OLESTR("\\\\server\\\\public\\\\cornelius.chmp");
    MULTI_QI mqJ = { &IID_IApe, 0, 0 };
    HRESULT hr = CoGetInstanceFromFile(0, 0, 0, CLSCTX_SERVER,
                                      STGM_READWRITE, pwszObject, 1, &mqi);
    if (SUCCEEDED(hr))
        rpApe = mqJ.pItf;
    else
        rpApe = 0;
    return hr;
}
```

就可以被简化为以下调用 `CoGetObject` 的函数：

```
HRESULT GetCornelJus(IApe * &rpApe) {
    OLECHAR *pwszObject =
        OLESTR("\\\\server\\\\public\\\\cornelius.chmp");
    return CoGetObject(pwszObject, 0, IID_IApe, (void**)&rpApe);
```

如同前面使用类名字对象的例子一样，`CoGetObject` 带来的间接性允许客户指定任意复杂的激活策略，而无须改变一行代码。

## 3.11 服务器生命周期

前面章节已经讲述了 COM 如何自动地装载 DLL，以便把对象实现带入到客户程序的地址空间中。剩下尚未讨论的是这些 DLL 如何以及何时被卸载。一般情况下，服务器 DLL 可以防止过早卸载，但是只有客户才能够选择什么时候 DLL 被真正释放掉。如果客户希望释放空闲的 DLL，那么它可以调用 COM API 函数 `CoFreeUnusedLibraries`：

```
void CoFreeUnusedLibraries(void);
```

客户通常在空闲的时候调用这个函数，以便回收其地址空间中的垃圾。当 `CoFreeUnusedLibraries` 被调用的时候，COM 询问每一个已经被装载进来的 DLL，以便找到不再需要的 DLL。为了做到这一点，COM 调用每个 DLL 的 `DllCanUnloadNow` 函数，它要求每个 DLL 必须显式地引出这个函数。

每个服务器 DLL 引出的 `DllCanUnloadNow` 函数必须具有下面的原型：

```
HRESULT DllCanUnloadNow(void);
```

如果 DLL 希望自己被释放的话，那么它就返回 S\_OK。如果 DLL 希望继续留在内存中，那么它返回 S\_FALSE。只要还存在未用完的接口指针指向服务器 DLL 中的对象，那么它就必须继续留在内存中。这意味着，DLL 有必要记录下所有未用完的对象引用。为了简化其实现过程，大多数 DLL 维护一个专门的锁计数变量，并且用两个函数来自动增加和减小锁计数：

```
LONG g_cLocks = 0;
void LockModule(void) { InterlockedIncrement(&g_cLocks); }
void UnlockModule(void) { InterlockedDecrement(&g_cLocks); }
```

有了这些函数之后，`DllCanUnloadNow` 的实现就特别简单：

```
STDAPI DllCanUnloadNow(void)
{ return g_cLocks == 0 ? S_OK : S_FALSE; }
```

剩下的事情就是在适当的时候调用 `LockModule` 和 `UnlockModule` 函数。

有两股基本力量可以让服务器 DLL 保留在内存中：指向类实例和类对象的未完结引用，以及未完结的 `IClassFactory::LockServer` 调用。为类实例和类对象加上 `DllCanUnloadNow` 支持非常直截了当，基于堆（heap-based）的对象（比如类实例）只要简单地在第一个 `AddRef` 调用时增加锁计数即可。如下：

```
STDMETHODIMP_(ULONG) Chimp::AddRef(void) {
    if (m_cRef == 0)
        LockModule();
    return InterlockedIncrement(&m_cRef);
}
```

然后在最后一个 `Release` 调用时减小锁计数：

```
STDMETHODIMP_(ULONG) Chimp::Release(void) {
    LONG res = InterlockedDecrement(&m_cRef);
    if (res == 0) {
        delete this;
        UnlockModule();
    }
    return res,
}
```

因为不基于堆的对象（比如类对象）并没有记录引用计数，所以每个 `AddRef` 和 `Release` 调用都应该增加或者减小锁计数：

```
STDMETHODIMP_(ULONG) ChimpClass::AddRef(void) {
    LockModule();
    return 2,
```

```

    }

STDMETHODIMP_(ULONG) ChimpClass::Release(void) {
    UnlockModule();
    return 1;
}

```

实现了 IClassFactory 接口的类对象应该在 IClassFactory::LockServer 内部调整服务器的锁计数值：

```

STDMETHODIMP_(ULONG) ChimpClass::AddRef(void) {
    LockModule ();
    return 2;
}

STDMETHODIMP_(ULONG) ChimpClass::Release(void) {
    UnlockModule();
    return 1;
}

```

正如第 6 章将要讨论的，IClassFactory::LockServer 存在的主要目的是针对进程外服务器，但是在进程内服务器中，它的实现可以非常简单。

值得注意的是，在 CoFreeUnusedLibraries/DllCanUnloadNow 协议中存在一个固有的竞争条件(race condition)。有可能发生这样的情况：一个线程在 DLL 引出的最后一个实例上执行最后的 Release 操作，同时第二个线程执行 CoFreeUnusedLibraries 过程。COM 考虑到了每一种可能的预防措施，以避免这种情形。特别地，在 Windows NT 4.0 Service Pack 2 下 COM 的实现版本加入了特别的设施来解决这种潜在的竞争条件。Service Pack 2 版本的 COM 库检测服务器 DLL 是否被多个线程访问，然后在 CoFreeUnusedLibraries 内部不是立即卸载 DLL，而是把 DLL 挂到一个 DLL 队列上，这个队列上的 DLL 都是要被释放的。然后 COM 将会等待一段不定长的时间，之后再次调用 CoFreeUnusedLibraries，允许这些空闲的服务器 DLL 被释放掉，从而保证了不会再有剩余的 Release 仍在被执行。<sup>16</sup>这意味着，在多线程环境中，把 DLL 从客户进程中卸载出去所花的时间要比预期的更长一些。

## 3.12 类和 IDL

正如本章前面已经说明的那样，COM 把接口和类看作不同的实体。因此，COM 类

<sup>16</sup> 有可能 Windows NT 5.0 会提供更好的支持，以保证 DLL 被迅速而安全地释放掉。更多的细节请参照 SDK 文档。

(与 COM 接口一样)也应该被定义在 IDL 中,以便对于“服务器可能引出的具体数据类型”提供独立于程序语言的描述。COM 类的 IDL 定义包含一组接口的列表,除非有重大的灾难发生,否则该类的实例将引出这些接口。举例如下:

```
[ uuid (753A8A7D-A7FF-11d0-8C30-0080C73925BA) ]
coclass Gorilla {
    interface IApe;
    interface IWarrior;
}
```

IDL 的 coclass 定义总是出现在库定义的上下文环境内部。在 IDL 中,库定义可以用来把一组数据类型(比如 interface、coclass、typedef)组织到一个逻辑单元或者名字空间中。凡是出现在 IDL 库定义上下文环境中的所有数据类型都将被符号化(tokenized),并放到结果类型库(type library)中。在 Visual Basic 和 Java 环境中,用类型库来代替 IDL 文件。

一个 IDL 文件至多只能有一个 library 语句,在库定义内部定义的或者用到的所有数据类型都将出现在最终得到的类型库中。举例如下:

```
// apes.idl /////////////////////////////////
// 引入 ape 接口的 IDL 定义
import "apeitfs.idl";
[
    uuid(753A8AS0-A7FF-11d0-8C30-0080C73925BA), // LIBID
    version(1.0), //库的版本号
    lcid(9), //库的地域 ID(英语)
    helpstring("Library of the Apes") //库名
]
library ApeLib
{
    importlib("stdole32.tlb"); // 引入标准定义
    [uuid(753ASA7D-A7FF-11d0-8C30-0080C73925BA) ]
    coclass Gorilla {
        [default] interface IApe;
        interface IWarrior;
    }
    [uuid(753A8A7E-A7FF-11d0-8C30-0080C73925BA) ]
    coclass Chimpanzee {
        [default] interface IApe;
        interface IEgghead;
    }
    [uuid(753A8A7F-A7FF-11d0-8C30-0080C73925BA) ]
    coclass Orangutan {
        [default] interface IApe;
        interface IKeeperOfTheFaith;
    }
}
```

[default] 属性说明了哪个接口最能够表达这个类的本质类型。在所有能够识别这个属性的语言中，[default] 属性允许程序员直接使用 COM 的 coclass 名字来声明对象引用，例如：

```
Dim ursus as Gorilla
```

根据 Gorilla 的 IDL 定义，这条语句等价于

```
Dim ursus as IApe
```

因为 IApe 是 Gorilla 类的缺省接口。在这两种情况下，程序员都可以对 ursus 变量调用 EatBanana 和 SwingFromTree 方法。如果 COM 类没有指定 [default] 属性，那么隐含着 coclass 定义中第一个接口具有 [default] 属性。

有了前面给出的 IDL 库定义之后，结果头文件 apes.h（译注 2）将使用 C 预处理器来包含 apesitfs.h 文件。apesitfs.h 包含四个 COM 接口 IApe、IWarrior、IKeeperOfTheFaith 和 IEgghead 的抽象类定义。文件 apes.h 也将包含每个类的 GUID 的声明：

```
extern "C" const CLSID CLSID_Gorilla;
extern "C" const CLSID CLSID_Chimpanzee;
extern "C" const CLSID CLSID_Orangutan;
```

对应的 apes\_i.c 文件将包含这些 CLSID 的定义。结果产生的类型库 apes.tlb，将包含每个接口和类的描述，使得 Visual Basic 程序员可以编写下面这样的代码：

```
Dim ape As Iape
Dim warrior as Iwarrior
Set ape = New Gorilla ' 请求 COM 创建一个新的 Gorilla
Set warrior = ape
```

同样地，Java 版本的代码如下：

```
IApe ape;
IWarrior warrior;
ape = new Gorilla(); // 缺省时无需转换
warrior = (IWarrior)ape;
```

这些代码片段告诉底层的虚拟机使用 Gorilla 的 CLSID，以指示 CoCreateInstanceEx 创建哪种类型的对象。

在前面的 IDL 中，接口 IApe、IWarrior、IKeeperOfTheFaith 和 IEgghead 都被库定义引用了，这使得它们的定义都将出现在最终产生的类型库中，尽管它们是在库定义范围之外被定义的。事实上，任何被使用到的数据类型，无论是作为这些接口的参数还是

---

译注 2：可以由 IDL 编译器产生，比如 MIDL.EXE。

作为基接口，它们都将出现在最终产生的类型库中。把一个实现的 library 语句定义在独立的 IDL 文件中，然后它从另一个外部 IDL 文件引入任何它所需要的接口定义，而这个外部 IDL 文件只包含接口定义，这是一种很好的做法。对于包含多个 IDL 文件的大工程，应该强制使用这种做法，因为包含库定义的 IDL 文件要想引入另一个包含库定义的 IDL 文件将会导致出错。把库定义分散到不同的 IDL 文件中，于是一个库用到的接口也可以被引入到其他的工程中，而无需担心多个库定义冲突的问题。如果不使用这种做法，那么要想从一个包含库定义的 IDL 文件中引入接口定义，唯一的办法就是使用 importlib 指示符，引入已经被产生的类型库。举例如下：

```
// humans.idl ///////////
// apeitfs.idl 没有库语句，引入
import "apeitfs.idl";
[
    uuid(753A8AC9-A7FF-11d0-8C30-0080C73925BA),
    version(1.0), lcid(9), helpstring("Humans that need apes")
]
library HumanLib {
    importlib("stdole32.tlb"); //引入标准定义
// Dogs.idl 有库定义，引入
// 相应的类型库
    importlib("dogs.tlb");

    [uuid (753A8AD1-ATFF-11d0-8C30-0080C73925BA) ]
coclass DogApe {
    interface IDog;
    interface IApe;
}
```

简单的工程通常只使用一个 IDL 文件来定义该工程所引出的接口和类。对于简单的接口，这样做当然是合理的，因为结果产生的类型库将包含原来 IDL 定义中一对一的映射结果，这样类型库用户使用 importlib 而不会丢失信息。不幸的是，对于复杂的接口，在结果类型库中，原来 IDL 中的许多信息都丢失了，使用 importlib 也不能如预期的那样工作。将来版本的 MIDL 编译器也许能够产生“包含原始 IDL 所有信息”的类型库。

### 3.13 类模仿 (class emulation)

通常有这样的情况：类的实现者希望对现有的类配置新的版本，以便能够修补缺陷或者增强功能。为这些新的实现赋予新的 CLSID，是非常有用的，这样可让客户显式地指明它要求哪个版本。例如，我们考虑配置一个类的第二个版本时的情况。如果使用一个新的 CLSID 来标识新的类（比如 CLSID\_Chimp2），那么“希望使用新版本类”的客

户将会在激活时候使用新的 CLSID:

```
//新客户
IApe *pApe = 0;
hr = CoCreateInstance(CLSID_Chimp2, 0, CLSCTX_ALL,
                      IID_Ape, (void**)&pApe);
```

使用第二个 CLSID 可以保证“客户不会意外地获得旧版本的 Chimp 类”。然而，老的客户仍然使用以前的 CLSID 来发出激活请求：

```
//老客户
IApe *pApe = 0;
hr = CoCreateInstance(CLSID_Chimp, 0, CLSCTX_ALL,
                      IID_Ape, (void**)&pApe)
```

为了继续支持这些老客户，Chimp 实现者需要在注册表中保留原来的 CLSID 信息，以便能够满足这些激活请求。如果这个类的语义已经发生变化了，那么原来的服务器也将需要保留下，以便满足这些客户。然而，常见的情形是，语义只是简单地被扩展了（并不是发生了大的变化）。在这种情况下，应该优先考虑简单地变更原来激活请求的路线，使它创建新类的实例。

为了允许新版本类的实现者能够透明地满足其他 CLSID 的激活请求，COM 支持类模仿（class emulation）的概念。类模仿允许一个组件的实现者指定“老的 CLSID 已经被新的 CLSID 所替代，并且这个新的 CLSID 可以模仿原来的类的语义”。这使得“仍然使用原来的 CLSID 发出激活调用”的老客户也能够得到新的、被更新之后的类的实例。为了指明某一个类有了新的替代版本，COM 提供了下面的 API 函数：

```
HRESULT CoTreatAsClass ( [in] REFLCLSID rclsidOld,
                         [in] REFLCLSID rclsidNew);
```

假定 Chimp2 是 Chimp 类的新版本，于是下面的代码通知 COM，把对 Chimp 的激活请求改换成对 Chimp2 的激活请求：

```
// 使 Chimp 的激活调用激活 Chimp2
HRESULT hr = CoTreatAsClass(CLSID_Chimp, CLSID_Chimp2);
```

这个 API 函数在注册表中插入下面的键：

```
[HKCR\CLSID\{ CLSID_Chimp }\TreatAs]17
@= {CLSID_Chimp2}
```

用 CLSID\_NULL 作为第二个参数调用 CoTreatAsClass 函数则会删除 TreatAs 设置：

<sup>17</sup> 注意 CLSID\_Chimp 和 CLSID\_Chimp2 是规范形式 32 位 GUID 的缩写形式。

```
//使 Chimp 的激活调用激活多个 Chimp
HRESULT hr = CoTreatAsClass(CLSID_Chimp, CLSID_NULL);
```

这个调用恢复 Chimp 类的原来实现，回到被模仿之前的状态。客户可以使用 CoGetTreatAsClass API 函数，来询问指定类的模仿设置（emulation setting）：

```
HRESULT CoGetTreatAsClass([in] REFCLSID rclsidOld,
                           [out] REFCLSID *pclsidNew);
```

如果被询问的类已经被另一个类模仿了，那么模仿类的 CLSID 将通过第二个参数被返回给客户，并且函数返回 S\_OK。如果被询问的类未被其他类模仿，那么第二个参数将返回原来的 CLSID，并且函数返回 S\_FALSE。值得一提的是，在本书写作时刻，对于远程激活请求，类模仿并不能如预期的那样正常工作。

### 3.14 组件类别

正如本章所重点强调的，基本的 COM 激活方式要求调用者必须知道精确的类名，以便创建新的实例。然而，有时候，我们只是简单地请求某个“遵循了某些语义限制”的任何类，在有些情况下这种做法会非常有用。而且，客户在发出激活请求之前，就知道一个类要求客户具有什么样的服务（service），这也会非常有用，因为这样可以避免创建“客户无法为它提供相应支持”的对象。这些问题促成了组件类别（component category）的概念。

COM 允许实现者可以把相关的一组 COM 类，组织到逻辑组或者组件类别中。通常，一个类别（category）中的所有类都将实现同一组接口。然而，对于许多应用来说，简单地根据“每个类实现了哪些接口”对类空间进行划分，并不能提供合适的划分单位。组件类别可以被看作“元信息（metainformation）”，用来指示“哪些类与特定的语义限制相兼容”。

一个组件类别是一组逻辑相关的 COM 类，它们共享同一个类别 ID（category ID，CATID）。CATID 也是 GUID，它作为类的属性被保存在注册表中。每个类可能有两个子键：“Implemented Categories”和“Required Categories”。假设现在有两个组件类别：Simians 和 Mammals。这两个类别各自有一个唯一的 CATID（分别为 CATID\_Simians 和 CATID\_Mammals）。也假设 Chimp 类是这两个类别的成员，那么 Chimp 的“Implemented Categories”注册表键将包含两个独立的子键，分别对应于这两个类别的 GUID：

```
[HKCR\CLSID\{CLSID_Chimp}\Implemented Categories\{CATID_Mammals}]
[HKCR\CLSID\{CLSID_Chimp}\Implemented Categories\{CATID_Simians}]
```

这些注册表项通常是在自注册的时候被加进去的。在系统中，每一个已知的组件类别在下面的键下都有一个表项：

```
HKEY_CLASSES_ROOT\Component Categories
```

每个类别都有它自己唯一的子键，由它的 CATID 命名。在它的子键下面，每个类别有一个或者多个已命名的值（named value，也被称为名字值），这些名字值包含类别的友好文本描述。例如，上面显示的两个类别将要求下面的注册表项：

```
[HKCR\Component Categories\{CATZD_Mammals}]
```

```
409="Bears live young"
```

```
[HKCR\Component Categories\{CATZD_Simians}]
```

```
409="Eats Bananas"
```

注意，这个例子使用了 409 值，它正是“U.S. English（美国英语）”的 LCID（地区 ID）。只要加入其他的名字值就可以支持其他的地区。

对于类而言，它有可能利用组件类别来表达“它需要客户端提供某些特殊类型的功能”。这种支持通常以“站点接口（site interface）”的形式出现，站点接口是在对象被激活之后由客户提供给对象的接口。为了使这些由客户端提供的接口能够被分类，并且独立于任何特定的接口，COM 允许一个类公布第二种类型的类别 ID，通过这种机制客户可以确保“不会激活一个自己无法容纳的组件”。假设存在下面两种“由客户提供服务”的类别：CATID\_HasOxygen 和 CATID\_HasWater。由于黑猩猩需要氧气和水才能活下去，所以 Chimp 的实现者可能会公布这样的要求：客户只有提供了这两类服务才可能激活对象。这可以通过“Required Categories”子键来完成：

```
[HKCR\CLSID\{CLSID_Chimp}\Required Categories\{CATID_HasOxygen}]
```

```
[HKCR\CLSID\{CLSID_Chimp}\Required Categories\{CATID_HasWater}]
```

这两个类别 ID 也需要被注册到以下位置：

```
HKEY_CLASSES_ROOT\Component Categories
```

有了这些注册信息之后，客户有责任在激活之前首先满足必要的类别条件。COM 并不强迫客户必须符合这样的条件。

我们既可以使用显式的注册表函数调用来注册与组件类别有关的注册项，也可以使用 COM 提供的组件类别管理器（component category manager）。COM 的组件类别管理器以一个可实例化的 COM 类（CLSID\_StdComponentCategoriesMgr）的形式暴露出来，它实现了 ICatRegister 和 ICatInformation 接口，分别用来注册和查询类别信息。ICatRegister 接口使服务器 DLL 可以很容易地在注册表中加入必要的注册项目。ICatRegister 接口的 IDL 定义如下：

```

[ object, uuid(0002E012-0000-0000-C000-000000000046) ]
interface ICatRegister : IUnknown
// 类别信息
typedef struct tagCATEGORYINFO {
    CATID      catid
    LCID       lcid;
    OLECHAR szDescription [128] ;
} CATEGORYINFO;
// 注册 cCts 类别
HRESULT RegisterCategories([in] ULONG cCts,
                           [in, size_is(cCts)] CATEGORYINFO rgCatInfo[]);
// 取消注册 cCategories 类别
HRESULT UnRegisterCategories([in] ULONG cCategories,
                           [in, size_is(cCategories)] CATID rgcatid[]);
// 注册一个类，实现一个或更多类别
HRESULT RegisterClassImplCategories([in] REFCLSID rclsid,
                                      [in] ULONG cCategories,
                                      [in, size_is(cCategories)] CATID rgcatid[]);
// 取消注册
HRESULT UnRegisterClassImplCategories([in] REFCLSID rclsid,
                                       [in] ULONG cCategories,
                                       [in, size_is(cCategories)] CATID rgcatid[]);
// 注册一个类要求一个或更多类别
HRESULT RegisterClassReqCategories([in] REFCLSID rclsid,
                                     [in] ULONG cCategories,
                                     [in, size_is(cCategories)] CATID rgcatid[]);
// 取消注册
HRESULT UnRegisterClassReqCategories([in] REFCLSID rclsid,
                                       [in] ULONG cCategories,
                                       [in, size_is(cCategories)] CATID rgcatid[]);
}

```

用户定义的 COM 类没有必要实现这个接口。它存在的目的是为了让服务器使用 COM 提供的类别管理器实现，以便自注册它们的组件类别信息。

对于 Chimp 例子的情形，下面的代码将为每一个类别注册正确的信息：

```

// 获取标准类别管理器
ICatRegister *pcr = 0;
HRESULT hr = CoCreateInstance(
    CLSID_StdComponentCategoriesMgr, 0,
    CLSCTX_ALL, IID_ICatRegister, (void**)&pcr);
if (SUCCEEDED(hr)) {
// 建立每个类别的信息
    CATEGORYINFO rgcc[4];
    rgcc[0].catid = CATID_Simian;
    rgcc[1].catid = CATID_Mammal;
    rgcc[2].catid = CATID_HasOxygen;
    rgcc[3].catid = CATID_HasWater;
    rgcc[0].lcid = rgcc[1].tcid
        = rgcc[2].lcid = rgcc[3].lcid = 0x409;
}

```

```

wcscpy(rgcc[0].szDescription,OLESTR("Eats Bananas"));
wcscpy(rgcc[1].szDescription,OLESTR("Bearslive young"));
wcscpy(rgcc[2].szDescription,OLESTR("Provides Oxygen"));
wcscpy(rgcc[3].szDescription,OLESTR("Provides Water"));
// 注册类别信息
pcr->RegisterCategories(4, rgcc);
// 注明黑猩猩是类人猿和哺乳动物
CATID rgcid[2];
rgcid[0] = CATID_Simian, rgcid[1] = CATID_Mammal;
pcr->RegisterClassImplCategories(CLSID_Chimp, 2, rgcid)
// 注明黑猩猩需要氧气和水
rgcid[0] = CATID_HasOxygen, rgcid[1] = CATID_HasWater;
pcr->RegisterClassReqCategories(CLSID_Chimp, 2, rgcid)
pcr->Release();
}

```

注意，这段代码并没有直接调用 Win32 注册表 API 函数，而是使用标准的类别管理器来操纵注册表。

标准类别管理器也可以让应用查询注册表，以便找到与类别有关的信息。这项功能通过 **ICatInformation** 接口被提供出来：

```

[ object, uuid(0002E013-0000-0000-C000-000000000046)
interface ICatInformation : IUnknown {
// 获取已知类别列表
HRESULT EnumCategories([in] LCID lcid,
[out] IEnumCATEGORYINFO** ppeci);
// 获取特定类别信息
HRESULT GetCategoryDesc([in] REFCATID rcatid,
[in] LCID lcid,
[out] OLECHAR ** ppszDesc);
// 获取与指定类别兼容的类的列表
HRESULT EnumCl assesOfCategories (
[in] ULONG cImplemented, // -1 表示忽略
[in, size_is (cImplemented) ] CATID rgcatidImpl [],
[in] ULONG cRequired, // -1 表示忽略
[in, size_is (cRequired) ] CATID rgcatidReq [],
[out] IEnumCLSID** ppenumClsid);
// 验证类与指定类别兼容
HRESULT IsCIassOfCategories([in] REFCLSID rclsid,
[in] ULONG cImplemented,
[in, size_is (cImplemented) ] CATID rgcatidImpl [],
[in] ULONG cRequired,
[in, size_is (cRequired) ] CATID rgcatidReq []);
// 获取类的已实现类别列表
HRESULT EnumImplCategoriesOfClass([in] REFCLSID rclsid,
[out] IEnumCATID** ppenumCatid);
// 获取类的已要求类别列表
HRESULT EnumReqCategoriesOfClass([in] REFCLSID rclsid,

```

```

    [out] IEnumCATID** ppenumCatid);
}
}

```

这些方法大多数返回一个游标（cursor），指向类别列表或者类 ID 的列表。这些游标被称为枚举器（enumerator），第 7 章将讨论有关枚举器的细节。

下面的代码演示了如何提取出 Simian（译注 2）类别中成员类的列表：

```

// 获取标准类别管理器
ICatlnformation *pci = 0;
HRESULT hr = CoCreateInstance(
    CLSID_StdComponentCategoriesMgr, 0,
    CLSCTX_ALL, IID_ICatlnformation, (void**)&pci);
if (SUCCEEDED(hr)) {
// 获取是 Simians 的类(忽略 cat.s)
    IEnumCLSID *pec = 0;
    CATID rgcid[1];
    rgcid[0] = CATID_Simian;
    hr = pci->EnumClassesOfCategories(1, rgcid, -1, 0, &pec);
    if (SUCCEEDED(hr)) {
// 一次遍历 CLSIDs 64 列表
        enum { MAX = 64 };
        CLSID rgclsid[MAX];
        do {
            ULONG cActual = 0,
            hr = pec->Next(MAX, rgclsid, &cActual);
            if (SUCCEEDED(hr)) {
                for (ULONG i = 0; i < cActual; i++)
                    DisplayClass(rgclsid[i]);
            }
        } while (hr == S_OK);
        pec->Release ();
    }
    pci->Release (),
}
}

```

这段代码片断没有考虑这样的事实，即客户有可能不支持结果得到的类列表中所要求的类别。如果客户已经知道它所支持的站点类别（site category），那么它可以指定所有已被支持的类别的列表。

考虑下面对 `EnumClassesOfCategories` 的调用：

```

CATID rgimpl[1]; rgimpl[0] = CATID_Simians;
CATID rgreq[3]; rgreq[0] = CATID_HasWater;
rgreq[1] = CATID_HasOxygen; rgreq[2] = CATID_HasMilk;
hr = pci->EnumClassesOfCategories(1, rgimpl, 3, rgreq, &pec);

```

---

译注 2：原文为 Mammal，有误。

结果得到的类列表将包含所有“要求客户环境只提供 Oxygen（氧气）、Water（水）和 Milk（牛奶）”的 Simians（类人猿）。以前面注册过的 Chimp 类为例，Chimp 将是这样一个相兼容的类，因为它实现了指定的类别 Simian，并且要求“此查询中被指定的类别”的一个子集。

关于组件类别最后有必要讨论的一个概念是组件类别的缺省类。COM 允许一个 CATID 被当作 CLSID 注册在 HKEY\_CLASSES\_ROOT\CLSID 的下面。为了把一个 CATID 映射到缺省的 CLSID，我们要用到类模仿技术引入的 TreatAs 设施。

```
HKEY_CLASSES_ROOT\CLSID
```

为了表明 Gorilla 类是 Simian 的缺省类，我们要加入下面的注册表键：

```
\CLSID\{CATID_Simian\TreatAs}
@={CLSID_Gorilla }
```

这个简单的约定使得客户可以在应该出现 CLSID 的地方使用 CATID，例如：

```
// 创建缺省 Simian 类的一个实例
hr = CoCreateInstance(CATID_Simian, 0, CLSCTX_ALL,
                      IID_IApe, (void**) &pApe);
```

如果对于指定的类别没有相应的缺省类被注册，那么激活调用将会失败，返回 REGDB\_E\_CLASSNOTREG。

### 3.15 我们走到哪儿了？

本章讲述了 COM 类的概念。COM 类是被命名的实体数据类型，它引出一个或者多个接口，而且它也是 COM 中对象激活所用到的基本抽象。COM 支持三种激活方式。CoGetClassObject 绑定到类对象的引用，类对象是每个类范围内唯一的单体，它表现了一个类中与每个实例独立的功能。CoCreateInstanceEx 绑定到指向一个新的类实例的引用，而 CoGetInstanceFromFile 则绑定到指向文件中永久实例的引用。名字对象则被用作一种统一的抽象机制，用来向客户暴露绑定和激活策略，而 MkParseDisplayName 则可以作为进入 COM 名字空间的入口点。

# 第 4 章

## 对 象

```

class object
{
public:
template <class T> virtual
T * dynamic_cast(const type_info& t = typeid(T))
};

Anonymous, 1995

```

第 2 章从一般意义上讨论了 COM 接口的基础，并特别讨论了 `IUnknown` 接口。同时第 2 章也阐述了这样的概念：只要对象从多个接口继承，它就可以暴露多种功能。同时这一章还展示了一种机制，通过这种机制客户可以询问对象发现对象其他的功能。这种机制就是 `QueryInterface`，它可以被看作 C++ 中 `dynamic_cast` 操作符的“与编程语言和编译器均无关”的版本。

上一章演示了 `QueryInterface` 可以直接通过静态类型转换来实现，限制一个对象的 `this` 指针的范围，使它指向客户所请求的接口的类型。在物理层次上，这项技术只是简单地把接口的标识符映射到对象内部适当的偏移量上，每个 C++ 编译器在实现 `dynamic_cast` 时也用到了这样的技术。

尽管上一章中 `QueryInterface` 的实现是完全合法的 COM，但是 `IUnknown` 的规则允许对象实现者可以拥有更大的灵活性。这一章将探究这些规则，并演示它们所隐含的实现技术。

## 4.1 再谈 IUnknown

在 COM 的系统调用接口中，IUnknown 并没有缺省的实现。SDK 头文件没有包含任何能够提供 QueryInterface、AddRef 和 Release 实现代码的基类、宏或者模板，而这三个函数却是每一个 COM 程序（用 C 或者 C++）所必须用到的。相反，COM 规范提供了非常明确的规则，考虑到了客户和对象对这三个方法可能做出的种种假设。这组规则构成了 IUnknown 的协议，允许每个对象的实现者把这三个方法映射到他（或者她）的对象中有意义的部分中。

第 2 章展示的实际上是这三个方法的 C++ 实现，但是 COM 并没有强制对象必须这样做。COM 所要求的是，每个对象必须遵从 IUnknown 的基本规则。到底如何实现这些规则则并不是 COM 所关心的。这使得 COM 非常不引入注意，因为它并不要求对象必须调用系统函数、从系统提供的实现中派生新的功能，或者做任何除了“暴露 COM 兼容的 vptr”之外的任何事情。实际上，正如本章后文将要提到的，即使从那些并没有继承任何 COM 接口的类中，我们也有可能暴露出从 IUnknown 派生（IUnknown-derived）而来的 vptr。

IUnknown 的规则合起来实际上定义了 COM 对象的含义。为了理解 IUnknown 的规则，从一个具体的例子开始会非常有帮助。考虑下面的接口层次：

```
import "unknwn.idl";

[object, uuid(CD538340-A56D-11d0-8C2F-0080C73925BA)]
interface IVehicle : IUnknown {
    HRESULT GetMaxSpeed([out, retval] long *pMax);
}

[object, uuid(CD538341-A56D-11d0-8C2F-0080C73925BA)]
interface ICar : IVehicle {
    HRESULT Brake(void);
}

[object, uuid(CD538342-A56D-11d0-8C2F-0080C73925BA)]
interface IPPlane : IVehicle {
    HRESULT TakeOff(void);
}

[object, uuid(CD538343-A56D-11d0-8C2F-0080C73925BA)]
interface IBoat : IVehicle {
    HRESULT Sink(void);
}
```

COM 使用一种标准技术来可视地表达对象。这项技术也遵从 COM 的“接口与实现分离”的原则，并没有强加对象的任何实现细节，而只是列出它所暴露的接口。这项技术也以可视化的形式强化了 **IUnknown** 的许多规则。图 4.1 显示了类 **CarBoatPlane** 的标准图形表示，类 **CarBoatPlane** 实现了刚才定义的每一个接口。注意，从这个图中我们唯一能够推断出的一点是，除非发生重大的灾难，否则 **CarBoatPlane** 将暴露出五个接口：**IBoat**、**IPlane**、**ICar**、**IVehicle** 和 **IUnknown**。

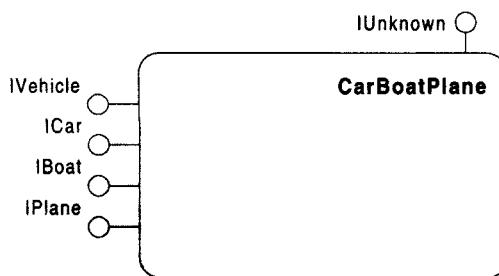


图 4.1 **CarBoatPlane** 对象表示法

我们要研究的第一个 **IUnknown** 规则是 **QueryInterface** 的对称性/传递性/自反性 (symmetric/transitive/reflexive) 要求。这个规则定义了一个对象的所有接口相互之间的关系，并开始定义 COM 中对象实体身份 (object identity) 的概念。与所有的 **IUnknown** 规则一样，这些要求在任何时候都必须要被遵守（除非发生重大的灾难），以便确定一个 COM 对象是否有效。

## 4.2 **QueryInterface** 是对称的

COM 规范要求，如果通过类型为 A 的接口指针，对接口 B 的 **QueryInterface** 请求能够被满足，那么在结果得到的类型为 B 的接口指针（指向同一对象）上，对接口 A 的 **QueryInterface** 请求应该永远也不会失败。这意味着，如果：

$QI(A) \rightarrow B$

是真 (true) 的，那么：

$QI(QI(A) \rightarrow B) \rightarrow A$

必须也是真的。

这个特性如图 4.2 所示，它隐含了下面代码中的断言 (assertion) 必须总是真的：

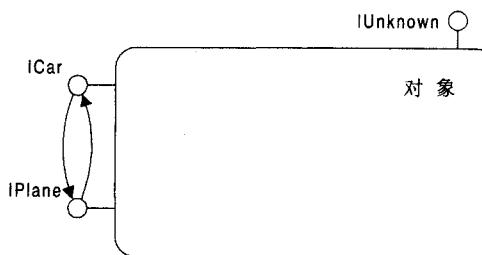


图 4.2 QueryInterface 和对称性

```

void AssertSymmetric(ICar *pCar) {
    if (pCar) {
        IPlane *pPlane = 0;
        // 请求另一种类型的接口
        HRESULT hr = pCar->QueryInterface(IID_IPlane,
                                             (void**)&pPlane);
        if (SUCCEEDED(hr)) {
            ICar *pCar2 = 0;
            // 请求原类型的接口
            hr = pPlane->QueryInterface(IID_ICar,
                                           (void**)&pCar2);
            // 如以判断失败, pCar
            // 不会指向合法 COM 对象
            assert(SUCCEEDED(hr));
            pCar2->Release ();
        }
        pPlane->Release ();
    }
}

```

QueryInterface 的对称性意味着，客户不必关心先获得哪个接口指针，因为任何两个接口类型有可能以任意顺序被获得。

### 4.3 QueryInterface 是可传递的

COM 规范也要求，如果通过类型为 A 的接口指针，对接口 B 的 QueryInterface 请求能够被满足，并且通过类型为 B 的接口指针，对接口 C 的 QueryInterface 请求也能够被满足，那么通过原来类型 A 的接口指针，对接口 C 的 QueryInterface 请求一定也会成功。这意味着，如果：

$QI(QI(A) \rightarrow B) \rightarrow C$

是真的，那么：

`QI(A) ->C`

必须也是真的。

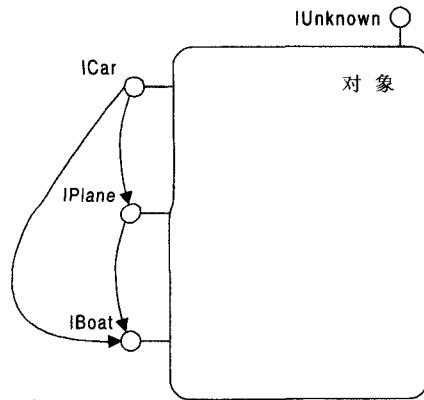


图 4.3 QueryInterface 和传递性

这个要求如图 4.3 所示，它隐含了下面代码中的断言必须总是真的：

```

void AssertTransitive(ICar *pCar) {
    if (pCar) {
        IPlane *pPlane = 0;
        // 请求中间类型接口
        HRESULT hr = pCar->QueryInterface(IID_IPlane,
                                             (void**)&pPlane);
        if (SUCCEEDED(hr)) {
            IBoat *pBoat1 = 0;
            // 请求最终类型接口
            hr = pPlane->QueryInterface(IID_IBoat,
                                         (void**)&pBoat1);
            if (SUCCEEDED(hr)) {
                IBoat *pBoat2 = 0;
                // 通过原指针请求最终类型
                hr = pCar->QueryInterface(IID_IBoat,
                                             (void**)&pBoat2);
                // 如以下断言失败，pCar
                // 不会指向合法 COM 对象
                assert(SUCCEEDED(hr));
                pBoat2->Release();
            }
            pBoat1->Release();
            pPlane->Release();
    }
}

```

QueryInterface 的对称性意味着，一个对象所暴露的所有接口都是平等的，客户无

需以任何特定的顺序来获得某个接口指针。如果这个要求不成立的话，那么客户就要考虑对象的哪个指针必须要使用相应的 `QueryInterface` 请求。`QueryInterface` 的传递性和对称性隐含着：对于任一个 `QueryInterface` 请求，对象的任何一个接口指针都会产生同样的 Yes/No 回答。传递性和对称性唯一未能涵盖的情形是“多次请求同一个接口”。这种情况要求 `QueryInterface` 必须是自反的。

## 4.4 `QueryInterface` 是自反的

COM 规范要求，如果客户在发出 `QueryInterface` 请求时，被请求的类型与发出请求所使用的指针的类型一致的话，那么通过这个接口指针发出的 `QueryInterface` 请求总是会成功的。这意味着：

$QI(A) \rightarrow A$

必须总是真的。

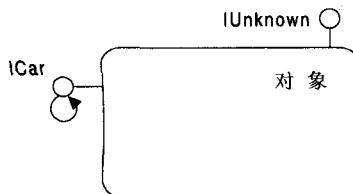


图 4.4 `QueryInterface` 和自反性

这个要求如图 4.4 所示，以及下面的代码片断：

```

void AssertReflexive(ICar *pCar) {
    if (pear) {
        ICar *pCar2 = 0;
        // 请求同类型接口
        HRESULT hr = pCar->QueryInterface(IID_ICar,
                                             (void**)&pCar2);
        // 如以下断言失败,pCar
        // 不指向合法 COM 对象
        assert (SUCCEEDED(hr));
        pCar2->Release();
    }
}

```

这段代码隐含着 `ICar` 的所有实现必须满足这样的条件：通过 `ICar` 接口指针，进一步查询 `ICar` 接口的 `QueryInterface` 请求必须能够得到满足。如果不是这样的话，那么就

不可能“通过基类型参数来传递派生类型接口，而不会丢失原来的类型”，如下面的代码：

```

extern void GetCar(ICar **ppcar);
extern void UseVehicle(IVehicle *pv);
ICar *pCar;
GetCar (&pCar);
UseVehicle(pCar); // ICar 类型失去了
void UseVehicle(IVehicle *pv) {
ICar *pCar = 0;
// try to regain syntactic ICar-ness
HRESULT hr = pv->QueryInterface(IID_ICar,
                                  (void**)&pCar);
}

```

因为 UseVehicle 函数用到的指针值与客户传递进来的 ICar 指针有同样的值，所以，如果在函数内部 ICar 类型指针不能被再次获取的话，这显然是违背常理的。

QueryInterface 具有对称性、自反性和传递性，这隐含着一个对象的任何一个接口指针，对于给定的 QueryInterface 请求都应该产生同样的 Yes/No 回答。这使得客户可以把对象的类型层次简单地看作一个图，图中所有的节点相互之间都直接显式地连接起来。图 4.5 显示了这样一个图。注意图中任何一个节点只要一步就能够到达任何其他的节点。

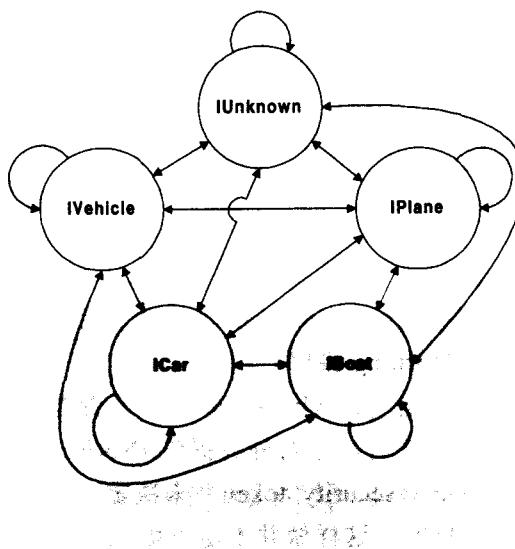


图 4.5 接口和 QueryInterface

## 4.5 对象具有静态类型

从 `QueryInterface` 的三个要求可以得出一个必然结论是，对象所支持的接口集合不能随时间的变化而变化。COM 规范明确要求，所有对象都必须满足这个结论。这个要求暗示着这样的事实：尽管客户必须通过动态协商的手段，才能确定对象到底支持哪些类型，但是对象的类型层次是静态不变的。如果一个对象对于接口 A 的查询请求回答“`Yes`”，那么从这一刻起，它必须总是回答“`Yes`”。如果一个对象对于接口 A 的查询请求回答“`No`”，那么从这一刻起，它必须总是回答“`No`”。这里“从这一刻起”可以解释为“只要这个对象还存在至少一个未完结的接口指针”。这往往对应于底层 C++ 对象的生命周期，但是 COM 规范的语言非常宽松，允许实现者有足够的灵活性（比如，对于全局变量而言，在所有的指针都被释放之后，它的类型层次有可能发生变化）。

所有的 COM 对象具有静态类型层次，这隐含着下面代码中的断言永远也不会是“假 (`false`)”的，不管第二个参数是什么样的接口标识符。代码如下：

```
void AssertStaticType(IUnknown *pUnk, REFIID riid) {
    IUnknown *pUnk1 = 0, *pUnk2 = 0;
    HRESULT hr1 = pUnk->QueryInterface(riid,
                                         (void**)&pUnk1);
    HRESULT hr2 = pUnk->QueryInterface(riid,
                                         (void**)&pUnk2);

    //同一接口的请求
    //yes/no 回答也相同
    assert(SUCCEEDED(hr1) == SUCCEEDED(hr2));
    if (SUCCEEDED(hr1)) pUnk1->Release();
    if (SUCCEEDED(hr2)) pUnk2->Release();
```

这个要求也意味着在 COM 中，下面的程序设计技术是禁止使用的：

1. 使用临时的信息来决定 `QueryInterface` 请求是否能够得到满足（比如，只有在中午 12:00 以前才对外界提供 `IMorning` 接口）。
2. 使用短暂状态信息来决定 `QueryInterface` 请求是否能够得到满足（比如，只有当未完结接口指针的数目少于 10 个时，才对外界提供 `INotBusy` 接口）。
3. 使用调用者的安全令牌（security token）来决定 `QueryInterface` 请求是否能够得到满足。正如第 6 章将要解释的，这样做并不足以提供真正的安全性，因为 COM 使用的是有线协议。
4. 根据动态申请资源成功与否来决定 `QueryInterface` 请求是否能够得到满足（比如，只有 `malloc (4096*4096)` 返回成功时，才对外界提供 `IHaveTonsOfMemory` 接口）。

上面最后一项技术可以考虑放宽一些，如果对象实现者愿意练习 COM 规范中“除

非发生重大灾难（barring catastrophic failure）”条款的话。

这些限制并不意味着“同一实现类的两个对象在被请求同一个接口的时候，不能够给出不同的 Yes/No 回答”。例如，有一个类可能实现了前面给出的 ICar、IBoat 和 IPlane 接口，但是对于特定的每个对象只允许它使用一个接口。这些限制也不意味着“一个对象不能够使用状态或者临时信息，来决定对于某个接口的初始请求回答 Yes/No”。在刚才提到的、三个接口中只允许一个接口被暴露出去的类的例子中，下面的用法完全是合法的：

```
class CBP : public ICar, public IPlane, public IBoat{
    enum TYPE { CAR, BOAT, PLANE, NONE };
    TYPE m_type;
    CBP(void) : m_type(NONE) {}
    STDMETHODIMP QueryInterface(REFIID riid,
                               void **ppv) {
        if (riid == IID_ICar) {
            // 第一个 QI 初始化对象类型
            if (m_type == NONE) m_type = CAR;
            // 如对象是一个 car, 才满足请求
            if (m_type == CAR)
                *ppv = static_cast(this);
            else
                return (*ppv = 0), E_NOINTERFACE;
        }
        else if (riid == IID_IBoat)
            // IBoat 和 Iplane 的处理相同
    };
}
```

“对象所支持的接口集合是静态的”这个要求隐含着，对象的实现者不允许根据“一个对象对于某个特定的接口回答不同的 Yes/No 答案”来进行设计。对象的类型层次应该被认为在整个生命周期过程中是不变的，其中一个理由是：对于远程对象，COM 并不保证客户的所有 `QueryInterface` 请求都将被传递给对象。这使得客户端的代理对象可以把 `QueryInterface` 的结果缓存起来，以避免过多的客户-对象之间通信。这样的优化对于 COM 的性能非常关键，但是却破坏了“利用 `QueryInterface` 与调用者动态交换语义信息”的设计原则。

## 4.6 唯一性和对象实体身份

上一节认为，`QueryInterface` 请求应该给调用者返回 Yes/No 回答。当然，`QueryInterface` 确实返回 `S_OK` (Yes) 或者 `E_NOINTERFACE` (No)。然而，当 `QueryInterface` 返回 `OK` 时，它也返回一个指向对象的接口指针。对于这个指针的值，COM 有一些特殊的要

求，客户由此可以确定两个接口指针是否指向同一个对象。

## 4.7 QueryInterface 和 IUnknown

`QueryInterface` 的自反性确保了“任何一个接口都能够满足对 `IUnknown` 的查询请求”，因为所有的接口指针都暗含着是 `IUnknown` 类型。COM 规范在描述针对 `IUnknown` 的 `QueryInterface` 请求的结果时，还有一些特殊的限制。不仅所有的对象对于这样的请求必须回答“*Yes*”，而且它必须对每个请求都要返回完全相同的指针值。这意味着下面代码中的两个断言都必须总是真的：

```
void AssertSameObject(IUnknown *pUnk) {
    IUnknown *pUnk1 = 0, *pUnk2 = 0;
    HRESULT hr1 = pUnk->QueryInterface(IID_IUnknown,
                                         (void**)&pUnk1);
    HRESULT hr2 = pUnk->QueryInterface(IID_IUnknown,
                                         (void**)&pUnk2);
    // QueryInterface(IUnknown) 必须总是成功
    assert(SUCCEEDED(hr1) && SUCCEEDED(hr2));
    // IUnknown 两个请求必须总是产生
    // 相同指针值
    assert(pUnk1 == punk2);
    pUnk1->Release();
    pUnk2->Release();
}
```

此项要求允许客户可以比较两个接口指针，判断它们是否指向同样的对象实体身份

```
bool IsSameObject(IUnknown *pUnk1, IUnknown *pUnk2) {
    assert(pUnk1 && punk2);
    bool bResult = true;
    if (pUnk1 != punk2) {
        HRESULT hr1, hr2;
        IUnknown *p1 = 0, *p2 = 0;
        hr1 = pUnk1->QueryInterface(IID_IUnknown,
                                      (void**)&p1);
        assert(SUCCEEDED (hr1));
        hr2 = pUnk2->QueryInterface(IID_IUnknown,
                                      (void**)&p2);
        assert (SUCCEEDED(hr1));
        // 比较两个指针值，它们
        // 代表对象身份
        bResult = (p1 == p2);
        p1->Release();    p2->Release();
    }
}
```

```

    }
    return bResult;
}
}

```

正如第5章将要讨论的，实体身份（identity）的观念是COM远程结构中用到的基本概念，在网络上它可以有效地表达接口指针。

有了IUnknown的规则作为武装之后，我们就可以用来检查一个对象的实现，并检验它是否满足所有必要的条件。下面的实现暴露了四个与交通有关的接口，以及IUnknown：

```

class CarBoatPlane : public ICar,
                     public IBoat,
                     public IPlane {

public:
    // IUnknown 方法
    STDMETHODIMP QueryInterface(REFIID, void**);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
    // IVehicle 方法
    STDMETHODIMP GetMaxSpeed(long *pMax);
    // ICar 方法
    STDMETHODIMP Brake(void);
    // IBoat 方法
    STDMETHODIMP Sink(void);
    // IPlane 方法
    STDMETHODIMP TakeOff(void);
};


```

以下是CarBoatPlane的QueryInterface实现代码：

```

STDMETHODIMP QueryInterface (REFIID riid, void **ppv) {
    if (riid == IID_IUnknown)
        *ppv = static_cast<ICar*>(this);
    else if (riid == IID_IVehicle)
        *ppv = static_cast<ICar*>(this);
    else if (riid == IID_ICar)
        *ppv = static_cast<ICar*>(this);
    else if (riid == IID_IBoat)
        *ppv = static_cast<IBoat*>(this) ;
    else if (riid == IID_IPlane)
        *ppv = static_cast<IPlane*>(this);
    else
        return (*ppv = 0), E_NOINTERFACE;
    (* (IUnknown*) *ppv) ->AddRef ();
    return S_OK;
}

```

为了使 CarBoatPlane 成为一个 COM 对象，它的 `QueryInterface` 实现必须完全遵守本章前面给出的 `IUnknown` 规则。

`CarBoatPlane` 类只暴露五个接口：`ICar`、`IPlane`、`IBoat`、`IVehicle` 和 `IUnknown`。每个 `CarBoatPlane` `vtbl` 都将指向上面显示的 `QueryInterface` 实现。对象所支持的任何一个接口都可以通过这个 `QueryInterface` 实现得到，所以要想找到两个不对称的接口是不可能的，也就是说，不存在这样的接口 A 和接口 B，它们不满足下面的条件：

```
If QI(A) ->B Then QI(QI(A) ->B) ->A
```

按照同样的逻辑，因为所有这五个接口都共享同样的 `QueryInterface` 实现，所以不存在这样三个接口 A、B 和 C，它们不满足下面的条件：

```
If QI(QI(A) ->B) ->C Then QI(A) ->C
```

最后，因为 `QueryInterface` 的实现总是能够满足对这五个可能的接口指针的请求，所以客户可以确信，下面的操作对于这五个接口都能够成立：

```
QI(A) ->A
```

因为多重继承的原因，它只为这五个接口生成一个 `QueryInterface` 实现，所以事实上要想违背对称性、传递性和自反性要求并不容易。

前面的 `QueryInterface` 实现也能够正确地满足 COM 的实体身份规则，当被请求查询 `IUnknown` 接口时，它只返回一个指针值：

```
if (riid == IID_IUnknown)
    *ppv = static_cast<ICar*>(this);
```

如果 `QueryInterface` 实现为每个请求返回不同的 vptr，例如：

```
if (riid == IID_IUnknown) {
    int n = rand() % 3,
        if (n == 0)
            *ppv = static_cast<ICar*>(this);
        else if (n == 1)
            *ppv = static_cast<IBoat*>(this);
        else if (n == 2)
            *ppv = static_cast<IPlane*>(this);
}
```

那么这个实现也能够正确地满足纯粹的 C++ 类型关系（也就是说，这三个接口都与被请求的 `IUnknown` 类型相兼容）。然而，这不是合法的 COM 实现，因为它违反了 `QueryInterface` 的实体身份规则。

## 4.8 多重接口和方法名字

要想在 C++类中实现 COM 接口，多重继承是一项非常有效和直接的技术。它只要求很少的编码工作，因为大部分“有关建立 COM 兼容的 vptr 和 vtbl”的工作已经由编译器和链接器来完成了。如果一个方法名字出现在多个基类中，而且具有同样的参数类型，那么编译器和链接器为这个类只产生一个方法实现，然后让每个 vtbl 中的表项都指向这个方法实现。这种行为同样适用于方法 QueryInterface、AddRef 和 Release，因为所有的 COM 接口都以这三个方法作为开始，然而类的实现者只需要为每个方法写一次代码即可。同样的行为也适用于任意接口中具有相同名字、相同原型的方法。这是多重继承潜在的陷阱。

本章前面提到的交通工具接口层次中有一个名字冲突。ICar 接口有一个被称为 GetMaxSpeed 的方法，IBoat 和 IPlane 接口也有名为 GetMaxSpeed 的方法，而且它们具有同样的原型。这意味着在使用多重继承时，类的实现者只能编写一个 GetMaxSpeed 方法，然后编译器和链接器将会初始化 ICar、IBoat 和 IPlane 的 vtbl，使它们的 GetMaxSpeed 表项都指向这个实现。

这种行为对于大多数的实现来说是非常合理的。但是如果对象需要根据被请求的接口类型的不同，而返回不同的最大速度，那又该怎么办呢？因为名字和原型都一样，所以我们必须采取特殊的措施，才能够使这些名字冲突的方法具有多个实现。一种常用的技术是，建立一个中间 C++类，让它从某一个接口继承，然后实现名字发生冲突的方法，但是这个方法只是简单地调用另一个不会发生名字冲突的纯虚函数。举例如下：

```
struct IXCar:public ICar {
    //增加新的非冲突纯虚方法
    virtual HRESULT STDMETHODCALLTYPE
        GetMaxCarSpeed(long *pval) = 0;

    //通过在派生类中调用
    //非冲突方法实现冲突方法
    STDMETHODIMP GetMaxSpeed(long *pval)
    { return GetMaxCarSpeed(pval); }
};
```

假定 IBoat 和 IPlane 接口都作了类似的处理，那么我们现在就可以实现 GetMaxSpeed 的不同版本，只要简单地从这些被扩展（即派生）之后的接口版本继承过来，然后覆盖每个 GetMaxSpeed 所对应的、未发生名字冲突的函数版本，代码如下：

```
class CarBoatPlane : public IXCar,
```

```

        public IXBoat,
        public IXPlane {

public:
    // IUnknown 方法
    STDMETHODIMP QueryInterface(REFIID, void**);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

    // Ivehicle 方法
    // 不要覆盖 GetMaxSpeed!
    // Icar 方法
    STDMETHODIMP Brake(void);
    // Iboat 方法
    STDMETHODIMP Sink(void);
    // IXPlane 方法
    STDMETHODIMP TakeOff(void);

    // 从 IXCar::GetMaxSpeed 调用
    STDMETHODIMP GetMaxCarSpeed(long *pval);
    // 从 IXBoat::GetMaxSpeed 调用
    STDMETHODIMP GetMaxBoatSpeed (long *pval);
    // 从 IXPlane:: GetMaxSpeed 调用
    STDMETHODIMP GetMaxPlaneSpeed(long *pval);
}

```

图 4.6 显示了这个类的布局结构和 vtbl 格式。注意，这个类并没有实现发生名字冲突

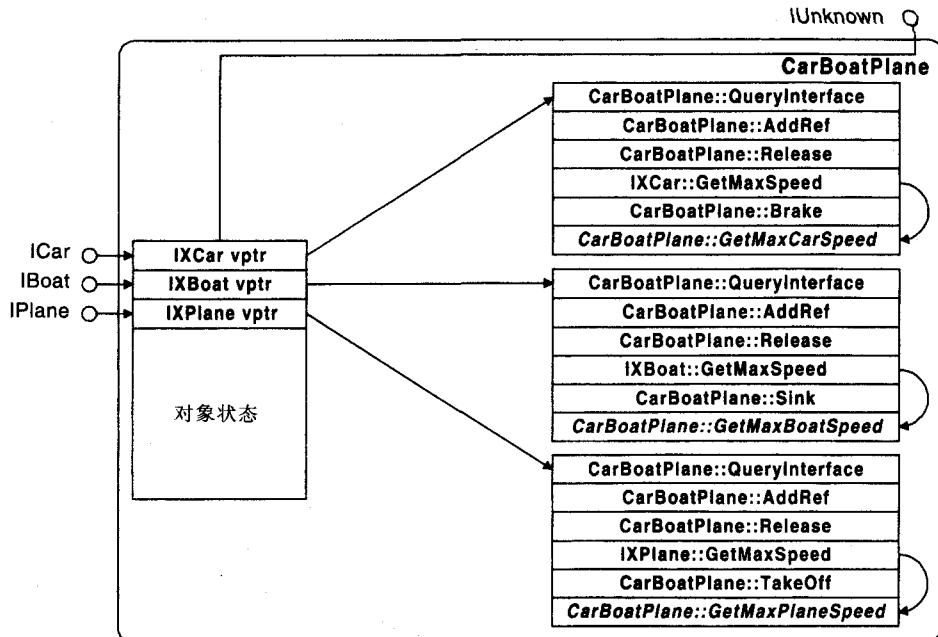


图 4.6 接口之间发生名字冲突的一种解决方案

突的方法 GetMaxSpeed。因为 CarBoatPlane 的每个基类都重载了这个纯虚函数，所以 CarBoatPlane 不需要再提供自己的实现。实际上，如果 CarBoatPlane 重载了 GetMaxSpeed 的话，那么它的方法实现将会覆盖掉每个基类中调用其他虚函数的版本，从而导致 IXCcar、IXBoat 和 IXPlane 的引入不再起任何作用。由于这个原因，这项技术只适用于能确保“实现类（或者任何可能的派生类）永远不会重载发生名字冲突的方法”的情形。

“为名字冲突的方法提供多个实现”的另一种技术是利用 IUnknown 的规则。COM 规范并不要求一个对象必须以 C++ 类的形式来实现。尽管基于多重继承的实现是“COM 对象和 C++ 类之间”最为自然的映射，但是这只不过是一种可能的实现技术而已。任何一种编程技术，只要它能够产生符合 COM 的 QueryInterface 规则的 vtbl 格式，它就可以被用来创建 COM 对象。解决名字冲突的一种常见技术是，把发生名字冲突的接口以不同的 C++ 类的形式实现，然后让目标 C++ 类组合（compose，即内嵌）这些类的实例。为了保证这些被组合进来的数据成员对于外界来说都是单个 COM 对象，通常的做法是使用 *QueryInterface* 的一个主实现（*master implementation*），然后每个被组合的数据成员的 *QueryInterface* 都委托给这个主实现。下面的代码说明了这项技术：

```

class CarPlane {
    LONG m_cRef;
    CarPlane(void) : m_cRef(0) {}
public:
    // IUnknown 主方法
    STDMETHODIMP QueryInterface(REFIID, void**);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
private:
    // 定义实现 Icar 的内嵌类
    struct XCar : public ICar {
        // 取回主对象的指针
        inline CarPlane* This();
        STDMETHODIMP QueryInterface(REFIID, void**);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);
        STDMETHODIMP GetMaxSpeed (long *pval);
        STDMETHODIMP Brake (void);
    };
    // 定义实现 IPlane 的内嵌类
    struct XPlane : public IPlane {
        // 取加主对象的指针
        inline CarPlane* This();
        STDMETHODIMP QueryInterface(REFIID, void**),
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void),
        STDMETHODIMP GetMaxSpeed (long *pval),
        STDMETHODIMP TakeOff(void);
    };
}

```

```
// 声明内嵌类的实例
XCar m_xCar;
XPlane m_xPlane;
};
```

上面代码中内嵌类的用法完全是可选的，但是它强调了一点，即这些附属类在 CarPlane 类的外部是没有意义的。图 4.7 说明了这个类的二进制布局结构和对应的 vtbl 布局结构。

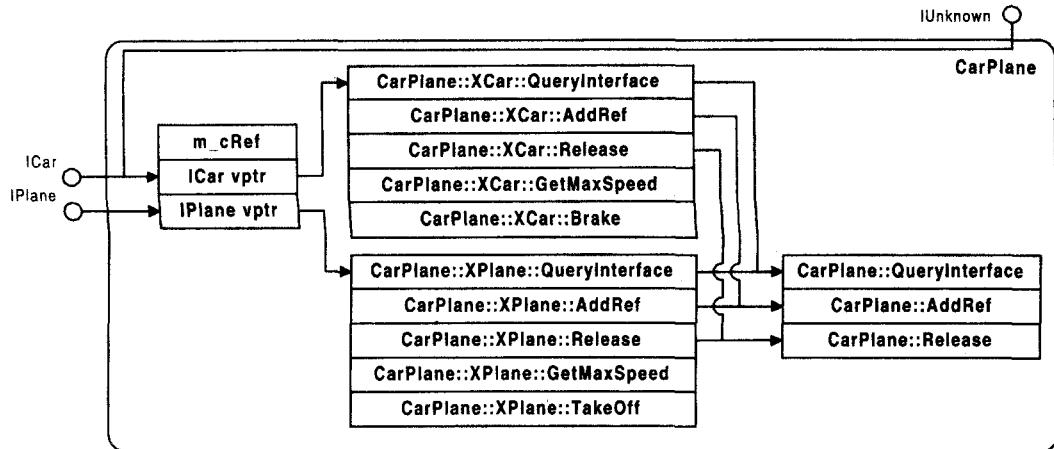


图 4.7 基于复合技术的实现

注意，这里有两个内嵌类的定义，分别对应于 CarPlane 将要实现的接口。这使得对象实现者可以提供两个 GetMaxSpeed 实现：

```
STDMETHODIMP CarPlane::XCar::GetMaxSpeed (long *pn) {
    // 将 *pn 设为汽车最大速度
}

STDMETHODIMP CarPlane::Xplane::GetMaxSpeed (long *pn) {
    // 将 *pn 设为行星最大速度
}
```

GetMaxSpeed 的两个实现分别位于不同的内嵌类定义中，这使得这个方法可以被定义两次，而且也保证了 ICar 和 IPlane 的 vtbl 将拥有不同的 GetMaxSpeed 表项。

同样值得一提的是，尽管上层类 CarPlane 也实现了 IUnknown 的方法，但是它并没有从 IUnknown（或者它的派生类）继承得来。相反，CarPlane 拥有两个数据成员，它们分别继承了相应的 COM 接口。这意味着，CarPlane 的 QueryInterface 不再针对对象自己使用 static\_cast 来找到特定的 vptr，而是返回一个指向数据成员（它实现了客户所请求的接口）的指针。QueryInterface 代码如下：

```

STDMETHODIMP CarPLane::QueryInterface(REFIID riid,
                                      void **ppv) {
    if (riid == IID_IUnknown)
        *ppv = static_cast<IUnknown*>(&m_xCar);
    else if (riid == IID_IVehicle)
        *ppv = static_cast<IVehicle*>(&m_xCar);
    else if (riid == IID_ICar)
        *ppv = static_cast<ICar*>(&m_xCar);
    else if (riid == IID_IPlane)
        *ppv = static_cast<IPlane*>(&m_xPlane);
    else
        return (*ppv = 0), E_NOINTERFACE;
    ((IUnknown*) (*ppv))->AddRef();
    return S_OK;
}

```

为了保证能够维护对象的实体身份，`CarPlane` 的每个数据成员必须要么在自己的 `QueryInterface` 实现中模仿这段代码，要么委托给 `CarPlane` 的 `QueryInterface` 主实现。为了做到这一点，我们必须要有种机制能够“让（组合）数据成员的成员函数可以得到外面的主对象”。`CarPlane::XCar` 类的定义包含一个内联函数，它根据被组合的数据成员的 `this` 指针，使用固定的偏移值来计算主对象的 `this` 指针，代码如下：

```

inline CarPlane CarPlane::XCar::This(void) {
    return (CarPlane*)((char*)this // 组合指针
                      - offsetof(CarPlane, m_xCar));
}

inline CarPlane CarPlane::XPlane::This(void) {
    return (CarPlane*)((char*)this // 组合指针
                      - offsetof(CarPlane, m_xPlane));
}

```

这种“后向指针计算”的技术是可移植的，而且也非常有效，因为它并不要求额外的数据成员，就可以在数据成员的方法实现内部找到外面的主对象。有了这样的后向指针函数之后，内部成员的 `QueryInterface` 实现就非常简捷了：

```

STDMETHODIMP CarPlane::XCar::QueryInterface(REFIID r,
                                             void**p) {
    return This()->QueryInterface(r, p);
}

STDMETHODIMP CarPlane::XPlane::QueryInterface (
    REFIID r, void**p) {
    return This()->QueryInterface(r, p);
}

```

对于 AddRef 和 Release 也可以使用同样的委托技术，从而让所有这些数据成员都维护一个统一的对象生命周期概念。

用复合类来实现接口，这样的技术比直接使用多重继承需要更多的代码。而且，最终产生的代码质量也可能不如使用多重继承更好（甚至可能非常差）。CarPlane 类并不需要从任何一个接口类派生，这并不意味着复合技术是一项非常合理的技术，可以把 COM 映射到传统的类库上（比如 MFC 就使用了这项技术）。在实现新的类的时候，使用复合技术的动机是，对于多个接口中定义完全相同的方法可以具有各自不同的实现。幸运的是，COM 定义的标准接口很少发生这样的名字冲突，而且当偶尔发生冲突时，这些发生名字冲突的方法往往被映射到语义上完全等价的功能。为了解决像前面提到的 GetMaxSpeed 这种情形的冲突，复合技术可能有点小题大做，因为按照第一种方法，用中间类把发生冲突的方法映射到唯一的虚函数调用，这种做法非常直接、非常有效，而且几乎不要求其他附加的代码。在新的代码中使用复合技术的主要动机，在于实现“以每个接口为基础的引用计数”。

有时候，我们非常希望在对象中以当前正在被使用的接口为基础来申请资源。然而，使用多重继承实现 COM 接口意味着所有的 vtbl 只使用一个 AddRef/Release 实现。尽管我们可以检测到对于给定接口的第一次请求，然后按需申请资源，例如：

```
STDMETHODIMP QueryInterface(REFIID riid, void **ppv) {
    if (riid == IID_IBoat) {
        // 第一次分配资源
        if (m_pTonsOfMemory == 0)
            m_pTonsOfMemory = new char[4096 * 4096];
        *ppv = static_cast<IBoat*>(this);
    }
    else if ...
}
```

但是我们没有办法检测到什么时候不存在未完结的 IBoat 接口指针了，因为客户通过 IBoat 接口发出的 Release 调用，与通过该对象上其他接口发出的 Release 调用是无法区分的。通常情况下，这正是我们所期望的，但是在上述情形下，通过 IBoat 接口发出的 AddRef 和 Release 调用必须要区别对待。如果使用复合技术来实现 IBoat 接口，那么它将会有自己专门的 AddRef 和 Release 实现，在其内部它可以维护自己的引用计数，与主对象的引用计数区别开来。代码如下：

```
class CarBoatPlane : public ICar, public IPlane {
    LONG m_cRef;
    char *m_pTonsOfMemory;
    CarBoatPlane(void): m_cRef(0), m_pTonsOfMemory(0) {}
public:
    // IUnknown 方法
    STDMETHODIMP QueryInterface(REFIID, void**);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release (void);
```

```

// IVehicle 方法
STDMETHODIMP GetMaxSpeed(long *pMax);
// ICar 方法
STDMETHODIMP Brake(void);
// IPlane 方法
STDMETHODIMP TakeOff(void);
// 定义实现 IBoat 的内嵌类
struct XBoat : public IBoat {
// get back pointer to main object
    inline CarBoatPlane* This();
    LONG m_cBoatRef;
// 每个接口的引用计数
    XBoat(void) : m_cBoatRef(0) {}
    STDMETHODIMP QueryInterface(REFIID, void**);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
    STDMETHODIMP GetMaxSpeed(long *pval);
    STDMETHODIMP Sink(void);
};
    XBoat m_xBoat;
};

现在，IBoat 的 AddRef 和 Release 实现可以跟踪记录 IBoat 类型的引用数目，当资源不再有用的时候就可以被释放掉，代码如下：
```

```

STDMETHODIMP_(ULONG) CarBoatPlane::XBoat::AddRef() {
    ULONG res = InterlockedIncrement(&m_cBoatRef);
    if (res == 1) { // 第一个 AddRef
// 分配资源并将 AddRef 传给对象
        This()->m_pTonsOfMemory = new char[4096*4096]
        This()->AddRef();
    }
    return res;
}

STDMETHODIMP_(ULONG) CarBoatPlane::XBoat::Release() {
    ULONG res = InterlockedDecrement(&m_cBoatRef);
    if (res == 0) { // 最后释放
// 自由资源并将 Release 传给对象
        delete [] This()->m_pTonsOfMemory;
        This()->Release();
    }
    return res;
}

```

为了使这项技术能够正常工作，所有接口指针用户必须遵从 COM 规范的要求，即 Release 调用必须作用在与它对应的 AddRef 调用的指针上。这就是为什么标准的 QueryInterface 实现的尾部会有这样的代码：

```
((IUnknown*) (*ppv)) ->AddRef(); // 使用 ptr
return S_OK;
```

而不会是这样：

```
AddRef(); // 只调用 this->AddRef
return S_OK;
```

前者可以确保当客户写下了下列正确的代码时：

```
IBoat *pBoat = 0;
HRESULT hr = pUnk->QueryInterface(IID_IBoat,
                                     (void**)&pBoat);
if (SUCCEEDED(hr)) {
    hr = pBoat->Sink();
    pBoat->Release();
}
```

AddRef 和 Release 使用的是同一个接口指针。

在表格驱动的 QueryInterface 实现的情况下，支持复合技术也是有可能的。以上一章展示的预处理器宏为基础，我们所需要做的就是再增加两个宏，一个用来指明被使用的是一个数据成员，而不是基类；另一个用来实现复合类中 IUnknown 的方法。代码如下：

```
class CarBoatPlane : public ICar, public IPlane {
public:
    struct XBoat : public IBoat {
        // 组合 QI/AddRef/Release/This()
        IMPLEMENT_COMPOSITE_UNKNOWN(CarBoatPlane,
                                     XBoat, m_xBoat)
        STDMETHODIMP GetMaxSpeed(long *pval),
        STDMETHODIMP Sink(void);
    };
    XBoat m_xBoat;

    // IVehicle 方法
    STDMETHODIMP GetMaxSpeed(long *pMax);
    // ICar 方法
    STDMETHODIMP Brake(void);
    // IPlane 方法
    STDMETHODIMP TakeOff(void);

    // 标准基于堆的 QI/AddRef/Release
    IMPLEMENT_UNKNOWN(CarBoatPlane)
    BEGIN_INTERFACE_TABLE(CarBoatPlane)
        IMPLEMENTS_INTERFACE_AS(IVehicle, ICar)
        IMPLEMENTS_INTERFACE(ICar)
        IMPLEMENTS_INTERFACE(IPlane)
```

```
// 计算数据成员偏移的宏
IMPLEMENTS_INTERFACE_WITH_COMPOSITE (IBoat,
                                      XBoat, m_xBoat)
END_INTERFACE_TABLE ()
};

上面的类定义中所缺少的只是对象的方法 QueryInterface、AddRef 和 Release 的定义。类定义中用到的两个新的宏被定义如下：
```

```
// inttable.h // (本书专用的头文件) /**
#define COMPOSITE_OFFSET(ClassName, BaseName, \
                      MemberType, MemberName) \
(DWORD(static_cast<BaseName*>(\
reinterpret_cast<MemberType*>(0x10000000 + \
offsetof(ClassName, MemberName))) - 0x10000000)

#define IMPLEMENTS_INTERFACE_WITH_COMPOSITE(Req, \
                                         MemberType, MemberName) \
{ &IID_##Req, ENTRY_IS_OFFSET, COMPOSITE_OFFSET(_IT, \
Req, MemberType, MemberName) },
// impunk.h // (本书专用的头文件) /**
#define IMPLEMENT_COMPOSITE_UNKNOWN (OuterClassName, \
                                   InnerClassName, DataMemberName) \
OuterClassName *This() \
{ return (OuterClassName*)((char*)this - \
offsetof(OuterClassName, DataMemberName)); } \
STDMETHODIMP QueryInterface(REFIID riid, void **ppv) \
{ return This()->QueryInterface(riid, ppv), } \
STDMETHODIMP_(ULONG) AddRef(void) \
{ return This()->AddRef(); } \
STDMETHODIMP_(ULONG) Release(void) \
{ return This()->Release(); }
```

这些预处理器宏只是简单地复制了复合类中用到的 QueryInterface、AddRef 和 Release 的实际实现而已。

## 4.9 动态复合

当我们在 C++类中使用多重继承或者复合技术来实现接口时，该类的每个对象针对每个它所支持的接口，都将需要承担 4 字节 vptr 的开销[假定 sizeof (void\*) == 4]。如果一个对象引出（暴露给外界）的接口数目非常小，那么这样的代价可以忽略不计，特别是相对于 COM 编程模型所带来的利益而言更是如此。然而，如果对象所支持的接口

的数目非常大，那么这些 vptr 开销有可能会增长到“使对象中与 COM 无关部分的代码反而变得更小的地步”。如果这些接口经常会被使用到，那么这些开销自然不可避免；但是，如果这些接口有的可能永远也不会被用到，或者只是在很短的时间内被用到，那么我们有可能利用 COM 规范中的漏洞对它进行优化处理，使得这些接口不在使用时，把它们从对象的 vptrs 中优化出去。

请回忆前面提到的这条规则：对同一个对象发出的所有 `QueryInterface` 请求，如果被请求的都是 `IUnknown` 接口，那么它们必须返回同样的指针值。这也正是 COM 中对对象实体身份得以建立起来的基本方式。然而，COM 规范明确地规定：对于 `IUnknown` 之外的其他类型接口请求，`QueryInterface` 实现每次可以返回不同的指针值。这意味着，对于不常被用到的接口，一个对象可以动态地按需分配 vptr 空间，而无须担心“每次针对同一个接口的请求，它必须返回完全相同的动态分配内存块”。这种“短暂分配复合技术”首次出现在 Microsoft 的白皮书“The COM Programmer's Cookbook”（作者 Crispin Goswell，<http://www.microsoft/com/oledev>）中。这份白皮书把这些短暂接口称为 `tearoff`。

把一个接口以 `tearoff` 的形式来实现，这有点类似于以复合技术来实现接口。为了 `tearoff` 我们也必须要定义第二个类，它从该 `tearoff` 所实现的接口派生。为了维护对象的实体身份，`tearoff` 的 `QueryInterface` 也必须委托给主类的 `QueryInterface`。但是 `tearoff` 技术和复合技术两者之间存在两个明显的区别：（1）主对象动态分配 `tearoff`，而不是拥有实例数据成员；（2）`tearoff` 复合类必须显式地维护一个后向指针，指向主对象，因为复合技术中的固定偏移技术不能正常工作了，这是由于 `tearoff` 和主对象不是被连续分配的。下面的类以 `tearoff` 的形式实现 `IBoat`：

```
Class CarBoat : public ICar {
    LONG m_cRef;
    CarBoat (void): m_cRef(0) {}
public:
    // IUnknown 方法
    STDMETHODIMP QueryInterface(REFIID, void**);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
    // IVehicle 方法
    STDMETHODIMP GetMaxSpeed(long *pMax);
    // ICar 方法
    STDMETHODIMP Brake(void);
    // 定义实现 IBoat 的
    struct XBoat : public IBoat {
        LONG m_cBoatRef;
        // 主对象以后向指针是显式成员
        CarBoat *m_pThis;
        inline CarBoat* This() { return m_pThis; }
        XBoat(CarBoat *pThis);
        ~XBoat(void);
        STDMETHODIMP QueryInterface(REFIID, void**);
```

```

STDMETHODIMP_(ULONG) AddRef(void);
STDMETHODIMP_(ULONG) Release(void);
STDMETHODIMP GetMaxSpeed(long *pval);
STDMETHODIMP Sink(void);
};

// 注意:类型 XBoat 没有数据成员
};

```

主对象的 **QueryInterface** 每次接到 **IBoat** 接口请求时，必须要动态地分配一个新的 tearoff：

```

STDMETHODIMP CarBoat::QueryInterface(REFIID riid,
                                      void **ppv)
{
    if (riid == IID_IBoat)
        *ppv = static_cast<IBoat*>(new XBoat(this));
    else if (riid == IID_IUnknown)
        *ppv = static_cast<IUnknown*>(this);
    :
    :
}

```

每次接收到 **IBoat** 接口请求时 **QueryInterface** 都会分配一个新的 tearoff。由于标准的 **QueryInterface** 实现会通过最终得到的结果指针来调用 **AddRef**，即：

```
( (IUnknown*) *ppv ) ->AddRef();
```

因此，只有 tearoff 会被 **QueryInterface** 直接调用 **AddRef**。很重要的一点是，只要 tearoff 还存在，那么主对象就必须留在内存中。为确保这一点，最容易的办法是让 tearoff 本身也代表一个未完结的引用。这可以在 tearoff 的构造函数和析构函数中完成：

```

CarBoat::XBoat::XBoat(CarBoat *pThis)
: m_cBoatRef(0), m_pThis(pThis) {
    m_pThis->AddRef();
}

CarBoat::XBoat::XBoat(void)
{
    m_pThis->Release();
}

```

如同采用复合技术来实现多个接口的情形一样，tearoff 的 **QueryInterface** 方法也需要维护对象的实体身份，所以它可以把 **QueryInterface** 委托给主对象的实现函数。然而，tearoff 也可以检测到对于它所实现的（一个或者多个）接口的查询请求，这时它只要简单地调用 **AddRef** 函数，然后返回指向自身的指针即可。示例代码如下：

```

STDMETHODIMP
CarBoat::XBoat::QueryInterface(REFIID riid,
                               void**ppv) {

```

```

    if (riid != IID_IBoat)
        return This()->QueryInterface(riid, ppv);
    *ppv = static_cast<IBoat*>(this);
    reinterpret_cast<IUnkown*> (*ppv)->AddRef();
    return S_OK;
STDMETHODIMP_(ULONG) CarBoat::XBoat::AddRef(void)
    return InterlockedIncrement(&m_cRef);
}

```

由于当 `tearoff` 不再需要的时候，它应该把自己销毁掉，所以它也应该为自己维护一个引用计数，然后当该计数到达 0 的时候它把自己删除掉。正如前面所指出的，`tearoff` 的析构函数将在删除自己之前，先释放主对象。代码如下：

```

STDMETHODIMP_(ULONG) CarBoat :: XBoat :: Release(void)
ULONG res = InterlockedDecrement(&m_cBoatRef);
if (res == 0)
    delete this; // dtor 释放主对象
return res;
}

```

与采用复合技术实现多个接口的情形一样，当 `tearoff` 需要访问主对象的状态时，不管它在什么地方，它都可以使用 `This()` 方法。两者的区别在于，`tearoff` 要求一个显式的后向指针，而普通的复合技术实现使用固定的偏移，所以 `tearoff` 可以为每个这样的复合类节约 4 个字节。

粗看之下，好像 `tearoff` 是所有可能的解决方案中最好的一种。当某个接口没有被使用的时候，对象不需要为该接口付出任何代价；当接口被用到的时候，对象间接地为 `tearoff` 付出 4 字节的代价。但是事实上，这种感觉是建立在一些不可靠的假设的基础上的。第一，当 `tearoff` 被使用的时候，它的代价不仅仅是 `vptr` 的 4 字节内存。`tearoff` 还需要一个后向指针和一个引用计数。<sup>1</sup> 第二，除非是使用了自定义的内存分配器，否则 `tearoff` 将要求至少 4 字节的附加内存，因为 C 运行库中 `malloc` 或者 `new` 操作符的实现用这些附加字节作为填补字节和（或者）头信息。这意味着，虽然当接口没被使用的时候对象确实节省了 4 个字节，但是当接口被使用之后，若使用自定义的内存分配器则 `tearoff` 至少要消耗 12 字节，若使用缺省的 `new` 操作符则 `tearoff` 需要 16 字节。如果客户很少请求这个接口，那么这样做可能是个合理的优化，特别是在“客户获得接口之后很快就会释放”的情况下。然而，如果客户在对象的整个生命周期过程中一直拥有 `tearoff` 接口的话，那么这种情况下 `tearoff` 接口的优势就不复存在。

不幸的是，`tearoff` 的故事还有更加糟糕的情况。以前面给出的实现为基础，如果对象接收到两个 `QueryInterface` 查询请求，并且它们请求同一个 `tearoff` 接口，那么对象将

---

<sup>1</sup> 如果实现者愿意限制客户对 `AddRef` 的使用的话，那么引用计数的代价可以被优化掉。但是在智能指针越来越流行的情况下（智能指针将会导致冗余[但是无害]的 `AddRef/Release` 对），这是非常危险的优化。

会创建 `tearoff` 的两份拷贝，因为一旦主对象把第一个 `tearoff` 返回给调用方之后，它就完全忘掉了指向这第一个 `tearoff` 的指针。这意味着现在 `tearoff` 需要消耗至少 24 到 32 字节的内存，因为内存中存在两个 `tearoff` 的 `vptr`，两个 `QueryInterface` 请求各一个。在客户释放掉这两个 `tearoff` 之前，这些内存无法被回收利用。两个 `QueryInterface` 请求查询一个“拥有对象生命周期全程”的指针，这种情形有特别的意义，因为这正是当一个对象被远程访问的时候所发生的情形。COM 的远程层将向对象两次 `QueryInterface` 请求同一个接口，并且在对象的整个生命周期过程中一直拥有这两个请求的结果。这使得 `tearoff` 对于这种可能会被远程访问的对象来说风险更大。

既然 `tearoff` 存在这些潜在的缺陷，那么我们自然就会有这样一个问题：什么时候使用 `tearoff` 是合适的呢？这个问题没有绝对的答案；然而，如果要想支持大量的、而又彼此排斥的接口，那么 `tearoff` 是一项非常好的技术。我们考虑前面的例子中给出的三个有关交通工具的接口，再加上 `ITruck`、`IMonsterTruck`、`IMotorcycle`、`IBicycle`、`IUnicycle`、`ISkateboard` 和 `IHelicopter` 接口，并且所有这些接口都从 `IVehicle` 派生。如果一个通用的交通工具类要能够支持所有这些接口，但是对于给定的实例，它只支持其中的一个接口，那么 `tearoff` 将会是非常合适的解决方案，主对象只要把指向第一个 `tearoff` 的指针缓存起来。主对象的类定义将如下所示：

```
class GenericVehicle : public IUnknown {
    LONG m_cRef;
    IVehicle *m_pTearOff; // 缓存 tearoff 指针
    GenericVehicle(void) : m_cRef(0), m_pTearOff(0) {}
// IUnknown 方法
    STDMETHODIMP QueryInterface(REFIID, void**);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
// 定义 tearoff 类
    class XTruck : public ITruck { ... };
    class XMonsterTruck : public IMonsterTruck { ... };
    class XBicycle : public IBicycle { ... };
};


```

在这个类中，当所有的接口都没被使用的时候，对象只付出附加的 4 字节，用于空的缓存指针。当某个 `QueryInterface` 请求到达，并且请求查询这 10 个交通工具接口之一时，对象就会为新的 `tearoff` 分配内存（仅此一次），并且缓存起来供以后使用：

```
STDMETHODIMP
GenericVehicle::QueryInterface(REFIID riid, void**ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IUnknown*>(this);
    else if (riid == IID_ITruck) {

```

```

    if (m_pTearOff == 0) // 还没有 tearoff, 制造一个
        m_pTearOff = new XTruck(this);
    if (m_pTearOff) // tearoff 存在
        return m_pTearOff->QueryInterface(rIID, ppv);
    else // 内存分配失败
        return (*ppv = 0), E_NOINTERFACE;
} else if (rIID == IID_IMonsterTruck) {
    if (m_pTearOff == 0) // 还没有 tearoff, 制造一个
        m_pTearOff = new XMonsterTruck(this);
    if (m_pTearOff) // tearoff 存在
        return m_pTearOff->QueryInterface (rIID, ppv);
    else // 内存分配失败
        return (*ppv = 0), E_NOINTERFACE;
} else ...
: : :
}

```

根据这里显示的 `QueryInterface` 实现，我们可以知道每个对象至多只有一个 `tearoff`。这意味着，如果客户不需要请求任何交通工具接口，那么对象将总共消耗 12 字节 (`IUnknown` vptr + 引用计数 + 指向 `tearoff` 的缓存指针)。如果有一个交通工具接口指针被请求到了，那么对象将消耗 24 到 28 个字节（原来的 12 个字节 + 从 `IVehicle` 派生的接口的 vptr + 引用计数 + 指向主对象的后向指针 + (可选) `malloc` 代价）。

这种情况下如果不使用 `tearoff`，那么类定义将如下所示：

```

class GenericVehicle
: public ITruck, public IHelicopter, public IBoat,
public ICar, public IMonsterTruck, public IBicycle,
public IMotorcycle, public ICar, public IPlane,
public ISkateboard {
    LONG m_cRef;
// IUnknown 方法
: : :
};

```

这个类的对象将总是消耗 44 字节 (10 个 vptr + 引用计数)。尽管通用交通工具类看起来显得有点夸张，但是 COM 永久接口却属于类似的情况，因为当前共有 8 个不同的永久接口可供使用，但是每个实例对象通常只暴露其中一个永久接口。然而，类的实现者往往无法预测对象的客户将会请求哪个接口。而且，这 8 个接口每个都需要支持一组不同的数据成员，以便能够正确地实现接口的方法。如果这些数据成员被设计成 `tearoff` 的成员，而不是主对象的成员，那么每个对象将只会分配其中一组数据成员。对于 `tearoff` 来说，这样的情形是最合适不过了，但是同样地，为了使 `tearoff` 技术更为有效，主对象有必要缓存一个指针，指向 `tearoff`。

## 4.10 二进制复合

复合技术和 tearoff 技术是两种源代码层次上的技术，都用于在 C++ 中实现 COM 对象。这两种技术都要求对象的实现者拥有每个复合类或者 tearoff 类的 C++ 源代码定义，以便能够在 QueryInterface 返回之前构造子对象的实例。在许多情况下，这是很合理的要求。然而，我们也可以把一个或者多个接口的可重用实现（reusable implementation）包装到一个二进制组件中，然后在跨越 DLL 边界之后仍然可以构造它的实例，而无需这个子组件的源代码，在有些情况下，这样做会更加方便。这将使得子组件可在更广泛的范围内被重用，而不需要如第 1 章所说的那样与源代码紧密地捆绑在一起。然而，要想使二进制复合对象或者 tearoff 成为可能，可重用组件需要参与构造对象实体的全部过程。

为了解决与“跨越组件边界唯一标识对象实体”有关的问题，请考虑下面 ICar 的简单实现：

```

class Car : public ICar {
    LONG m_cRef;
    Car(void) : m_cRef(0) {}
    STDMETHODIMP QueryInterface(REFIID, void **);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
    STDMETHODIMP GetMaxSpeed(long *pn);
    STDMETHODIMP Brake(void);
};

STDMETHODIMP Car::QueryInterface(REFIID riid,
                                Void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IUnknown*>(this);
    else if (riid == IID_IVehicle)
        *ppv = static_cast<IVehicle*>(this);
    else if (riid == IID_ICar)
        *ppv = static_cast<ICar*>(this);
    else
        return (*ppv = 0), E_NOINTERFACE;
    (* (IUnknown*) *ppv) ->AddRef();
    return S_OK;
}

//Car class 对象的 IClassFactory::CreateInstance
STDMETHODIMP CarClass::CreateInstance(IUnknown *pUnkOuter,
                                      REFIID riid, void **ppv) {

```

```

Car *pCar = new Car;
if (!pCar) return (*ppv = 0), E_OUTOFMEMORY;
pCar->AddRef();
HRESULT hr = pCar->QueryInterface(riid, ppv);
pCar->Release();
return hr;
}

```

这个类只是简单地使用了 `QueryInterface`、`AddRef` 和 `Release` 的实际实现而已。考虑下面第二个 C++ 类，它设法使用 `Car` 的实现作为二进制复合对象：

```

class CarBoat : public IBoat {
    LONG m_cRef;
    IUnknown *m_pUnkCar;
public:
    CarBoat(void);
    virtual ~CarBoat(void);
    STDMETHODIMP QueryInterface(REFIID, void **);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
    STDMETHODIMP GetMaxSpeed(long *pn);
    STDMETHODIMP Sink(void);
};

```

为了模拟复合类的情形，构造函数将创建 `Car` 子对象，而析构函数将释放指向子对象的指针：

```

CarBoat::CarBoat(void) : m_cRef(0) {
    HRESULT hr = CoCreateInstance(CLSID_Car, 0, CLSCTX_ALL,
                                  IID_IUnknown, (void**)&m_pUnkCar);
    assert (SUCCEEDED(hr));
}
CarBoat::~CarBoat(void) {
    if (m_pUnkCar)
        m_pUnkCar->Release();
}

```

在 `QueryInterface` 中，一个有趣的问题发生了：

```

STDMETHODIMP CarBoat::QueryInterface (REFIID riid,
                                      void **ppv) {
    if (riid == IID_IUnknown)
        *ppv = static_cast<IUnknown*>(this);
    else if (riid == IID_IVehicle)
        *ppv = static_cast<IVehicle*>(this);
    else if (riid == IID_IBoat)
        *ppv = static_cast<IBoat*>(this);
    else if (riid == IID_ICar) // 转发请求...
}

```

```

    return m_pUnkCar->QueryInterface(riid, ppv);
else
    return (*ppv = 0), E_NOINTERFACE;
((IUnknown*)*ppv) ->AddRef();
return S_OK;
}

```

因为 Car 子对象并不知道它是另一个对象实体的一部分，所以对于 IBoat 的 QueryInterface 查询请求，它将会失败。这意味着：

`QI(IBoat)->ICar`

将会成功，但是：

`QI(QI(IBoat)->ICar)->IBoat`

将会失败，结果导致 `QueryInterface` 不满足对称性条件。更进一步，通过 ICar 和 IBoat 接口指针查询 IUnknown 的 `QueryInterface` 请求将会返回不同的值，这就暗示着两个不同的对象实体身份，从而再次违反 IUnknown 协议，这就表明 CarBoat 对象不是一个合法的 COM 对象。

从二进制子组件组合成复合对象的思想听起来非常直接，但是实际上，COM 规范显式地描述了如何用一种可预测的方式来实现这种思想。直接通过 `QueryInterface` 把一个子组件暴露给客户的技术被称为 COM 聚合（COM aggregation）。COM 聚合是一组规则，它们定义了外部对象（聚合方）和内部对象（被聚合方）之间的关系。COM 聚合只是一组简单的 IUnknown 规则，它们允许多个二进制组件可以表现为单个 COM 对象实体。

COM 聚合决不是 COM 重用的首要工具。它远远不只是“简单地实例化一个对象，然后在另一个对象的方法实现中使用它的方法”。我们很少会直接把另一个对象的接口，当作自身对象的一部分暴露给客户。考虑下面的情形：

```

class Handlebar : public IHandlebar {...};
class Wheel : public IWheel. {};

class Bicycle : public IBicycle {
    IHandlebar * m_pHandlebar;
    IWheels *m_pFrontWheel;
    IWheels *m_pBackWheel;
};

```

对于 Bicycle 类来说，如果它的 `QueryInterface` 方法直接把 IHandlebar 或者 IWheels 接口递交给客户，那么这显然会违反直觉。`QueryInterface` 被用来表达“是一个（is-a）”关系，很明显，自行车（bicycle）不是一个车轮（wheel）或者车把（handlebar）。如果 Bicycle（自行车）设计者希望针对对象的这些方面提供直接的访问，那么 IBicycle 接

口应该显式地提供属性访问器（property accessor）用于这样的目的，举例如下：

```
[object, uuid (753A8A60-A7FF-11d0-8C30-0080C73925BA) ]
interface IBicycle : IVehicle {
    HRESULT GetHandlebar([out, retval] IHandlebar**pph);
    HRESULT GetWheels([out] IWheel **ppwFront,
                      [out] IWheel **ppwBack);
}
```

然后 Bicycle 实现过程只需简单地返回指向子对象的指针即可，如下：

```
STDMETHODIMP Bicycle::GetHandlebar(IHandlebar**pph) {
    if (*pph = m_pHandlebar)
        (*pph)->AddRef();
    return S_OK;
}

STDMETHODIMP Bicycle::GetWheels (IWheel **ppwFront,
                                 IWheel **ppwBack) {
    if (*ppwFront = m_pFrontWheel)
        (*ppwFront)->AddRef();
    if (*ppwBack = m_pBackWheel)
        (*ppwBack)->AddRef();
    return S_OK;
}
```

在使用了这样的技术之后，客户仍然可以直接访问子对象。然而，因为这些指针是通过显式的方法调用获得的，而不是通过 QueryInterface 获得的，所以这并不暗示在这些组件之间存在任何实体身份关系。

不再考虑这个例子中的情形，在其他一些情形下，我们希望能够提供一个接口的实现，使它能够被合并到另一个对象实体中。为了支持这种做法，COM 聚合要求内部对象（被聚合方）能够在“被作为另一个对象的一部分而创建”的时候接收到通知。这意味着，用于创建对象的创建函数必须要增加一个参数：一个 IUnknown 指针，它指向一个对象实体，在聚合情形下聚合对象的 QueryInterface、AddRef 和 Release 方法必须委托给这个指针所标识的对象。注意到 IClassFactory 接口的 CreateInstance 方法的定义：

```
HRESULT CreateInstance([in] IUnknown *pUnkOuter,
                      [in] REFIID riid, [out, iid_is(riid)] void **ppv);
```

这个方法（以及对应的 API 函数 CoCreateInstanceEx 和 CoCreateInstance）有双重负载，它可以同时支持独立对象的创建和聚合对象的创建。如果调用者在 CreateInstance 的第一个参数（pUnkOuter）中传递一个空指针（null），那么结果对象将是一个独立对象，惟一标识了它自己的实体身份。相反，如果调用者在第一个参数中传递了一个非空指针，那么结果对象将是 pUnkOuter 所指对象实体的聚合。在聚合的情形下，聚合对象

必须无条件地把所有的 `QueryInterface`、`AddRef` 和 `Release` 请求直接传递给 `pUnkOuter`。这对于维护对象实体身份是十分重要的。

给出了上面的函数原型之后，`CarBoat` 类需要作一点修改，以便符合聚合的规则：

```
CarBoat::CarBoat(void) : m_cRef(0) {
    //需要传递自身身份给创建函数
    //以通知 car 对象是一个聚合
    HRESULT hr = CoCreateInstance(CLSID_Car, this,
        CLSCTX_ALL, IID_IUnknown, (void**)&m_pUnkCar);
} assert(SUCCEEDED(hr));
```

`CarBoat` 的 `QueryInterface` 实现只是简单地把对 `ICar` 的接口请求传递给内部的聚合对象：

```
STDMETHODIMP CarBoat::QueryInterface(REFIID riid,
                                      void **ppv) {
    if (riid == IID_IUnknown)
        *ppv = static_cast<IUnknown*>(this);
    else if (riid == IID_ICar) // 转发请求...
        return m_pUnkCar->QueryInterface(riid, ppv);
    else if (riid == IID_IBoat)
        :
        :
```

理论上，这种做法应该能够正常工作，因为聚合对象将总是把所有后续的 `QueryInterface` 请求传回给主对象，从而可以保持对象实体身份的一致性。

在前述情形下，`Car` 类的 `CreateInstance` 方法返回一个指向外部对象的、从 `IUnknown` 派生的接口指针。如果这个接口指针只是简单地委托给外部对象的 `IUnknown`，那么它将没有办法做到下面两件事情：（1）通知聚合对象它将不再被需要了；（2）请求接口指针并交给主对象的客户。实际上，上面给出的 `QueryInterface` 实现将导致一个无限循环，因为外部对象委托给内部对象，而内部对象又传回给外部对象。

为了解决这个问题，最初被返回给外部对象的接口指针不能够委托回到外部对象的 `IUnknown` 实现。这意味着，支持 COM 聚合的对象必须有两个 `IUnknown` 实现。委托实现把所有的 `QueryInterface`、`AddRef` 和 `Release` 请求传递给外部实现。这是缺省实现，对象的所有 `vtbl` 都将引用这个实现，而且外部客户也会看到这个实现。对象还必须拥有另一个非委托的 `IUnknown` 实现，它只是被暴露给聚合情形下的外部对象。

可以有几种办法在一个单独的对象中提供两个不同的 `IUnknown` 实现。最直接的技术<sup>2</sup> 是使用复合类技术，并且使用一个数据成员来实现非委托的 `IUnknown` 方法。下面是一个可被聚合的 `Car` 实现：

<sup>2</sup> 我曾经相信使用方法染色(method coloring)技术来提供两个 `IUnknown` 实现是更好的技术。但是随着时间的推移，事实证明本书中给出的技术更好维护，而且效率也不差。

```

class Car : public ICar {
    LONG m_cRef;
    IUnknown *m_pUnkOuter;
public:
    Car(IUnknown *pUnkOuter);
    // 非委托 IUnknown 方法
    STDMETHODIMP InternalQueryInterface(REFIID,
                                         void **);
    STDMETHODIMP_(ULONG) InternalAddRef(void);
    STDMETHODIMP_(ULONG) InternalRelease(void);
    // 委托 IUnknown 方法
    STDMETHODIMP QueryInterface(REFIID, void **);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
    STDMETHODIMP GetMaxSpeed(long *pn);
    STDMETHODIMP Brake(void);
    // 映射 IUnknown vptr 到主对象中的
    // 非委托 InternalXXX 方法
    class XNDUnknown : public IUnknown {
        Car* This() { return (Car*)((BYTE*)this-
                                     offsetof(Car, m_innerUnknown)); }
        STDMETHODIMP QueryInterface(REFIID r, void**p)
        { return This()->InternalQueryInterface(r,p); }
        STDMETHODIMP_(ULONG) AddRef(void)
        { return This()->InternalAddRef(); }
        STDMETHODIMP_(ULONG) Release(void)
        { return This()->InternalRelease(); }
    };
    XNDUnknown m_innerUnknown; // 复合实例
};

```

这个对象的二进制布局结构如图 4.8 所示，这个类的委托方法非常简单，如下：

```

STDMETHODIMP_(ULONG) Car::AddRef(void)
{ return m_pUnkOuter->AddRef(); }

STDMETHODIMP Car::QueryInterface (REFIID riid,
                                 void **ppv)
{ return m_pUnkOuter->QueryInterface(riid, ppv); }
STDMETHODIMP_(ULONG) Car::Release(void)
{ return m_pUnkOuter->Release(); }

```

这些函数都是要被填充到对象的接口 vtbl 中的函数版本，所以，不管客户正在访问哪个接口，**IUnknown** 方法总是会委托给主对象实体。

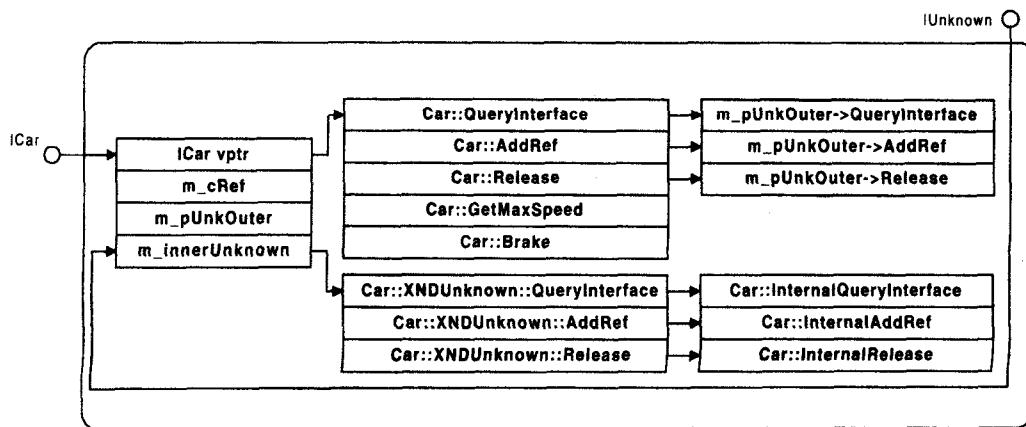


图 4.8 一个可被聚合的 Car

为了使这个对象能够同时用于“聚合”和“独立”两种情形下，对象的构造函数必须设置它的 `m_pUnkOuter` 数据成员，使得它在独立模式下指向自己的非委托 `IUnknown`，代码如下：

```

Car::Car(IUnknown *pUnkOuter)
{
    if (pUnkOuter) // 委托给 pUnkOuter
        m_pUnkOuter = pUnkOuter;
    else // 委托给非委托的自己
        m_pUnkOuter = &m_innerUnknown;
}
  
```

构造函数保证在任何一种情况下，`m_pUnkOuter` 都会指向正确的 `QueryInterface`、`AddRef` 和 `Release` 实现。

`QueryInterface`、`AddRef` 和 `Release` 的正常非委托实现非常普通，正如预期的那样：

```

STDMETHODIMP Car::InternalQueryInterface(REFIID riid,
                                         void **ppv) {
    if (riid == IID_IUnknown)
        *ppv = static_cast<IUnknown*>(&m_innerUnknown);
    else if (riid == IID_IVehicle)
        *ppv = static_cast<IVehicle*>(this);
    else if (riid == IID_ICar)
        *ppv = static_cast<ICar*>(this);
    else
        return (*ppv = 0), E_NOINTERFACE;
    (*IUnknown * *ppv)->AddRef();
    return S_OK;
}
  
```

```

STDMETHODIMP_(ULONG) Car:: InternalAddRef(void)
{
    return InterlockedIncrement (&m_cRef);
}

STDMETHODIMP_(ULONG) Car:: Internal Release (void)
{
    ULONG res = InterlockedDecrement (&m_cRef);
    if (res == 0)
        delete this;
    return res;
}

```

这三个方法唯一独特之处（除了它们的名字之外）在于，`InternalQueryInterface` 在接收到 `IUnknown` 请求时，返回指向非委托 `IUnknown` 接口的指针。这是 COM 规范规定的、必须要遵守的要求。

最后针对 `Car` 的创建函数也需要作一些修改，以便支持 COM 聚合：

```

STDMETHODIMP CarClass:: CreateInstance(IUnknown *pUnkOuter,
                                      REFIID riid, void **ppv) {
    // 验证聚合体仅请求 IUnknown 作为
    // 初始接口
    if (pUnkOuter != 0 && riid != IID_IUnknown)
        return (*ppv = 0), E_INVALIDARG;

    // 创建新的对象/聚合

    Car *p = new Car(pUnkOuter);
    if (!p) return (*ppv = 0), E_OUTOFMEMORY;
    // 返回结果指针
    p->InternalAddRef();
    HRESULT hr = p->InternalQueryInterface(riid, ppv);
    p->InternalRelease();
    return hr;
}

```

注意，这里用到了 `QueryInterface`、`AddRef` 和 `Release` 的非委托版本。如果被创建的是一个独立对象，那么这肯定是正确的。如果被创建的是一个聚合对象，那么这是必需的，这样可以确保是内部对象的引用计数增 1，而不是外部对象的引用计数增 1。同时也请注意，外部对象必须请求 `IUnknown` 作为初始接口。这是 COM 规范的要求。如果外部对象可以请求任何类型的初始接口，那么内部对象将必须保持两套 `vptr`，一套委托它的 `QueryInterface`、`AddRef` 和 `Release` 实现，而另一套不委托这些实现。在限制了初始接口必须是 `IUnknown` 之后，对象的实现者只需要隔离一个 `vptr` 作为非委托 `IUnknown` 即可。

在进行 COM 聚合编程时，在引用计数方面可能会发生危险。注意到，内部对象的引用计数复制了指向外部控制对象的指针，但是没有调用 `AddRef`。在这种特殊的情况下

下，调用 AddRef 是被禁止的，因为如果两个对象都调用了对方的 AddRef，那么就会构成一个永远也无法打破的“环”。有关聚合的引用计数规则要求：外部对象拥有一个已经被引用计数的指针，指向内部对象的非委托 IUnknown（这正是对象的创建函数所返回的指针）。内部对象拥有一个未被引用计数的指针，指向外部控制对象的 IUnknown。从技术上讲，这种关系已经被 COM 的引用计数条款所覆盖。一般来讲，“使用未被引用计数的指针”这种技术不应该被使用，因为当对象被远程访问的时候，这种技术是不可能奏效的。处理引用计数“环”的一种有效办法是引入中间对象实体，它的引用计数不会影响这两个对象实体的生命周期。

在 COM 聚合编程过程中，另一个问题发生在当内部对象与外部对象需要进行通信时。为了使内部对象能够与外部对象进行通信，它必须通过控制 IUnknown 指针调用 QueryInterface。然而，这个 QueryInterface 请求将会在结果得到的指针上调用 AddRef，它将会作用在外部对象的引用计数上。如果内部对象要把这个指针保留作为自己的数据成员，那么一个环又形成了，因为内部对象已经隐式地在外部对象上作了引用计数。这意味着内部对象必须要使用两种策略之一。内部对象可以按需获取指针，然后释放指针，只在它需要的时候才拥有这个指针，示例如下：

```
STDMETHODIMP Inner::MethodX(void) {
    ITruck *pTruck = 0;
    // 此调用后，外部对象是 AddRefed...
    HRESULT hr = m_punkOuter->QueryInterface(IID_ITruck,
                                                (void**)&pTruck);
    if (SUCCEEDED(hr)) {
        pTruck->ShiftGears();
        pTruck->HaulDirt();
    }
    // 释放外部对象指针
    pTruck->Release();
}
}
```

第二种技术是在初始化时刻获取指针，然后在获取之后立即释放对应的外部对象。

```
HRESULT Inner::Initialize(void) {
    // 此调用后，外部对象是 AddRefed...
    HRESULT hr = m_punkOuter->QueryInterface(IID_ITruck,
                                                (void**)&m_pTruck);
    // 释放外部对象指针
    // 但在后面对象机构函数中不释放它
    if (SUCCEEDED(hr))
        m_pTruck->Release();
}
}
```

这项技术能够工作，因为内部对象的生命周期是外部对象生命周期的一个子集而已。这意味着理论上 m\_pTruck 将总是指向一个有效的对象。当然，如果外部对象用 tearoff

技术来实现 `ITruck` 接口，那么所有的赌注都没了，因为 `Release` 调用将会销毁 `tearoff`。

那些聚合了其他对象的对象也需要意识到与“内部聚合对象请求接口指针”相关的问题。除了前面提到的与 `tearoff` 有关的告诫之外，其他还存在一些与对象的稳定性有关的危险。当客户访问一个对象的时候，对象必须处于一种稳定状态。特别地，它的引用计数必须不为 0。一般来讲，这不是一个问题，因为客户只能通过 `QueryInterface` 才能获取接口指针，而 `QueryInterface` 总是在返回之前先执行 `AddRef`。然而，如果一个对象在它的构造函数中创建了一个聚合对象，并且它的引用计数仍然为 0，那么上面显示的内部对象初始化代码有可能会执行外部对象的最终释放操作，从而强迫外部对象提前销毁自己。为了避免这个问题，如果一个对象要聚合其他的对象，那么它在创建聚合对象之前，应该临时把自己的引用计数增加到 1。代码如下：

```
Outer: :Outer(void) {
    ++m_cRef; // 保护不删除
    CoCreateInstance(CLSID_Inner, this, CLSCTX_ALL,
                     IID_IUnknown, (void**)&m_pUnkInner);
    --m_cRef; // 允许删除
}
```

这项稳定性技术可以阻止外部对象提前销毁自己（当内部对象释放它在初始化过程中获得的指针时）。这项技术非常通用，大多数的 COM 编程框架都提供了一个显式的可重载方法，在一对“`increment/decrement`（引用计数增 1/引用计数减 1）”之间执行。MFC 把这个方法称为“`CreateAggregates`”，ATL 把这个方法称为“`FinalConstruct`”。

因为前面给出的、实现可聚合对象的技术不需要任何附加的基类加入到 C++ 类中，所以另一种形式的 `IMPLEMENT_UNKNOW` 宏可以透明地实现分叉的两个 `IUnknown`。原来的类定义为：

```
class Car : public ICar {
    Car(void);
    IMPLEMENT_UNKNOW(Car)
    BEGIN_INTERFACE_TABLE(Car)
        IMPLEMENTS_INTERFACE(ICar)
        IMPLEMENTS_INTERFACE(IVehicle)
    END_INTERFACE()
    // IVehicle 方法
    STDMETHODIMP GetMaxSpeed(long *pn);
    // ICar 方法
    STDMETHODIMP Brake(void),
};
```

可以简单地被转变为：

```
class Car : public ICar {
    Car(void);
    // 表示需要聚合
```

```

IMPLEMENT_AGGREGATABLE_UNKNOWN(Car)
BEGIN_INTERFACE_TABLE(Car)
    IMPLEMENTS_INTERFACE(ICar)
    IMPLEMENTS_INTERFACE(IVehicle)
END_INTERFACE()
// IVehicle 方法
STDMETHODIMP GetMaxSpeed(long *pn);
// ICar 方法
STDMETHODIMP Brake(void);
};

IMPLEMENT_AGGREGATABLE_UNKNOWN 宏的内联展开（定义）包含在本书配
套的源代码中。

```

## 4.11 包容

并不是所有的类都是可聚合的。为了把一个不可被聚合的类（nonaggregatable class）暴露成为另一个对象实体的一部分，外部对象必须要显式地把方法调用传递给内部对象。这种技术通常被称为“COM 包容（COM containment）”。如图 4.9 所示，包容并不要求参与到内部对象的任何一部分。然而，它要求外部对象提供“内部对象已经实现的接口”的每一个方法的实现。这些外部实现只是简单地把客户的请求传递给内部对象。COM 包容并不特别关心 COM 的实体身份规则，因为客户永远也不会直接访问内部对象，因此内部对象也不会直接混淆外部对象的类型层次。尽管 COM 包容也是 COM 规范的一部分，但是它并不要求特殊的编程技术。实际上，被包容的对象并不能够检测到“外部对象正在把实际客户的方法调用传递给它”。

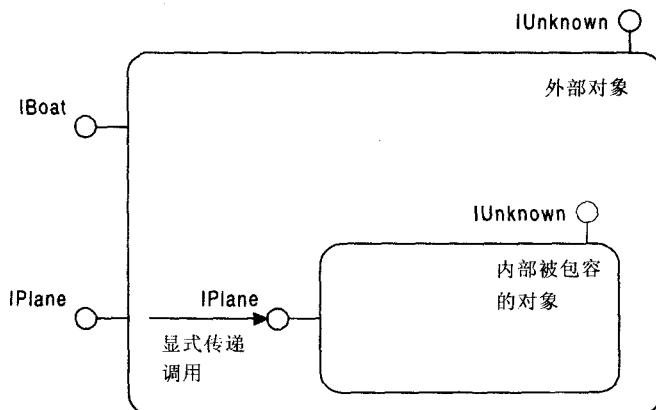


图 4.9 COM 包容

## 4.12 我们走到哪儿了？

本章讨论了 COM 的实体身份法则。这些法则定义了 COM 对象的真正含义是什么。COM 的实体身份法则使得对象实现者在分割对象实现时具有极大的灵活性。本章展示了复合技术（composition），它可以实现以每个接口（per-interface）为基础的引用计数。本章还展示了 tearoff 技术，它可以降低 vptr 的尺寸，并且更加有效地管理对象的状态。然后本章显示了聚合技术，它可以把两个或者更多个二进制组件组合成为单个对象实体。这些技术都可以把多个对象表现为单个 COM 对象。每项技术都有它自己的优势，不论使用哪一项技术（或者全部技术），对于客户来说，对象所使用的技术都是不可见的。

# 第 5 章

## 套 间

```
STDMETHODIMP CmyClass::MethodX(void) {
    EnterCriticalSection(&m_cs);
    if (TryToPerformX() == false)
        return E_UNEXPECTED;
    LeaveCriticalSection(&m_cs);
    Return S_OK;
}
Anonymous, 1996
```

上一章讨论了 COM 实体身份的基础，并且正式定义了 COM 对象与随机内存之间的区别。以外还提出了 IUnknown 的规则，以及一些“能够应用这些规则为对象提供极大灵活性”的技术。本章将进一步提炼 COM 实体身份的定义，同时充分考虑操作系统的基本原语（比如线程和进程），以及分布式访问特性。系统原语（primitive）与分布式特性的结合构成了 COM 远程结构的基础。

### 5.1 再谈接口和实现

有的开发人员广泛使用多线程程序设计技术，他们利用操作系统提供的线程同步原

语，能够编写出复杂程度惊人的软件。有的开发人员更加专注于解决与特定领域相关的问题，不希望分出精力来编写线程安全代码。有的开发人员对于多线程编程非常恐惧，因为许多窗口系统（包括 Windows）对于线程原语和窗口原语之间的关系有非常严格的规则。另外还有一类开发人员需要使用大量的、遗留下来的类库，而这些类库又与多线程有冲突，它们根本不能容忍同时有多个线程来访问。所有这四类开发人员都需要相互之间能够使用其他人的对象，而无需重新构造它们的线程策略以容纳所有情况。为了能够透明地使用一个对象而不必关心它是否感知线程，COM 把对象的并发性限制作为另一个实现细节，客户无需知道这些细节。为了解除客户与对象的并发性和重入限制之间的关联，COM 提供了一个非常规范的抽象概念，它建立起对象与进程和线程之间相互关系的模型。这个抽象概念的正式名称为套间（apartment）。<sup>1</sup> 套间定义了一组对象的逻辑组合，这些对象共享同一组并发性和重入限制。每个 COM 对象都属于某一个套间，然而，一个套间可以被多个对象所共享。对象所属的套间也是这个对象实体属性的一部分。

套间既不是进程，也不是线程，然而，套间拥有进程和线程的某些特性。每一个使用 COM 的进程都有一个或者多个套间；然而，套间只能包含在一个进程中。这意味着每个使用 COM 的进程都至少有一组对象共享同样的并发性和重入要求；然而，驻留在同一个进程中的两个对象有可能属于两个不同的套间，因此，它们可能具有不同的并发性和重入限制。这条原则使不同线程意识的库可以在同一个进程内部和平共处。

同一时刻，一个线程只在一个套间中执行。一个线程要想使用 COM，它必须先进入一个套间中。当线程进入到套间中时，COM 把关于这个套间的信息保存在线程局部存储（TLS，thread local storage）中，在线程退出这个套间之前，这一信息一直与线程联系在一起。COM 规定，只有运行在对象的套间中的线程才能够访问该对象。这意味着如果一个线程与对象位于同一个进程中，那么虽然这个线程能够看到、也可以访问到对象所占有的内存，但是它有可能被禁止访问这个对象。COM 定义了一个 HRESULT（RPC\_E\_WRONG\_THREAD），当外部套间直接访问某些特定的系统级对象时，对象将会返回这个错误值。用户定义的对象也可以返回这个 HRESULT 值，这是合法的；然而，很少有开发人员愿意进入这个层次来保护他们的对象能正确地使用。

Windows NT 4.0 发布的 COM 版本定义了两种类型的套间：多线程套间（MTA，multithreaded apartment）和单线程套间（STA，singlethreaded apartment）。每个进程至多只有一个 MTA；然而，一个进程可以包含多个 STA。顾名思义，MTA 中多个线程可以并发执行，但是 STA 中只有一个线程可以执行。更精确地说，对于给定的 STA，永远只有一个线程可以在其内执行，这就意味着，不仅驻留在 STA 中的对象永远也不可能被并发访问，而且只有一个特定的线程可以执行对象的方法。这种线程相依性（thread affinity）使得对象的实现者可以安全地把多个方法调用之间的中间状态保存在 TLS 中，也可以在跨越方法调用时拥有与线程相关的锁（比如 Win32 的临界区[critical section]和

<sup>1</sup> 套间(apartment)是比较新的术语，COM 规范原先把它称为执行环境(execution context)。

互斥体(mutex)。

在使用基于 MTA 的对象时，以上这些做法都将导致灾难发生，因为无法保证指定的方法调用会在哪个线程上执行。STA 的缺点是，不管这个套间有多少个对象，同一时刻它只允许执行一个方法调用。在 MTA 中，根据当前的需求，线程可以被动态申请，与套间中的对象数目完全没有关系。为了仅使用单线程套间建立可并发访问的服务器进程，很显然多个套间是必需的，但是稍不注意就会造成过度的线程负担。而且，以 STA 为基础的服务器进程中，并发程度也不可能超过进程中对象的总数。如果服务器进程只包含少量的、粗粒度（Coarse-grained）的对象，那么即使每个对象都驻留在它自己私有的 STA 中，也只有少量的线程会被用到。

将来版本的 COM 可能会引入第三种类型的套间，租用方式的线程套间（RTA，rentalthreaded apartment）。与 MTA 一样，RTA 也允许多个线程进入一个套间中。但是与 MTA 不同的是，当线程进入到 RTA 之后，它获得了一把套间范围内的锁（因此，它租用了套间），这把锁可以阻止其他的线程并发进入到这个套间中。当线程退出 RTA 时，这把套间范围的锁被释放，从而允许下一个线程进入到套间中来。从这个角度来看，RTA 好像就是 MTA，只不过所有的方法调用都是串行方式的。这使得 RTA 更加适合于那些“不知道是否线程安全”的类。尽管 STA 中的所有调用也都是串行的，但是基于 RTA 的对象与此不同，它们没有线程相依性；这就是说，任意一个线程都可以在 RTA 中执行，而不仅仅是创建该套间的初始线程。由于基于 RTA 的对象不具有线程相依性，这使得它们比基于 STA 的对象更加灵活、更加有效，因为任何线程都可以进入到对象的 RTA 中，并调用对象的方法。在本书写作的时候，如何创建和进入 RTA 套间的细节还没有最终确定。有关更多细节，请参照 SDK 文档。

在调用 CreateProcess 或者 CreateThread 时，操作系统会首先创建一个线程，新创建的线程没有与它相关联的套间。<sup>2</sup> 在使用 COM 之前，新线程必须调用下列三个 API 函数之一，以便进入套间。函数如下：

```
HRESULT CoInitializeEx(void *pvReserved, DWORD dwFlags);
HRESULT CoInitialize(void *pvReserved);
HRESULT OI eInitialize(void *pvReserved)
```

以上列出的这三个 API 函数的第一个参数被保留，必须为 0。

CoInitializeEx 是最底层的 API 函数，它允许调用者指定将要进入哪种类型的套间。为了进入到进程范围内的 MTA，调用者必须使用 COINIT\_MULTITHREADED 标志：

```
HRESULT hr = CoInitializeEx(0, COINIT_MULTITHREADED);
```

要想进入新建的 STA，调用者必须指定 COINIT\_APARTMENTTHREADED 标志：

---

<sup>2</sup> 在 Windows NT 5.0（即 Windows 2000）平台下，这一事实将会发生变化。请参阅 SDK 文档以获得更多的信息。

```
HRESULT hr = CoInitializeEx(0, COINIT_APARTMENTTHREADED);
```

进程内部所有调用 `CoInitializeEx` 并使用 `COINIT_MULTITHREADED` 标志的线程，都在同一个套间中执行。如果线程在调用 `CoInitializeEx` 时使用了 `COINIT_APARTMENTTHREADED` 标志，那么它将执行在私有的套间中，任何其他的线程都不会进入这个套间。`CoInitialize` 是一个遗留下来的函数，它只是简单地用 `COINIT_APARTMENTTHREADED` 标志调用 `CoInitializeEx` 函数。`OleInitialize` 首先调用 `CoInitialize`，然后对几个子系统进行初始化，OLE 应用将会用到这些子系统，例如 OLE 拖-放（OLE Drag and Drop）和 OLE 剪贴板（OLE Clipboard）。一般来说，如果不需要这些高层服务的话，那么应该优先考虑调用 `CoInitialize` 或者 `CoInitializeEx`。

这三个 API 函数中，每一个函数都可以被同一个线程调用多次。在每个线程上，第一个调用将返回 `S_OK`。后续的调用则会重新进入到相同的套间中，并且返回 `S_FALSE`。对于每一个成功的 `CoInitialize` 或者 `CoInitializeEx` 调用，同一个线程中也必须要有对应的 `CoUninitialize`。对于每个成功的 `OleInitialize` 调用，同一个线程中也必须要有对应的 `OleUninitialize` 调用。这些终结函数具有非常简单的原型：

```
void CoUninitialize(void);
void OleUninitialize(void);
```

在线程或者进程结束之前，如果这些函数调用失败，那么资源的再利用可能会被延后。一旦线程进入一个套间，再想使用 `CoInitializeEx` 来改变套间的类型，这将是非法的。这种做法将会导致 `HRESULT` 值为 `RPC_E_CHANGED_MODE`。然而，一旦线程已经使用 `CoUninitialize` 完全退出了原来的套间，那么它就可以再次调用 `CoInitializeEx`，进入另一个套间。

## 5.2 对象、接口和套间

客户想要调用对象上的方法，而对象只想把它们的方法暴露给客户。对象的并发性限制可能会不同于客户套间所隐含的并发性限制，这是对象的实现细节，客户不应该关心这些细节。同样地，如果对象的实现者选择把一个对象实现只安装在少量的主机上，而没有安装在客户所在的机器上，那么这又是一个客户不该关心的实现细节。然而，无论哪种情况下，对象必须驻留在一个不同于客户的套间中。

从程序设计的角度来看，套间的所属关系是接口指针的一个属性，而不是对象的属性。当 COM API 调用或者某个方法调用返回一个接口指针时，调用该 API 或者方法的线程决定了“结果接口指针将属于哪个套间”。如果调用返回一个指向实际对象的指针，那么对象本身必定驻留在调用线程的套间中。通常情况下，对象不可能驻留在调用者的

套间中，可能是因为对象已经位于另一个进程中或者另一台主机上，也可能因为对象的并发性要求与客户的套间不兼容。在这些情况下，客户会收到一个指向代理（proxy）的指针。

在 COM 中，代理也是一个对象，它在语义上等同于另一个套间中的某个对象。从某种意义上讲，代理表达了位于另一个套间中的另一个对象实体。代理与它所表达的对象暴露了同样的一组接口，然而，代理在实现每个接口的方法时，只是简单地把调用传递给它所表达的对象，从而保证对象的方法总是在对象的套间中执行。不管客户接收到的是指向实际对象的指针，还是指向代理的指针，它从 API 调用或者方法调用接收到的任何一个接口指针，对于调用者的套间来说，总是有效的。

对象将在哪种类型的套间中执行，这是由对象实现者来决定的。正如第 6 章将要讨论的，进程外的服务器（out-of-process server）通过调用 CoInitializeEx，并指定适当的参数，以此显式确定其套间类型。对于进程内服务器（in-process server）来说，我们需要其他的方法来指定对象的套间类型，因为在对象被创建之前，客户已经调用过 CoInitializeEx 了。为了允许进程内服务器可以控制它们的套间类型，COM 允许每个 CLSID 有它自己不同的线程模型，我们只需在本地注册表中用 ThreadingModel 名字值说明它的线程模型，如下：

```
[HKCR\CLSID\{96556310-D779-11d0-8C4F-0080C73925BA}\InprocServer32]
@="C:\racer.dll"
ThreadingModel="Free"
```

DLL 中的每个 CLSID 可以有自己独立的 ThreadingModel。在 Windows NT 4.0 下，COM 允许每个 CLSID 可以有四个可能的 ThreadingModel 值。ThreadingModel=“Both”表示这个类既可以在 MTA 中执行，也可以在 STA 中执行。ThreadingModel=“Free”表示这个类只能在 MTA 中执行。ThreadingModel=“Apartment”表示这个类只能在 STA 中执行。如果某个 CLSID 不存在对应的 ThreadingModel 值，那么这表示该类只能在主 STA 中运行。主 STA 是指这个进程中第一个被初始化的 STA。

如果客户的套间与 CLSID 的套间模型兼容，那么所有针对该 CLSID 的进程内激活请求都将直接在客户的套间中构造对象的实例。这是迄今为止效率最高的情形，因为不需要中间代理对象。<sup>3</sup> 如果客户的套间与 CLSID 的套间模型不兼容，那么针对该 CLSID 的进程内激活请求将迫使 COM 在另一个套间中构造对象（对客户是透明的），然后给客户返回一个代理。如果是基于 STA 的客户激活 ThreadingModel=“Free”的类，那么类对象（以及后续的实例）将在 MTA 中执行。如果是基于 MTA 的客户激活 Threading Model=“Apartment”的类，那么类对象（以及后续的实例）将在 COM 创建的 STA 中执行。无论哪种类型的客户要想激活基于主 STA 的类（也就是没有定义 ThreadingModel

<sup>3</sup> 由于线程切换的原因，跨越套间的方法调用所需要的性能代价，有可能是套间内部方法调用所需代价的几千倍。

值的类），类对象（以及后续的实例）必须在进程的主 STA 中执行。如果客户刚好是主 STA 线程，那么对象可以直接被客户访问到。否则的话，客户将得到一个代理。如果进程中不存在 STA 的话（也就是说，没有一个线程用 COINIT\_APARTMENTTHREAD 标记来调用 CoInitializeEx），那么 COM 将会创建一个新的 STA，作为该进程的主 STA。

如果类的实现者没有为他们的类标记出相应的线程模型，那么它们可以完全忽略线程因素，因为它们的 DLL 只可被一个线程访问，即主 STA 线程。如果实现者把他们的类标记为支持某一种显式的线程模型，这就暗示着进程中可能会有多个套间，每个套间都有可能包含该类的实例。由于这个原因，实现者必须要保护所有可被多个实例共享的资源，以避免并发访问。这意味着所有的全局变量和静态变量必须要被保护起来，可以使用适当的某种线程同步原语。对于 COM 进程内服务器来说，用来记录服务器生命周期的全局锁计数器必须要用 InterlockedIncrement 和 InterlockedDecrement 保护起来，正如前面第 3 章所演示的。其他与服务器相关的状态也必须要被保护起来。

如果类实现者把他们的类标记为 ThreadingModel=“Apartment”，那么这相当于声明了这样的事实：在对象的生命周期内，只有一个线程可以访问它们。这隐含着实现者不需要保护实例的状态，正如前面所提到的，只有被多个实例共享的状态才需要被保护。如果类实现者把他们的类标记为 ThreadingModel=“Free”或者 ThreadingModel=“Both”，那么这相当于声明了类的实例可能运行在 MTA 中，这意味着类的每一个实例都有可能被多个线程并发访问。由于这样的原因，实现者必须保护每个实例用到的所有资源，以避免并发访问。这不仅适用于共享的静态变量，同样也适用于实例的数据成员。对于基于堆（heap）的对象来说，这暗示着引用计数器数据成员必须要用 InterlockedIncrement 和 InterlockedDecrement 保护起来，如第 2 章所示。所有其他的、特定于这个类的实例状态也需要被保护起来。

粗看起来，ThreadingModel=“Free”存在的必要性并不很明显，因为“在 MTA 中运行”这样的要求往往可以被看作 STA 兼容性要求的一个超集。如果对象实现者计划创建一个辅助线程（worker thread），这个辅助线程将会访问这个对象，那么他应该避免在 STA 中创建这样的对象，这将会有极大的好处。因为辅助线程不可能进入到对象所在的 STA 中，因此它必须运行在不同的套间中。如果一个类被标记为 Threading Model=“Both”，并且激活请求来自于基于 STA 的线程，那么对象将运行在 STA 中。这意味着辅助线程（它将运行在 MTA 中）必须用跨套间的方法调用才能访问对象，这比套间内部的方法调用要低效得多。然而，如果一个类被标记为 ThreadingModel=“Free”，那么任何基于 STA 的激活请求都强制新的实例被创建在 MTA 中，任何一个辅助线程都可以直接访问这样的对象实例。这意味着，当基于 STA 的客户调用对象的方法时，性能将有所下降；然而，辅助线程将会获得更好的性能。如果对象被辅助线程访问的次数，要高于被实际的 STA 客户访问的次数，那么这是一个合理的平衡策略。把 COM 的规则放宽，使得有些对象可被多个套间直接访问而不会引起错误，这实在很吸引人。然而，在一般情况下，这并不是真的，特别是对于那些“使用其他对象来执行自己的工作”的对象来说，更是如此。

### 5.3 跨套间访问

为了允许对象可以驻留在不同于客户的套间中，COM 允许从一个套间中引出接口，并且引入到另一个套间中。使对象的套间之外也能看到对象的接口，这是引出（export）接口。使套间内部也能够看到外部的接口，这是引入（import）接口。当接口指针被引入的时候，结果得到的接口指针指向一个代理，引入套间中的任何一个线程都可以合法地访问这个代理。<sup>4</sup> 把控制传回给对象的套间，这是代理的任务，它保证所有的方法调用都会在正确的套间中执行。从一个套间到另一个套间的控制传递过程被称为方法远程传递（method remoting），这也是 COM 中所有的跨线程、跨进程、以及跨主机通信的基础。

缺省情况下，方法远程传递过程使用了 COM 对象 RPC(ORPC)通信协议。COM ORPC 是一个轻量级协议，它位于 MS-RPC（派生自 DCE 的协议）层次之上。MS-RPC 是一套协议独立的通信机制，对它进行扩展，可以支持新的传输协议（通过可动态装载的传输 DLL）和新的认证包（通过可动态装载的安全支持提供者 DLL[Security Support Provider DLL]）。COM 根据引入套间和引出套间的类型和亲近关系，选择最为有效的传输协议。在跨主机通信的时候，COM 优先选择 UDP（User Datagram Protocol）协议，虽然它也支持其他的常用网络协议。<sup>5</sup> 在本地通信的时候，COM 使用几种传输协议之一，每一种协议都针对特定的套间类型进行了优化。

COM 使用一种被称为列集（marshal）的技术，允许接口指针可被跨越套间边界传递出去。列集一个接口指针，实际上只是简单地把接口指针转换成一个可被传输的字节流，字节流的内容唯一标识了对象和它所拥有的套间。这个字节流正是接口指针被列集后的状态，它使得任何一个套间都可以引入这个接口指针，然后调用该对象上的方法。注意，因为 COM 只处理接口指针，而不是对象本身，所以接口指针被列集后的状态并没有代表对象的状态，而是该对象的一个“与套间无关”的引用的序列化状态而已。这些被列集后的对象引用简单地包含了建立连接的信息，它与对象的状态是完全独立的。

接口指针的列集过程往往被隐式完成，作为 COM 普通操作的一部分。当一个进程内激活请求碰到不兼容的线程模型的时候，COM 隐式地把接口指针从对象的套间中列集出来，然后散集（unmarshal）成客户套间的一个代理。当进程外或者远程激活请求的时候，COM 也会把结果指针从对象的套间中列集出来，然后为客户散集一个代理。当

<sup>4</sup> 如果引入套间就是对象所属的套间，那么不需要使用代理，而且被引入的指针直接指向对象。

<sup>5</sup> 由于建立 TCP 连接所需要的额外负担，所以 UDP 优先于 TCP。COM 与 DCE RPC 一样，把它的安全信息和协议同步信息放在“用于传输第一个 RPC 请求的数据包”的包头上。因为基于 COM 的系统大都倾向于“建立和撤销大量的短暂连接”，所以 UDP 是更好的选择。在使用诸如 UDP 这样的包传输协议时，RPC 运行库重新实现了 TCP 的纠错(error correction)和流/拥塞控制(flow/congestion control)算法。

在代理对象上执行方法的时候，作为方法参数被传递的所有接口指针都将被列集，目的是为了让这些对象引用可同时被用于客户套间和对象套间。偶尔我们也有必要显式地把接口指针从一个套间列集到另一个位于激活请求（或者方法调用）的环境之外的套间中。为了支持这种做法，COM 提供了一个底层 API 函数 CoMarshalInterface，用于显式地列集接口指针。

CoMarshalInterface 以一个接口指针作为输入，然后把该指针序列化之后的表示写到一个由调用者提供的字节流中。然后这个字节流可被传递到另一个套间中，在那里，CoUnmarshalInterface API 函数使用这个字节流，返回一个接口指针，在语义上它完全等同于原来的对象。执行 CoUnmarshalInterface 调用的套间可以通过这个接口指针，合法地访问原来的对象。在调用 CoMarshalInterface 的时候，调用者必须指明期望中的引入套间离它有多远。COM 定义了一个枚举类型，用来表达这个距离，如下：

```
typedef enum tagMSHCTX {
    MSHCTX_INPROC = 4           // 进程内/同一主机
    MSHCTX_LOCAL = 0,           // 进程外/同一主机
    MSHCTX_NOSHAREDMEM = 1,     // 16/32 位/同一主机
    MSHCTX_DIFFERENTMACHINE = 2, // 远程
} MSHCTX;
```

指定一个超过实际使用要求的距离值也是合法的，但是我们应该尽可能地使用正确的 MSHCTX，从而获得更好的效率。CoMarshalInterface 也允许调用者使用下面的列集标志来指定列集语义：

```
typedef enum tagMSHLFLAGS {
    MSHLFLAGS_NORMAL,          // 列集一次，散集一次
    MSHLFLAGS_TABLESTRONG,      // 列集一次，散集多次
    MSHLFLAGS_TABLEWEAK,        // 列集一次，散集多一次
    MSHLFLAGS_NOPING = 4,       // 制止分布式垃圾收集
} MSHLFLAGS;
```

上面的常规列集（normal marshaling，有时也称调用列集[call marshaling]）是指，被列集的对象引用只能被散集一次，如果需要多个代理的话，必须多次调用 CoMarshalInterface。表格列集（table marshaling）是指被列集的对象引用可被散集 0 次或者多次，而无需再次调用 CoMarshalInterface。表格列集的细节将在本章后文进一步讲述。

为了使接口指针可被列集到各种介质中，CoMarshalInterface 通过一个由调用者提供的 IStream 接口，对接口指针进行序列化操作。IStream 接口对任意的 I/O 设备进行抽象，建立其接口模型，它包含 Read 和 Write 方法。CoMarshalInterface 只是简单地调用（由调用者提供的）IStream 接口的 Write 方法，而不考虑字节流实际被保存在哪里。调用者通过下面的 CreateStreamOnHGlobal API 函数可以获得针对基本内存的 IStream 封装，函数原型如下：

```

HRESULT CreateStreamOnHGlobal(
    [in] HGLOBAL hglobal, // 将 null 传给 autoalloc
    [in] BOOL bFreeMemoryOnRelease,
    [out] IStream **ppStm);

```

有了 IStream 的语义之后，下面的代码片断中的 memcpy 可以用 IStream 接口来替代：

```

void UseRawMemoryToPrintString(void) {
    void *pv = 0;
    // 分配内存
    pv = malloc(13);
    if (pv != 0) {
        // 向底层内存写一个字符串
        memcpy(pv, "Hello, World", 13);
        printf((const char*)pv);
    }
    // 释放所有资源
    free (pv);
}
}

```

等价的代码片断如下：

```

void UseStreamToPrintString(void) {
    IStream *pStm = 0
    // 分配内存并封装 IStream 接口
    HRESULT hr = CreateStreamOnHGlobal(0, TRUE, &pStm);
    if (SUCCEEDED(hr)) {
        // 向底层内存写一个字符串
        hr = pStm->Write("Hello, World", 13, 0);
        assert(SUCCEEDED(hr));
    }
    // 吸出内存
    HGLOBAL hglobal = 0;
    hr = GetHGlobalFromStream(pStm, &hglobal);
    assert(SUCCEEDED(hr));
    printf((const char*)GlobalLock(hglobal));
    // 释放所有资源
    GlobalUnlock(hglobal);
    pStm->Release ();
}
}

```

API 函数 GetHGlobalFromStream 允许调用者提取出一个内存句柄，它指向 Create StreamOnHGlobal 中分配的内存。HGLOBAL 的使用完全是由于历史的原因，并不是由于共享内存的原因。

理解了每个参数的类型之后，CoMarshalInterface API 函数就非常简单了：

```

HRESULT CoMarshal Interface(
    [in] IStream *pStm,           // 写列集状态的位置
    [in] REFIID riid,            // 列集指针类型

```

```

[in, iid_is(riid)] IUnknown *pItf, // 列集指针
    [in] DWORD dwDestCtx,           // 目标的 MSHCTX
    [in] void *pvDestCtx,          // 保留，必须为 0
    [in] DWORD dwMshlFlags        // 标志常规还是表格列集
);

```

下面的代码片断把一个接口指针列集到一块内存中，这块内存可被传输到网络上任何一个套间中。代码如下：

```

HRESULT WritePtr(IRacer *pRacer, HGLOBAL& rhglobal) {
    IStream *pStm = 0; rhglobal = 0;
    // 分配并封装内存块
    HRESULT hr = CreateStreamOnHGlobal(0, FALSE, &pStm);
    if (SUCCEEDED(hr)) {
        // 将列集对象写入内存
        hr = CoMarshalInterface(pStm, IID_IRacer, pRacer,
                               MSHCTX_DIFFERENTMACHINE, 0,
                               MSHLFLAGS_NORMAL);

        // 抽出底层内存的句柄
        if (SUCCEEDED(hr))
            hr = GetHGlobalFromStream(pStm, &rhglobal);
        pStm->Release();
    }
    return hr;
}

```

图 5.1 显示了接口指针和底层内存之间的关系，这里的底层内存包含了列集之后的对象引用。在调用了 `CoMarshalInterface` 之后，对象的套间现在就可以接收来自外部套间的连接请求了。因为用到了 `MSHCTX_DIFFERENTMACHINE` 标志，所以引入套间有可能位于另外一台机器上。

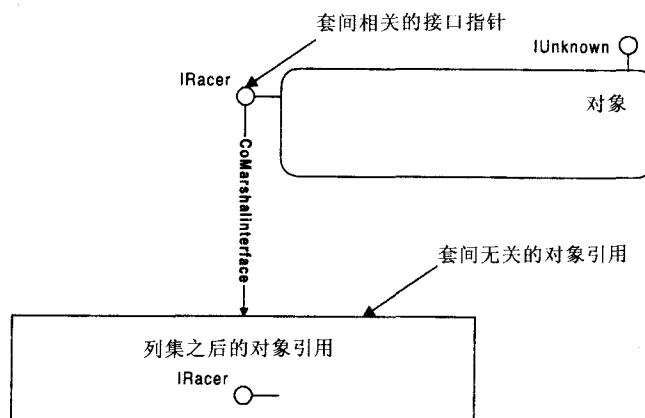


图 5.1 列集之后的对象引用

为了对前面代码片断中创建的“列集之后的对象引用”重新解码，使它成为一个有效的接口指针，引入套间需要调用 CoUnmarshalInterface API 函数，原型如下：

```
HRESULT CoUnmarshalInterface(
    [in] IStream *pStm,           // 读取列集状态
    [in] REFIID riid,            // 散集指针类型
    [out,iid_is(riid)] void **ppv// 存放散集指针的位置
);
```

CoUnmarshalInterface 简单地读入一个经过序列化 (Serialize) 的对象引用，然后返回一个指向原来对象的指针，通过这个指针，调用线程的套间可以合法地访问这个对象。如果引入套间不同于原来引出接口的套间，那么 CoUnmarshalInterface 传回的结果指针将是一个指向代理的指针。如果因为某些原因，CoUnmarshalInterface 调用是由对象所驻留的套间发出的，那么返回的指针将指向实际的对象，这种情况下不需要创建代理。下面的代码把列集之后的对象引用翻译为一个有效的接口指针。

```
HRESULT ReadPtr(HGLOBAL hglobal, IRacer * &rpRacer) {
    IStream *pStm = 0; rpRacer = 0;
    // 现有内存封装块从输入传入
    HRESULT hr = CreateStreamOnHGlobal(hglobal, FALSE, &pStm);
    if (SUCCEEDED(hr)) {
        // 获得对象的有效指针
        hr = CoUnmarshalInterface(pStm,
                                   IID_IRacer, (void**)&rpRacer);
        pStm->Release ();
    }
    return hr;
}
```

结果代理也将实现对象引出的每个接口，但是它只是把方法调用请求转发到对象所在的套间中。

在 Windows NT 4.0 之前的 COM 版本中，被列集之后的对象引用的格式并没有公开的文档介绍。为了允许第三方厂商能够创建 COM 可感知的网络产品，其格式于 1996 年被公开，并且被提交成为 Internet 草案标准。图 5.2 显示了被列集之后的对象引用的格式。被列集之后的对象引用的头部以一个可被识别的签名符号（“MEOW”）<sup>6</sup> 作为开始，紧接着是一个标志字段，它表明所使用的列集技术（比如标准列集方式或者自定义列集方式），然后是引用内部包含的接口 IID。假定采用了标准列集方式，对象引用的子首部 (subheader) 指明了这个被列集之后的引用代表了多少个外部引用。此外部引用计数器是 COM 分布式垃圾收集协议的一部分，它并不直接对应于对象所实现的 AddRef/Release 引用计数器。对象引用中一个很有意思的元素是 OXID/OID/IPID 组，它

<sup>6</sup> Microsoft 的一位不愿透露姓名的程序管理员声称，MEOW 代表 Microsoft Extended Object Wire。虽然我很容易上当，但是我对此仍然很是怀疑。

们唯一标识了一个接口指针。网络上的每个套间在创建的时候，就被分配了一个唯一的 OXID（Object Exporter Identifier，对象引出标识符）。当一个代理第一次连接到对象上的时候，这个 OXID 可用来找到网络/IPC 地址信息。OID（Object Identifier，对象标识符）唯一标识了网络上的一个 COM 实体，CoUnmarshalInterface 利用此 OID，来维护代理的 COM 实体身份法则。IPID（Interface Pointer Identifier，接口指针标识符）唯一标识了套间中的一个接口指针，它被放在每个后续方法调用请求的头部。IPID 可用来有效地把 ORPC 请求分发到对象的套间中正确的接口指针上。

MEOW	签名符号
FLAGS	标准/自定义/控制器标志
IID	被列集的接口 IID
STD FLAGS	与标准列集有关的档志 (no-ping)
CpublicRefs	OBJREF 代表的引用计数
OXID	引出套间的套间标识符
OID	对象实体的对象标识符
IPID	与进程相关的接口指针标识符
cch	主机的字符数/安全信息的偏移
secOffset	
主机地址	DXID 解析器的主机地址
安全包信息	对象引出器使用的安全包

图 5.2 标准列集对象引用

有意思的是，OXID 作为逻辑标识符，其本身并没有用处，但是代理却需要使用某种 IPC 机制或者网络协议，以便与对象的套间进行联系。为了把 OXID 翻译为完全有效的网络地址或者 IPC 地址，每台支持 COM 的机器必须提供一个 OXID 解析器(OR, OXID Resolver) 服务。在 Windows NT 4.0 下面，OR 是 RPCSS 服务的一部分。当一个套间被初始化的时候，COM 分配一个 OXID，然后在本地的 OR 中注册这个 OXID。这意味着每个 OR 都知道本地系统上所有正在运行的套间。OR 也跟踪记录了每个套间的本地 IPC 端口。当 CoUnmarshalInterface 需要连接一个新的代理（指向对象的套间）的时候，它询问本地 OR，以便把 OXID 解析为一个有效的 IPC 或者网络地址。如果散集过程发生在与对象的套间相同的机器上，那么 OR 只是简单地在它本地的 OXID 表中查找 OXID，然后返回本地 IPC 地址。如果散集过程发生在与对象套间不同的机器上，那么本地 OR 首先检查本地的“最近解析过的远程 OXID”缓冲区，以确定是否它在最近已经碰到过

这个 OXID。如果最近还没有碰到过这个 OXID，于是它使用 RPC，把请求转发给对象所在的主机上的 OR。注意，被列集后的对象引用包含有对象的主机地址，其格式适用于各种网络协议，所以散集方的 OR 知道该把请求转发到哪里。

为了实现分布式 OXID 解析过程，每台主机上的 OR 服务针对它所支持的每个网络协议，在一个为大家所熟知的端点上（对 TCP 和 UDP 来说是 135 端口）监听远程的 OXID 解析请求。当主机接收到来自网络的 OXID 解析请求时，便去查询本地 OXID 表。这个解析请求指示了客户机支持哪些网络协议。如果被请求的套间所在的进程还没有开始使用这些协议，那么 OR 将使用 IPC，与对象套间中的 COM 库联系，使对象的进程启动客户指定的协议之一。一旦完成这一步之后，OR 就把套间的新网络地址填入本地 OXID 表中。在新网络地址被记录下来之后，这个地址被返回给散集方的 OR，在那里它被缓存起来，以便对于常用的套间避免多次网络请求。虽然“不把对象的完整网络地址直接记录到被列集后的对象引用中”这种做法看起来非常怪异，但是由于“套间 ID（OXID）与协议独立”而带来的间接性，使得基于 COM 的进程直到真正必要的时候才使用网络协议。从本质上讲，这是非常重要的，因为 COM 可以在各种不同的协议（不仅仅是 TCP）上面运行，所以要求每个进程对所有的协议都监听请求显然是非常低效的。实际上，如果基于 COM 的进程从来也没有把接口指针引出到远程主机上，那么它永远也不会消耗任何网络资源。

## 5.4 进程内列集辅助函数

尽管上一节的 WritePtr 和 ReadPtr 代码片断可以非常直接地编写出来，但是 COM 认识到，大多数显式的 CoMarshalInterface 调用都是为了将接口指针从一个线程传递到同一进程内的另一个线程中。为了简化这项任务，COM 提供了另外两个封装函数，它们分别实现了围绕 CoMarshalInterface 和 CoUnmarshalInterface 的一系列动作。COM API 函数 CoMarshalInterThreadInterfaceInStream 的原型如下：

```
HRESULT CoMarshalInterThreadInterfaceInStream(
    [in] REFIID riid,
    [in, iid_is(riid)] IUnknown *pItf,
    [out] IStream **ppStm
);
```

它简单地封装了 CreateStreamOnHGlobal 和 CoMarshalInterface 调用，如下所示：

```
// 摘自 OLE32.DLL (近似)
HRESULT CoMarshalInterThreadInterfaceInStream(
    REFIID riid, IUnknown *pItf, IStream **ppStm) {
```

```

HRESULT hr = CreateStreamOnHGlobal(0, TRUE, ppStm);
if (SUCCEEDED(hr))
    hr = CoMarshallInterface(*ppStm, riid, pItf,
                           MSHCTX_INPROC, 0, MSHLFLAGS_NORMAL);
return hr;
}

```

COM 也提供了围绕 CoUnmarshalInterface 函数的封装函数：

```

HRESULT CoGetInterfaceAndReleaseStream(
    [in] IStream *pStm,
    [in] REFIID riid,
    [out, iid_is(riid)] void **ppv
);

```

它只是在 CoUnmarshalInterface 基础上极为简单的封装：

```

// 摘自 OLE32.DLL (近似)
HRESULT CoGetInterfaceAndReleaseStream(
    IStream *pStm, REFIID riid, void **ppv) {
HRESULT hr = CoUnmarshalInterface(pStm, riid, ppv);
pStm->Release();
return hr;
}

```

这两个函数并没有增加特殊的功能，但是它们比相应的低层函数用起来更为方便。

下面的代码片断可用于把一个接口指针传递到同一进程内另一个套间中，并且利用全局变量来存放中间结果，即被列集之后的对象引用，代码如下：

```

HRESULT WritePtrToGlobalVariable(IRacer *pRacer) {
// 存放列集指针
extern IStream *g_pStmPtr;
// 读写线程同步
extern HANDLE g_hEventWritten;
// 将列集对象引用写入全局变量
HRESULT hr = CoMarshalInterfaceInStream(
    IID_IRacer, pRacer, &g_pStmPtr);
// 通知其他线程，指针可用了
SetEvent(g_hEventWritten);
return hr;
}

```

下面对应的代码可以把对象引用正确地散集到位于同一进程中的调用者套间中：

```

HRESULT ReadPtrFromGlobalVariable(IRacer * &pRacer) {
// 存放列集指针
extern IStream *g_pStmPtr;
// 读写线程同步
extern HANDLE g_hEventWritten;

```

```

// 等待其他线程通知指针可用
WaitForSingleObject(g_hEventWritten, INFINITE);
// 从全局变量读取列集对象引用
HRESULT hr = CoGetInterfaceAndReleaseStream(
    g_pStmPtr, IID_IRacer, (void**)&rpRacer);
// MSHLFLAGS_NORMAL 意味着没有其他有效散集
g_pStmPtr = 0;
return hr;
}

```

当我们把指针从一个套间传递到另一个套间中时，就需要用到上面的代码。<sup>7</sup> 注意到，把一个指针从正在 MTA 或者 RTA 中执行的线程，传递到位于同一套间的另一个执行线程中，这种情况下并不需要列集调用。然而，通常的做法是，接口指针的写操作一方（writer）在传递给读线程（reader）之前，先要调用 AddRef。读线程在用完了指针之后，当然要调用 Release。

注意，前面的“读指针”代码在散集之后，把全局变量 g\_pStmPtr 设置为空（null）。这是因为对象引用是用 MSHLFLAGS\_NORMAL 标志列集得到的，它只能被散集一次。在许多情况下，这样做不是个问题。然而，在有些情况下，我们希望在一个线程中列集了一个指针之后，将来可能有多个辅助线程在必要的时候会散集该接口指针。如果所有的辅助线程都运行在 MTA 中，这也不是问题，因为只需一个线程执行散集操作，它可以代表 MTA 中所有的线程。然而，如果辅助线程运行在任意的套间中，那么这种方法就不能正常工作，因为每个辅助线程将需要独立地散集对象引用。这时候大多数开发人员会转向 MSHLFLAGS\_TABLESTRONG 标志，希望只要列集一次，以后就可以根据需要散集多次（每个套间一次）。不幸的是，如果原来的指针是个代理的话，那么它就不支持表格列集方式（不同于常规列集），而这种情形（指代理接口指针）又是非常常见的，特别是在分布式应用中。为了解决这种需求，Windows NT 4.0 的 Service Pack 3 发放的 COM 引入了全局接口表（GIT，Global Interface Table）。

GIT 是 CoMarshalInterface/CoUnmarshalInterface 的一个优化，它允许接口指针可被进程中所有的套间访问。COM 内部为每个进程实现了一个 GIT。GIT 包含有经过列集的接口指针，在同一个进程中这些接口指针可被有效地散集多次。这在语义上等价于使用表格列集，但是 GIT 既可以被用于对象，也可以被用于代理。GIT 暴露了 IGlobalInterfaceTable 接口：

```

[ uuid(00000146-0000-0000-C000-00000000046), object, local ]
interface IGlobalInterfaceTable : IUnknown {
    // 列集一个接口进 GIT
    HRESULT RegisterInterfaceInGlobal (

```

<sup>7</sup> 看起来奇怪的是，全局变量是一个接口指针，它在写套间中被初始化，而在读套间中被使用。CoMarshalInterfaceInStream 的说明文档指出了这种不一致性，它声明：结果得到的 IStream 接口指针可被进程中任何一个套间访问。

```

        [in, iid_is(riid)] IUnknown *pltf,
        [in] REFIID riid,
        [out] DWORD *pdwCookie);

// 销毁列集对象引用
HRESULT RevokeInterfaceFromGlobal (
    [in] DWORD dwCookie);

// 从 GIT 散集一个接口
HRESULT GetInterfaceFromGlobal (
    [in] DWORD dwCookie
    [in] REFIID riid,
    [out, iid_is(riid)] void **ppv);
}

```

客户只需使用类 `CLSID_StdGlobalInterfaceTable` 调用 `CoCreateInstance`，就可以访问到其进程的 GIT。每次用这个 `CLSID` 调用 `CoCreateInstance` 都会返回一个指针，但它们指向该进程唯一的 GIT。与 `CoMarshalInterThreadInterfaceInStream` 返回的 `IStream` 接口一样，指向 GIT 的接口指针可被任何一个套间访问，而无需列集。

为了使一个接口指针可被进程中所有的套间使用，拥有接口指针的套间必须把它注册到 GIT 中，这可以通过调用 `RegisterInterfaceInGlobal` 方法来完成。GIT 将给调用者返回一个 `DWORD`，该进程中所有的套间都可以使用这个 `DWORD` 值。进程中任何一个套间通过调用 `GetInterfaceFromGlobal` 方法，用这个 `DWORD` 就可以散集出一个新的代理。这个 `DWORD` 可被多次重复地用来散集代理对象，一直到调用 `RevokeInterfaceFromGlobal` 把全局接口指针注销为止。使用全局接口表的应用通常在启动的时候，绑定一个全进程范围内有效的接口指针，如下：

```

IGlobalInterfaceTable *g_pGIT = 0;
HRESULT InitOnce(void) {
    assert(g_pGIT == 0);
    return CoCreateInstance(CLSID_StdGlobalInterfaceTable, 0,
                           CLSCTX_INPROC_SERVER,
                           IID_IGlobalInterfaceTable,
                           (void**)&g_pGIT);
}

```

一旦获得了全局接口表以后，“把一个接口指针传递到另一个套间中”这项工作仅仅意味着把指针注册到全局接口表中：

```

HRESULT WritePtrToGlobalVariable(IRacer *pRacer) {
    // 存放列集指针
    extern DWORD g_dwCookie;
    // 线程同步
    extern HANDLE g_hEventWritten;
    // 将列集对象引用写入全局变量
    HRESULT hr = g_pGIT->RegisterInterfaceInGlobal(
        pRacer, IID_IRacer, &g_dwCookie);
}

```

```
// 通知其他线程指针可用了
SetEvent (g_hEventWritten);
return hr;
}
```

下面的代码可以正确地散集出对象引用，进程中任何一个套间都可以调用这段代码。代码如下：

```
HRESULT ReadPtrFromGlobalVariable(IRacer * &rpRacer,
                                  bool bLastUnmarshal) {
    // 存放列集指针
    extern DWORD g_dwCookie;
    // 线程同步
    extern HANDLE g_hEventWritten;
    // 等待其他线程通知指针可用
    WaitForSingleObject(g_hEventWritten, INFINITE);
    // 全局变量读取列集对象引用
    HRESULT hr = g_pGIT->GetInterfaceFromGlobal(
        g_dwCookie, IID_IRacer, (void**)&rpRacer);
    // 如果散集到最后，销毁指针
    if (bLastUnmarshal)
        g_pGIT->RevokeInterfaceFromGlobal(g_dwCookie);
    return hr;
}
```

注意，这些代码片断与前面的“使用 CoMarshalInterThreadInterfaceInStream”的例子之间的关键区别在于：基于 GIT 的代码支持散集多个代理。

## 5.5 标准列集结构

正如本章前面所提到的，COM 使用 ORPC 协议来解决跨套间访问的问题。从结构上来看，这个事实很让人感兴趣，但是很少有开发人员会去编写底层的通信代码。为了使用 ORPC 通信，COM 对象并不需要作特殊的工作，只要实现 IUnknown 就可以提供基于 ORPC 的跨套间访问功能。缺省情况下，当我们第一次针对一个对象调用 CoMarshalInterface 时，CoMarshalInterface 会询问这个对象，它是否希望使用自己的跨套间通信机制。这个问题是以“向对象调用 QueryInterface 查询 IMarshal 接口”的形式进行的。大多数对象并没有实现 IMarshal 接口，所以这个 QueryInterface 调用会失败，这就表明它们很愿意让 COM 通过 ORPC 来处理所有通信过程。如果一个对象实现了 IMarshal 接口，那么它表示 ORPC 并不合适，对象实现者更愿意用一个自定义的代理来处理所有的跨套间通信过程。如果一个对象实现了 IMarshal 接口，那么指向这个对象的

所有引用都将是自定义列集方式的。自定义列集过程将在本章后面讨论。如果一个对象没有实现 IMarshal 接口，那么指向这个对象的所有引用都将是标准列集方式的。大多数对象选择使用标准列集方式，这也是本节要讨论的焦点。

当 CoMarshalInterface 第一次确定了对象希望使用标准列集方式时，它创建一个特殊的 COM 对象，被称为存根管理器（stub manager）。存根管理器作为该对象在网络范围内的实体，并且被一个对象标识符（OID）唯一标识，这个 OID 跨越所有的套间代表了这个对象实体。在存根管理器和 COM 对象实体之间有一个一一对应关系。每个存根管理器恰好代表一个 COM 对象。每个使用标准列集方式的 COM 对象也正好只有一个存根管理器。存根管理器至少拥有一个指向对象的未完结引用，这使得对象的资源被保留在内存中。从这个意义上讲，存根管理器也是对象的另一个进程内客户。存根管理器跟踪记录了未完结外部引用的数目，只要网络上至少有一个未完结的引用，它就仍然存活在内存中。大多数外部引用都只是简单的代理，当第一个代理被创建的时候，被列集之后的中间对象引用使存根保持运行状态，以确保对象仍然处于存活状态。当未完结的代理/引用被销毁的时候，存根管理器就会接到通知，它减小它的外部引用计数器。当存根管理器中最后一个外部引用被销毁的时候，存根管理器就把自己也销毁掉，并且把它所拥有的指向实际对象的未完结引用释放掉。这种做法模拟了“客户端引用使对象保持活动状态”的效果。显式地控制存根生命周期的技术将在本章后文讨论。

存根管理器只是简单地扮演了对象的“网络实体身份”的角色，它并不理解如何处理“目标指向某个对象”的入 ORPC 请求。<sup>8</sup>为了把入 ORPC 请求转译为对象上实际的方法调用，存根管理器需要一个辅助对象，这个辅助对象知道关于接口方法特征的所有细节。这个辅助对象被称为接口存根（interface stub），它必须能够正常地把 ORPC 请求消息中出现的[in]参数都散集出来，并调用实际对象上的方法，然后把 HRESULT 和所有的[out]参数列集到 ORPC 响应消息中。接口存根在内部是用接口指针标识符（IPID）来标识的，并且 IPID 在一个套间内部是唯一的。与存根管理器一样，每个接口存根都拥有一个指向对象的引用；然而，被拥有的接口引用都是有类型的，不是简单的 IUnknown。图 5.3 显示了存根管理器、接口存根和对象三者之间的关系。注意，有的接口存根知道如何解释多个接口类型，而其他的接口存根只能解释一个接口类型。

当 CoUnmarshalInterface 把一个标准列集得到的对象引用散集出来的时候，它会返回一个指向代理管理器（proxy manager）的指针。代理管理器扮演了对象在客户端的实体身份的角色，与存根管理器一样，它并不理解任何有关 COM 接口的知识。然而，代理管理器却知道如何实现 IUnknown 的三个方法。任何多余的 AddRef 或者 Release 调用，都只是简单地增加或者减小代理管理器内部的一个引用计数，这些调用永远也不会通过 ORPC 被传递出去。对于代理管理器的最后一个 Release 会真正销毁代理，给对象的套间发送一个断开连接的请求。对代理管理器的 QueryInterface 请求的处理方式有些不同。

<sup>8</sup> 从逻辑上讲，存根管理器也会处理对 IUnknown 方法的远程调用；然而，这项功能其实是由套间对象（它暴露了 IRemUnknown 接口）实现的。

与存根管理器一样，代理管理器并没有任何关于 COM 接口的先验知识。相反，它必须要装载接口代理（interface proxy），接口代理暴露了被远程请求的实际接口。接口代理把方法调用翻译成为 ORPC 调用。与存根管理器不同的是，代理管理器对于程序员来说是直接可见的，它可以维护正确的实体身份关系，接口代理被聚合到代理管理器的实体中。这让客户感觉到所有的接口都是从同一个 COM 对象上暴露出来的。图 5.4 显示了代理管理器、接口代理和存根三者之间的关系。

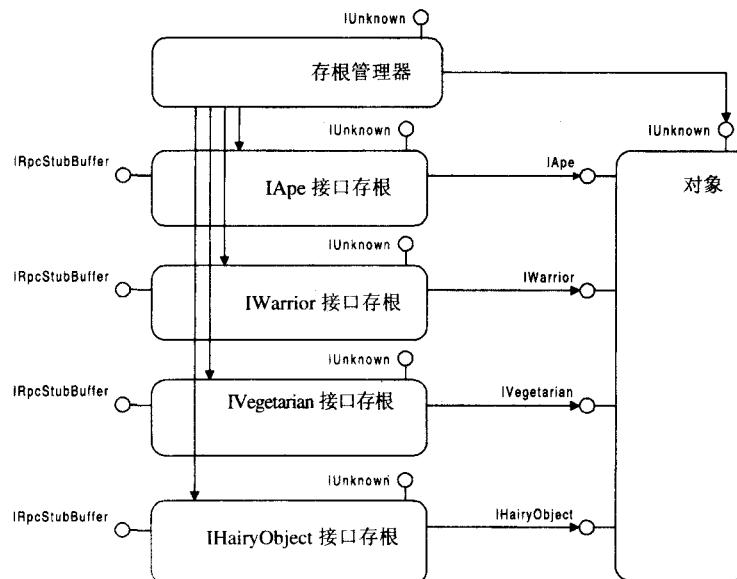


图 5.3 存根结构

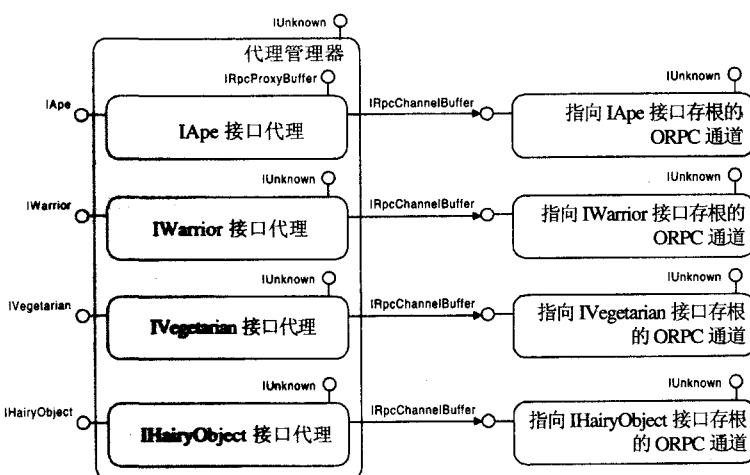


图 5.4 代理结构

如图 5.4 所示，代理与存根之间通过第三个被称为通道（channel）的对象进行通信。通道是由 COM 提供的对象，它封装了 RPC 运行时层。通道暴露了 IRpcChannelBuffer 接口：

```
[uuid(D5F56B60-593B-101A-B569-08002B2DBF7A),local,object]
interface IRpcChannelBuffer : "IUnknown" {
    // ORPC 消息的程序表示
    typedef struct tagRPCOLEMESSAGE {
        void             *reserved1;
        unsigned long    dataRepresentation; // endian/ebcdic 编码方式
        void             *Buffer;           // 载荷
        ULONG            cbBuffer;         // 载荷长度
        ULONG            iMethod;          // 方法
        void             *reserved2[5];
        ULONG            rpcFlags;
    } RPCOLEMESSAGE;

    // 分配传输缓冲
    HRESULT GetBuffer([in] RPCOLEMESSAGE *pMessage,
                      [in] REFIID riid);

    // 发送 ORPC 请求并接收 ORPC 响应
    HRESULT SendReceive([in,out] RPCOLEMESSAGE *pMessage,
                        [out] ULONG *pStatus);

    // 释放传输缓冲区
    HRESULT FreeBuffer([in] RPCOLEMESSAGE *pMessage);

    // 获取 CoMarshalInterface 到目标的距离
    HRESULT GetDestCtx([out] DWORD *pdwDestCtx,
                       [out] void **ppvDestCtx);

    // 检测是否显式断开连接
    HRESULT IsConnected(void);
}
```

接口代理使用上述接口中的 SendReceive 方法，让通道发送 ORPC 请求消息，以及接收 ORPC 响应消息。

接口代理和存根都是简单的进程内 COM 对象，它们分别由代理和存根管理器使用普通的 COM 激活技术创建得到。接口存根必须暴露 IRpcStubBuffer 接口：

```
[ uuid(D5F56AFC-593B-101A-B569-08002B2DBF7A),local,object ]
interface IRpcStubBuffer : IUnknown {
    // 调用以将存根与对象连接
    HRESULT Connect([in] IUnknown *pUnkServer);

    // 调用以通知存根释放对象
    void Disconnect(void);

    // 请求到来时调用
    HRESULT Invoke([in] RPCOLEMESSAGE *pmsg,
                  [in] IRpcChannelBuffer *pChannel);

    // 用于支持每个存根多个 itf 类型
}
```

```

IRpcStubBuffer *IsIIDSupported([in] REFIID riid);
// 用于支持每个存根多个 itf 类型
ULONG CountRefs(void);
// 用于 ORPC debugger 寻找对象指针
HRESULT DebugServerQueryInterface(void **ppv);
// 用于 ORPC debugger 释放对象指针
void DebugServerRelease(void *pv);
}

```

当 ORPC 请求到达对象一方时，COM 库将会调用 `Invoke` 方法。在输入时，`RPCOLEMESSAGE` 包含所有经过列集的 [in] 参数；在输出时，存根必须列集方法的 `HRESULT` 和所有的 [out] 参数，它们将在 ORPC 响应消息中被返回。

接口代理必须暴露所有“归它负责远程调用”的接口，以及 `IRpcProxyBuffer` 接口：

```

[ uuid(D5F56A34-593B-101A-B569-08002B2DBF7A), local, object ]
interface IRpcProxyBuffer : IUnknown {
    HRESULT Connect([in] IRpcChannelBuffer *pChannelBuffer);
    void Disconnect(void);
}

```

`IRpcProxyBuffer` 接口必须是接口代理的“非委托 `IUnknown` 接口”。接口代理暴露的所有其他接口必须把它们的 `IUnknown` 方法委托给代理管理器。在这些接口的方法实现内部，接口代理必须利用通道对象把 ORPC 请求发送给接口存根的 `Invoke` 方法，然后由 `Invoke` 方法执行对象套间中的实际方法。

接口代理和接口存根都是被动态绑定到代理管理器和存根管理器上的，两者共享同一个 `CLSID`。这个包含两个分叉实现的实体通常被称为接口列集器（interface marshall））。接口列集器的类对象暴露了 `IPSFFactoryBuffer` 接口：

```

[ uuid(D5F569D0-593B-101A-B569-08002B2DBF7A), local, object ]
interface IPSFactoryBuffer : IUnknown {
    HRESULT CreateProxy(
        [in] IUnknown *pUnkOuter,           // 代理服务器指针
        [in] REFIID riid,                 // 远程请求 itf
        [out] IRpcProxyBuffer **ppProxy,   // 代理 itf 指针
        [out] void **ppv                  // 远程接口指针
    );
    HRESULT CreateStub(
        [in] REFIID riid,                 // 远程请求 itf
        [in] IUnknown *pUnkServer,         // 实际对象指针
        [out] IRpcStubBuffer **ppStub     // 输出存根指针
    );
}

```

代理管理器调用 `CreateProxy` 方法来聚合一个新的接口代理。存根管理器调用 `CreateStub` 方法来创建一个新的接口存根。

当客户请求对象上一个新的接口时，代理和存根管理器必须把被请求的 IID 解析为接口列集器的 CLSID。在 Windows NT 5.0 下面，类存储（class store）把这些映射关系保存在 NT 目录中，在每台机器上，这些映射关系被缓存在本地注册表中。全机器范围内的“IID-CLSID”映射关系被缓存在下面的键下：

```
HKEY_CLASSES_ROOT\Interface
```

而针对每个用户的映射关系被缓存在下面的键下：

```
HKEY_CURRENT_USER\Software\Classes\Interface
```

这些键中，每个已知的接口都有一个子键。Windows NT 4.0 或者更早的版本没有类存储，它们只使用本地注册表的 HKEY\_CLASSES\_ROOT\Interface 区域。

如果一个接口的接口列集器已经被安装了，那么就会有一个附加的子键（ProxyStubClid32），用来指示接口列集器的 CLSID。下面显示了可列集的接口所需要的注册表键：

```
[HKCR\Interface\{1A3A29F0-D87E-11d0-8C4F-0080C73925BA}]
@="IRacer"
[HKCR\Interface\{1A3A29F0-D87E-11d0-8C4F-0080C73925BA}\ProxyStubClid32]
@="{1A3A29F3-D87E-11d0-8C4F-0080C73925BA}"
```

这些注册表项描述了一个进程内服务器，其 CLSID 为 {1A3A29F3-D87E-11d0-8C4F-0080C73925BA}，它实现了接口 IRacer {1A3A29F0-D87E-11d0-8C4F-0080C73925BA} 的接口代理和存根。这暗示着 HKCR\CLSID 将包含一个针对这个接口列集器的子键，可以把 CLSID 映射到对应的 DLL 文件名。而且，在 Windows NT 5.0 下面，这个映射关系可能位于类存储中，类存储会动态地生成本地注册表信息。因为接口列集器必须运行在与代理管理器或者存根管理器相同的套间中，所以它们必须使用 ThreadingModel=“Both”，以确保它们总是能够被装入到正确的套间中。

## 5.6 实现接口列集器

上一节说明了标准列集结构所用到的四个接口。虽然我们有可能使用手工编写 C++ 代码的方式来实现接口列集器，但是实践中很少有这样做的。这是因为 IDL 编译器可以在接口的 IDL 定义的基础上，自动产生接口列集器的所有 C 代码。MIDL 产生的接口列集器使用网络数据表示（NDR，Network Data Representation）协议来序列化方法的参数，这使得这些参数可被散集到各种主机结构上。NDR 考虑到了不同字节顺序之间的差异、不同浮点数的格式、不同的字符集以及边界对齐的因素。NDR 支持几乎所有与 C 语言

兼容的数据类型。为了支持把接口指针当作参数来传递， MIDL 产生相应的 CoMarshalInterface/CoUnmarshalInterface 调用代码，完成任何一个接口指针的列集过程。如果参数是一个类型固定的接口指针，比如：

```
HRESULT Method([out] IRacer **ppRacer);
```

那么结果产生的列集代码在列集 ppRacer 参数时，会把 IRacer 的 IID (IID\_IRacer) 传递给 CoMarshalInterface/CoUnmarshalInterface 调用。相反，如果接口指针是动态类型的，比如：

```
HRESULT Method([in] REFIID riid,
               [out, iid_is(riid)] void **ppv);
```

那么结果产生的列集代码将会使用在运行时第一个方法参数中传递的 IID 来列集接口。

针对定义在库 (library) 语句范围之外的每一个非本地接口， MIDL 也产生接口列集器源代码。在下面的 IDL 伪代码中：

```
// sports.idl
[local, object] interface IBoxer : IUnknown { ... }
[object] interface IRacer : IUnknown { ... }
[object] interface ISwimmer : IUnknown { ... }
[helpstring("Sports Lib")]
library SportsLibrary {
    interface IRacer; // 包含 TLB 中 IRacer 的定义
    [object] interface IWrestler : IUnknown { ... }
}
```

只有 IRacer 和 ISwimmer 接口将会有接口列集器源代码。但是 MIDL 不会产生 IBoxer 的列集代码，因为 [local] 属性抑制了列集过程。 MIDL 也不会为接口 IWrestler 产生列集器，因为它被定义在库语句的内部。

对于上述 IDL 来说， MIDL 编译器将会产生五个文件。文件 sports.h 将包含接口的 C/C++ 定义， sports\_i.c 将包含 IID 和 LIBID 的定义， sports.tlb 将包含接口 IRacer 和 IWrestler 的符号化的 IDL (tokenized IDL)，适合用于 COM 可感知的开发环境。文件 sports\_p.c 将包含实际的代理/存根实现代码，它们会完成从方法调用到 NDR 的转换过程。这个文件也包含了针对每个接口和存根的、基于 C 语言的 vtbase 定义，以及其他与 MIDL 相关的管理代码。因为接口列集器是 COM 进程内服务器，所以 MIDL 还必须定义四个标准的入口函数 (DllGetClassObject 等)。这四个方法被定义在第 5 个文件 dlldata.c 中。

有了这些文件之后，建立接口列集器所需要做的工作就是写一个 makefile，由它来编译三个 C 源文件 (sports\_i.c、sports\_p.c、dlldata.c)，然后把它们链接起来得到 DLL。四个标准 COM 入口点函数必须被显式引出，或者通过模块定义文件，或者使用链接器开关。注意，在缺省情况下， dlldata.c 只包含 DllGetClassObject 和 DllCanUnloadNow 的

定义。这是因为 Windows NT 3.50 下面的 RPC 运行库只支持这两个函数。如果接口列集器只被用于 Windows NT 3.51 或者以后的版本(或者 Windows 95), 那么在编译 dldata.c 文件的时候, 应该定义 C 预处理器符号 REGISTER\_PROXY\_DLL, 以便同时也编译出标准的自注册入口点函数。一旦接口列集器被建立起来之后, 它应该被安装到本地注册表中, 或者类存储中。

Windows NT 4.0 的 COM 库版本引入了对“解释性列集 (interpretive marshaling) ”的全面支持。根据不同的接口, 使用解释性列集可以降低工作集 (working set) 的大小, 从而极大地提高应用的性能。针对所有 COM 标准接口而预先安装的接口列集器就使用了解释性列集器。MTS 要求接口列集器必须使用解释性列集器。<sup>9</sup> 为了能够使用解释性列集器, 我们只需在运行 MIDL 编译器时使用/Oicf 命令行开关即可, 例如:

```
midl.exe /Oicf sports.idl
```

在我写作这本书的时候, MIDL 编译器并不会覆盖掉原先已有的\_p.c 文件, 所以在改变这个设置时, 这个文件必须被删除掉。因为基于/Oicf 命令行开关生成的接口列集器将无法在 Windows NT 4.0 之前的 COM 版本下正常工作, 而且在编译列集器的源代码的时候, C 预处理器符号\_WIN32\_WINNT 必须要被定义为大于或者等于 0x400 的某个整数。C 编译器在编译时会强制这项要求。

产生接口列集器的第三项技术专门针对有特殊限制的一类接口。如果一个接口只使用 VARIANT<sup>10</sup> 所支持的基本数据类型, 那么它就可以使用通用的列集器。在接口定义中加上[oleautomation]属性就可以使通用列集器对该接口起作用了:

```
[ uuid(F99D19A3-D8BA-11d0-8C4F-0080C73925BA), version(1.0) ]
library SportsLib {
    importlib ("stdole32.tlb");
    [
        uuid (F99D1907-D8BA-11d0-8C4F-0080C73925BA), object,
        oleautomation
    ]
    interface IWrestler : IUnknown {
        import "oaidl.idl";
        HRESULT HalfNelson([in] double nmsec);
    }
}
```

[oleautomation]属性的出现相当于告诉 RegisterTypeLib 函数, 在注册类型库时加入下面附加的注册表项:

```
[HKCR\Interface\{F99D1907-D8BA-11d0-8C4F-0080C73925BA}]
```

<sup>9</sup> MTS 也要求列集器必须使用一个特殊的运行库来创建, 这个运行库可让 MTS 根据它的解释格式, 来发现有关接口的信息。

<sup>10</sup> VARIANT 是脚本环境使用的一种数据类型, 第 2 章曾经讲述过这种类型。

```

@="IWrestler"
[HKCR\Interface\{F99D1907-D8BA-11d0-8C4F-0080C73925BA}\ProxyStubClSID32]
@="{00020424-0000-0000-C000-00000000046}"
[HKCR\Interface\{F99D1907-D8BA-11d0-8C4F-0080C73925BA}\ProxyStubClSID]
@="{00020424-0000-0000-C000-00000000046}"
[HKCR\Interface\{F99D1907-D8BA-11d0-8C4F-0080C73925BA}\TypeLib]
@:{F99D19A3-D8BA-11d0-8C4F-0080C73925BA}"
Version:"1.0"

```

CLSID {00020424-0000-0000-C000-00000000046} 对应于通用列集器，它被预装在所有支持 COM 的平台上，包括 16 位 Windows。

使用通用列集器最主要的好处是，它是 16 位应用和 32 位应用之间唯一同时被支持的标准列集技术。通用列集器也兼容于 MTS。通用列集器的另一个好处是，如果类型库同时被安装在客户和对象所在的主机上，那么就不再需要附加的接口列集器 DLL。使用通用列集器主要的缺点是对于参数数据类型的支持有所限制。这与“动态调用”和“脚本环境”所强加的限制完全一样，但是对于设计底层系统编程接口来说，这是一个很严重的限制。<sup>11</sup> 在 Windows NT 4.0 环境下，调用 CoMarshal Interface/CoUnmarshalInterface 的初始开销会高于通用列集器所需要的开销。然而，一旦接口代理和存根已经被构造出来之后，方法调用的性能基本上等同于基于/Oicf 的列集器。

## 5.7 标准列集、线程和协议

COM 如何把 ORPC 请求映射到线程的实际细节并没有公开的文档，有可能会随着 COM 库版本的发展而有所变化。本节所讲述的信息在我写作本书的时候还是正确的，但是某些特定的细节在以后的 COM 版本中会发生变化。

当进程中第一个套间被初始化的时候，COM 就会使 RPC 运行时层起作用，并且把进程转变为一个 RPC 服务器。如果套间类型是 MTA 或者 RTA，那么就会用到 ncalrpc RPC 协议序列，这是对 Windows NT LPC 端口的一个包装。如果套间类型为 STA，于是就会用到一个私有的协议序列，它建立在 Windows MSG 队列的基础上。当驻留在进程中的对象第一次被远程客户访问的时候，其他的网络协议序列也被注册到进程中。当进程首次使用 Windows MSG 协议之外的其他协议的时候，RPC 线程缓冲区就会被启动起来。这个线程缓冲区启动一个线程，由它监听进入到本机的连接请求、RPC 请求或者其他与协议有关的活动。当任何一个事件发生时，RPC 线程缓冲区就会为这个请求分配一个线程，由它为这个请求提供服务，而自己继续等待其他的活动。为了避免过多的线程创建和销毁负担，这些线程又会回到线程缓冲区，在那里等待其他的工作。如果没有其他的

<sup>11</sup> 有可能将来的 COM 库版本会去掉这项限制。请参考你手头的文档以获取更多的信息。

工作到来的话，那么这些线程在经过了一个预先设定的时间周期之后就会销毁自己。这种做法的直接效果是，根据进程的套间所引出的对象的忙碌程度，RPC 线程缓冲区会相应地增加或者缩减。从程序设计的角度来看，重要的一点在于，RPC 线程缓冲区会根据各种协议（除了 Windows MSG 协议之外）所到达的 ORPC 请求动态地分配线程。本节后面将会进一步讨论 Windows MSG 协议。

当缓冲区为一个入 ORPC 请求分配了一个线程时，该线程从 ORPC 调用的头部提取出 IPID，然后找到对应的存根管理器和接口存根。线程判断对象所在的套间的类型，如果对象位于 MTA 或者 RTA 中，那么线程进入对象的套间，然后在接口存根上调用 IRpcStubBuffer::Invoke 方法。如果套间是 RTA，那么在方法调用期间，后续的线程将被阻塞。如果套间是 MTA，那么后续线程可以并行访问对象。对于进程内部的 RTA/MTA 通信，通道（译注 1）能够绕过 RPC 线程缓冲区，重用客户线程，只是简单地让客户线程临时进入到对象的套间中。如果 MTA 和 RTA 是仅有的两种套间类型的话，那么这就是所有需要考虑的情况了。

把调用分发给 STA 要更加复杂一些，因为其他的线程不可能进入到现有的 STA 套间中。不幸的是，当来自远程主机的 ORPC 请求到达的时候，它们会被分配到 RPC 线程缓冲区中的线程，而根据定义，线程缓冲区中的线程不可能在对象的 STA 内部执行。为了进入到 STA 中，并把调用分发给 STA 的线程，RPC 线程使用 PostMessage API 函数，把一个消息发送到 STA 线程的 MSG 队列中，如图 5.5 所示。这个队列就是窗口系统所使用的 FIFO 队列。这意味着，为了完成分发调用的任务，STA 线程必须通过下列代码的某种变化形式来提供队列服务：

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

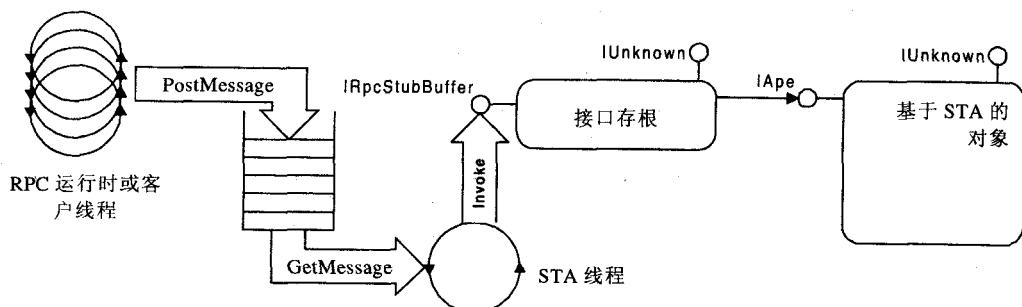


图 5.5 单线程套间的调用分发

译注 1：channel，即客户方代理与对象存根之间的第三个对象。

上述代码隐含了 STA 线程至少有一个窗口可以接收消息。当一个线程调用 CoInitializeEx 进入到一个新的 STA 中时，COM 调用 CreateWindowEx 创建一个不可见的窗口。这个窗口与一个由 COM 注册的窗口类相关联，这个窗口类的 WndProc 检查预定义的窗口消息，并且通过调用接口存根的 IRpcStubBuffer::Invoke 方法来响应对应的 ORPC 请求服务。注意，因为窗口与基于 STA 的对象一样，它们都具有线程相依性，所以 WndProc 将在对象的套间内部执行。为了避免过多的线程切换，Windows 95 发放的 COM 版本引入了“RPC 传输（RPC transport）”机制，它可以绕过 RPC 线程缓冲区，直接在调用者的线程中调用 PostMessage。这种传输机制只有当客户与对象位于同一台机器上时才有效，因为 PostMessage API 不能跨网络工作。

为了避免死锁，所有的 COM 套间类型都支持重入。<sup>12</sup> 当套间中的线程通过代理调用到其他套间中的对象的时候，调用者线程在等待此调用的 ORPC 应答的过程中，它可以继续处理其他的入方法请求。如果没有这样的支持，那么我们就不可能在相互协作的对象基础上建立起系统。在下面的代码中，假定 CLSID\_Callback 是一个进程内服务器，它支持调用线程的线程模型，而 CLSID\_Object 是一个被配置成在远程机器上激活的类：

```
ICallback *pcb = 0;
HRESULT hr = CoCreateInstance(CLSID_Callback, 0, CLSCTX_ALL,
                               IID_ICallback, (void**)&pcb);
assert(SUCCEEDED(hr)); // 回调对象存在此套间
IObject *po = 0;
hr = CoCreateInstance(CLSID_Object, 0, CLSCTX_REMOTE_SERVER,
                      IID_IObject, (void**)&po);
assert(SUCCEEDED(hr)); // 对象在不同套间
// 调用远程对象，列集一个
// 回调对象的引用为[in]参数
hr = po->UseCallback(pcb);
// 清除资源
pcb->Release();
pco->Release();
```

如图 5.6 所示，如果调用者的套间不支持重入，那么下面的 UseCallback 方法将会引起死锁：

```
STDMETHODIMP Object::UseCallback(ICallback *pcb) {
    HRESULT hr = pcb->GetBackToCallersApartment0
assert (SUCCEEDED (hr));
return S_OK;
}
```

<sup>12</sup> 在本书写作的时候，COM 没有提供非重入的套间类型。有可能将来的 COM 版本会提供不支持重入的新套间类型。

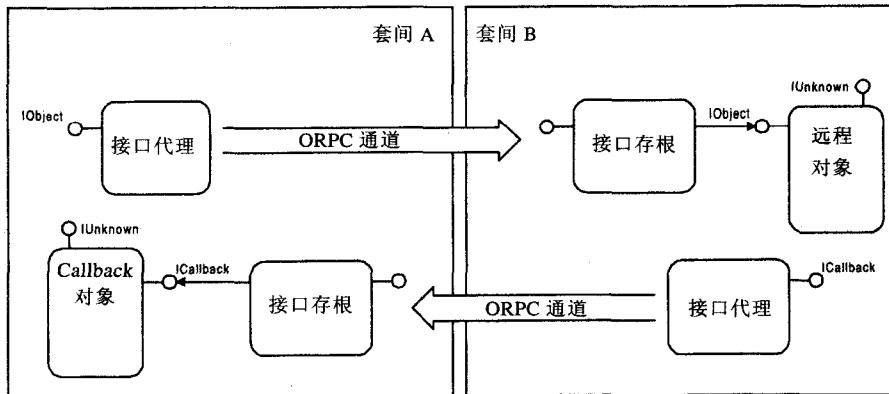


图 5.6 跨越套间的死锁

请回忆一下，当一个 [in] 参数被传递给代理的 `UseCallback` 方法时，代理调用 `CoMarshalInterface` 函数来列集 `ICallback` 接口指针。因为这个指针指向调用者套间中的一个对象，所以调用者套间变成了对象引出器，对于 `callback` 对象的所有跨套间调用都必须在调用者套间中被处理。当 `IObject` 接口存根散集 `ICallback` 接口时，它创建一个代理，并把代理传给 `UseCallback` 方法实现。这个代理代表了套间 B 与 `callback` 对象之间的短暂连接，仅在 `UseCallback` 方法调用过程中有效。如果 `UseCallback` 方法在这个代理上调用了 `AddRef` 的话，那么此代理或者连接的生命周期也可以超过 `UseCallback` 方法调用的范围：<sup>13</sup>

```
STDMETHODIMP Object::UseCallback(ICallback *pcb) {
    if (!pcb) return E_INVALIDARG;
    // 保持代理以备后用
    (m_pcbMyCaller = pcb)->AddRef();
    return S_OK;
}
```

于是，回到客户套间的连接将一直持续到远程对象把代理释放为止。因为所有的 COM 套间都可以接收 ORPC 请求，所以在任何时候只要对象愿意，它就可以回调到客户套间中。

不同类型的套间对于重入有不同的实现方式。MTA 的实现非常简单，因为 MTA 没有任何并发性保证，它们也不需要考虑给定的方法调用应该由哪个线程来处理。当 MTA 线程被阻塞在通道上，也就是正在等待 ORPC 应答的时候，如果这时有一个重入调用到达的话，那么接收重入请求的 RPC 线程进入到 MTA 中，并且 MTA 就用这个 RPC 线程来处理重入调用。“套间中另一个线程因等待 ORPC 应答而被阻塞”这样的事实与调用

<sup>13</sup> 一个常见的错误概念是：只有用连接点(Connection Point)才能建立起双向通信或者回调。正如第 7 章将要讲述的，只有当我们要在 Visual Basic 或者脚本环境中支持事件控制函数时，我们才必须使用连接点技术。

分发完全没有关系。在 RTA 的实现中，当一个正在 RTA 中运行的线程通过代理发出跨套间调用的时候，通道服从套间的控制，释放整个 RTA 范围内的锁，允许进来的调用可以被处理。而且，因为基于 RTA 的对象并没有线程相依性，所以接收 ORPC 请求的 RPC 线程一旦获得了 RTA 范围内的锁之后，它就可以进入到 RTA 中并处理该调用。

STA 的重入实现要复杂得多。因为基于 STA 的对象具有线程相依性，所以当一个线程从 STA 中发出跨套间调用时，COM 不允许这个线程发出阻塞式调用，这样的调用将会使所有的入 ORPC 请求无法得到处理。当调用者的线程进入到通道的 SendReceive 方法——发送 ORPC 请求并且接收 ORPC 应答时，通道偷取到调用者的线程，并把它放到一个内部的窗口消息 (MSG) 循环中。这与“此线程创建一个有模式对话框”的情形没有什么不同。在这两种情况下，当操作正在进行的时候，调用者线程都需要处理某些特定的窗口消息类型。对于有模式对话框的情形，线程需要处理基本的窗口消息，以保证所有的用户界面不至于出现被冻结的情况。对于跨套间 COM 方法调用的情形，线程不仅要处理通常的用户界面窗口消息，而且还要处理对应于入 ORPC 请求的窗口消息。缺省情况下，通道允许在客户线程等待 ORPC 应答的过程中，所有的入 ORPC 请求都可以被处理。我们可以为线程安装一个自定义的消息过滤器 (message filter) 来改变这种缺省行为。

消息过滤器是 STA 特有的。消息过滤器也是一个 COM 对象，每个 STA 只有一个消息过滤器，用来决定是否分发“入 ORPC 请求”。当 STA 线程在通道内部等待 ORPC 应答的时候，消息过滤器也可以用来处置未决的用户界面 (UI) 消息。消息过滤器暴露如下的 IMessageFilter 接口：

```
[ uuid(00000016-0000-0000-C000-000000000046), local, object ]
interface IMessageFilter : IUnknown {
    typedef struct tagINTERFACEINFO {
        IUnknown     *pUnk;           // 对象
        lid          lid;             // 接口
        WORD         wMethod;         // 方法
    } INTERFACEINFO;

    // 当“入 ORPC 请求”进入一个 STA 时调用
    DWORD HandleInComingCall(
        [in] DWORD dwCallType,
        [in] HTASK dwThreadIdCaller,
        [in] DWORD dwTickCount,
        [in] INTERFACEINFO *pInterfaceInfo
    );

    // 当另一个 STA 拒绝或延迟
    // ORPC 请求时调用
    DWORD RetryRejectedCall (
        [in] HTASK dwThreadIdCaller,
        [in] DWORD dwTickCount,
```

```

    [in] DWORD dwRejectType
};

// 当非 COM MSG 在线程等待一个
// ORPC 请求时调用
DWORD MessagePending(
    [in] HTASK dwThreadIdCaller,
    [in] DWORD dwTickCount,
    [in] DWORD dwPendingType
);
}

```

为了安装一个自定义的消息过滤器，COM 提供了 API 函数 `CoRegisterMessageFilter`：

```

HRESULT CoRegisterMessageFilter([in] IMessageFilter *pmfNew,
                                [out] IMessageFilter **ppmfOld),

```

`CoRegisterMessageFilter` 把参数中提供的消息过滤器与当前的 STA 关联起来。原先的消息过滤器被返回给调用者，这使得调用者以后可以恢复原先的消息过滤器。

无论什么时候当一个入 ORPC 请求到达 STA 线程时，消息过滤器的 `HandleIncomingCall` 方法就会被调用，这就给套间一个机会，它可以接受、拒绝或者推迟处理这个调用。`HandleIncomingCall` 既可以用于重入调用，也可以用于非重入调用。`dwCallType` 参数表示它接收到哪种类型的调用：

```

typedef enum tagCALLTYPE {
   CALLTYPE_TOPLEVEL,                      // STA 不在外出调用中
   CALLTYPE_NESTED,                        // 外出调用的回调
   CALLTYPE_ASYNC,                         // 异步调用
   CALLTYPE_TOPLEVEL_CALLPENDING,          // 等待时新调用
   CALLTYPE_ASYNC_CALLPENDING             // 等待时异步调用
} CALLTYPE;

```

当线程在通道中等待 ORPC 应答的时候，“嵌套的调用（重入的）”或者“顶级未决的（toplevel callpending）调用（非重入的）”就有可能发生。当套间中没有活动调用的时候，顶级调用就有可能发生。

COM 定义了一个枚举类型，`HandleIncomingCall` 的实现必须返回这种类型的值，以便指示如何处置该调用：

```

typedef enum tagSERVERCALL {
    SERVERCALL_ISHANDLED,                  // 接收调用并转给存根
    SERVERCALL_REJECTED,                   // 通知调用者调用被拒绝
    SERVERCALL_RETRYLATER                 // 通知调用者调用被延迟
} SERVERCALL;

```

如果消息过滤器的 `HandleIncomingCall` 返回 `SERVERCALL_ISHANDLED`，那么该

调用将被传递给接口存根进行散集处理。缺省消息过滤器总是返回 SERVERCALL\_ISHANDLED。如果 HandleIncomingCall 返回 SERVERCALL\_REJECTED 或者 SERVERCALL\_RETRYLATER，那么调用者的消息过滤器将会接收到相应的调用处置通知，并且 ORPC 请求也自然被丢弃。

当一个消息过滤器拒绝或者推迟了一个调用之后，调用者的消息过滤器会在 RetryRejectedCall 方法中接到相应的通知。这个调用（译注 2）是在调用者套间的环境中进行的，并且消息过滤器的 RetryRejectedCall 实现可以决定是否要对一个已经被推迟的调用进行重试。dwRejectType 参数指明这个调用是被拒绝还是被推迟的。调用者的通道将根据 RetryRejectedCall 的返回值来决定应该采取什么行动。如果 RetryRejectedCall 返回 -1，那么通道将假设不再进行重试，并且立即让代理返回 HRESULT RPC\_E\_CALL\_REJECTED。缺省的消息过滤器总是返回 -1。RetryRejectedCall 返回任何一个其他值，都被解释为重试调用之前所需等待的毫秒数。因为这个协商过程发生在通道的内部，所以代理不需要重新产生 ORPC 请求。实际上，接口列集器完全可以不去理会消息过滤器的活动。

当一个基于 STA 的线程被阻塞在通道中，等待 ORPC 应答的时候，与 COM 无关的窗口消息也可能会到达线程的 MSG 队列中。当这样的事件发生时，STA 的消息过滤器会在 MessagePending 方法中接到相应的通知。缺省的消息过滤器允许分发某些特定的窗口消息，以便使窗口系统不会被冻结；然而，输入事件（比如鼠标点击、键盘按下）会被丢弃，从而阻止最终用户开始新的交互操作。正如前面已经说过的，每个 STA 套间都有自己的消息过滤器，而 RTA 或者 MTA 则没有消息过滤器。消息过滤器使 COM 与用户界面线程（UI thread）结合得更好。这意味着所有的用户界面线程都应该运行在单线程套间中。大多数用户界面线程都将安装一个自定义的消息过滤器，以确保当应用进入到某些关键的阶段（在这些阶段中重入可能会引起错误）时，入调用不会被分发出去。消息过滤器不应该被用作通用的流程控制机制。当调用被拒绝或者被推迟的时候，消息过滤器的实现将会非常低效，这使得对于高吞吐量的应用来说，消息过滤器非常不适合于用作流程控制机制。

## 5.8 生命周期管理和列集

本章前面已经讨论了存根管理器和对象之间的关系。当第一次针对某个特定的对象实体调用 CoMarshalInterface 的时候，存根管理器就被创建起来了。存根管理器拥有一个未完结的引用，指向它所表达的对象，并且只要有一个未完结的外部引用指向该存根，

---

译注 2：指 RetryRejectedCall 调用。

那么存根管理器就会存活在内存中。这些外部引用往往就是代理，而且被列集之后的对象引用也被计算在内，因为它们代表了潜在的代理。一旦所有指向存根管理器的外部引用都被销毁之后，存根管理器就把自己删除掉，并且释放它所拥有的、指向实际对象的所有引用。这种缺省行为正好模拟了普通 AddRef 和 Release 的进程内语义。许多对象并没有特殊的生命周期要求，这种缺省行为就足以满足需要了。而有些对象则希望能够定制外部引用、存根管理器和对象三者之间的关系。幸运的是，COM 提供了足够的控制机制，可以把钩子插入到存根管理器的生命周期过程中，从而实现各种策略。为了理解存根生命周期的管理工作是如何进行的，我们有必要首先看一看 COM 的分布式垃圾回收算法。

当一个存根管理器被创建的时候，它的对象标识符（OID，Object Identifier）被注册到 COM 的分布式垃圾回收器中，目前 COM 的分布式垃圾回收器是由 OXID 解析器服务（OR 服务，OXID Resolver service）实现的。OR 记录了本地机器上每个套间引出了哪些 OID。当代理管理器被创建的时候，CoUnmarshalInterface 告诉本地 OR，有一个对象引用被引入到某个套间中了。这就意味着本地 OR 也知道本地机器上每个套间已经引入了哪些 OID。当某个特定的 OID 被首次引入到一台机器上时，引入主机的 OR 建立起它与引出主机 OR 之间的 ping 关系。引入方的 OR 将周期性地通过 RPC 发送 ping 消息，表示引入方主机仍然在运行，网络也没有问题。当前实现的 OR 每隔两分钟发送一次 ping 消息。如果在最近一个 ping 间隔中，并没有其他的 OID 被引入，那么 OR 只是发送一个简单的 ping 通知。如果有新的引用被引入，或者现有的引用被释放了，那么它将发送一个更加复杂的 ping 消息，指示了上一个引用集合与当前的引用集合之间的差异。

在 Windows NT 4.0 的 COM 版本下面，如果连续的三个 ping 间隔（六分钟）过去后，某个主机仍然没有接收到 ping 通知，那么 OR 就会假设远程主机已经崩溃了，或者由于网络失败的原因远程主机已经不可到达。这时候，OR 将通知每个“被该主机（当前已经死亡）引入”的存根管理器：未完结的引用已经无效了，它们应该可以被释放了。如果某个特定的对象仅仅被“这些已经死亡的机器”上的客户才使用的话，那么存根管理器将不再有其他的未完结引用，所以它就会删除自己，从而进一步释放指向对象的 COM 引用。

以上介绍的情形正是“当网络上的主机变得不可到达的时候”所发生的情况。更为有趣的情形是这样的情况：一个进程拥有未完结的代理，这时它要提前退出。如果一个进程在退出之前没有调用 CoUninitialize 适当的次数（比如进程非正常崩溃了），那么 COM 库将没有机会来清除那些被遗漏的引用。这时候，本地 OR 最终会检测到进程的死亡情况，并且在下一个 ping 消息中，删除已经被它引入的引用，这将使引出方的 OR 最终释放掉已经被引出的引用。如果该进程（译注 3）拥有指向本地机器上对象的

---

译注 3：指已死亡的客户进程。

引入引用，那么在检测到客户的死亡信息之后，这些引用马上就可以被释放了。<sup>14</sup>

COM 分布式垃圾收集器有时候也遭受到批评，因为它的效率不高。事实上，如果对象必须要可靠地检测到客户的存活情况，那么 COM 的方法有可能比那些“与具体应用相关的特殊解决方案”要高效得多。这是因为 COM 的垃圾收集器能够把那些与“某台特定机器持有的所有引用”相对应的“保持存活”消息合并为一个定期消息。与应用相关的技术并没有如此的全局知识，所以结果很可能是每个应用都有不同的“保持存活”握手过程，而不是整个主机共享同一套机制。对于有的应用来说，COM 的垃圾回收器确实影响了它们的性能，这种情况下，我们可以针对某些特定的存根管理器使用 MSHLFLAGS\_NOPING 标志，从而禁止 pinging 机制；然而，垃圾回收器的缺省行为对于大多数应用而言都是合适的，并且在性能上要超过其他许多“与具体应用相关的解决方案”。

存根管理器记录了有多少个外部引用是尚未完结的。当存根被创建的时候，这个计数值为 0。当使用 MSHLFLAGS\_NORMAL 调用 CoMarshalInterface 的时候，这个计数值被增加某个数 n，这个数被写到经过列集之后的对象引用中。当代理管理器散集这个引用的时候，它把 n 加到它所拥有的引用的计数器上。如果在代理管理器上 CoMarshalInterface 被调用，以便把引用的一份拷贝传递给另一个套间，那么代理管理器为了初始化第二个代理，它可以让出一定数目的引用。如果一个代理只剩下一个引用了，那么它必须回头向存根管理器请求更多的引用。

把列集之后的接口引用保存在可被一个或者多个客户访问的地方，这样做往往会有非常有用。有些名字对象实现所使用的运行对象表（Running Object Table，ROT），就是这样一个典范。如果被列集之后的接口指针是用 MSHLFLAGS\_NORMAL 标志创建的，那么只有一个客户可以散集这个对象引用。如果有多个客户要散集这个对象引用的话，那么该引用必须要用 MSHLFLAGS\_TABLESTRONG 或者 MSHLFLAGS\_TABLEWEAK 来列集获得。只要使用了这两个标志之一，那么被列集之后的对象引用就可以被散集多次。

强表格列集和弱表格列集之间的区别涉及到“被列集之后的对象引用”与“存根管理器”之间的关系。当使用 MSHLFLAGS\_TABLEWEAK 标志列集出一个对象引用时，存根管理器的外部引用计数器并没有被增加。这意味着，被列集之后的对象引用将包含 0 个引用，每个代理管理器需要与存根管理器进行联系，以便获得一个或者多个外部引用。因为弱表格列集的引用并不代表存根管理器上一个已经被计数了的外部引用，所以当最后一个代理管理器与存根管理器断开连接的时候，存根管理器就会销毁自己，当然也会释放任何它所拥有的、并且指向实际对象的 COM 引用。如果在此之前还没有代理管理器与存根管理器连接起来，那么在一段不确定的时间内，存根管理器仍然存活在内存中。这样带来的直接结果就是，经过列集之后的未完结对象引用并不会强迫存根管理器的生命周期，或者说不会强迫对象保留在内存中。相反，当使用 MSHLFLAGS\_TABLESTRONG

<sup>14</sup> 技术上讲，本地 OR 会等待一个比较短的周期，以确保死亡客户所创建的所有现存的、但是已经被列集的对象引用都有机会被散集(unmarshaled)。

标志列集出一个对象引用时，存根管理器的外部引用计数会随之增加。这意味着被列集之后的对象引用代表了存根管理器上一个已经被计数了的外部引用。与弱表格列集的情形一样，每个代理管理器必须要与存根管理器进行联系，才能获得一个或者多个额外的外部引用。因为强表格列集的引用代表了存根管理器上的一个外部引用计数，所以当最后一个代理管理器与存根管理器脱开连接的时候，存根管理器不会销毁自己，并且它实际上仍将拥有指向对象的 COM 引用。强表格列集的直接结果是，经过列集之后的未完结对象引用会影响存根管理器或者对象的生命周期。这意味着必须要有一种机制来释放这些被“强表格列集的对象引用”所拥有的引用。COM 提供了一个 API 函数 `CoReleaseMarshalData`，它可以用来告诉存根管理器“被列集之后的对象引用已经被销毁了”，函数的原型如下：

```
HRESULT CoReleaseMarshalData([in] IStream *pStm);
```

与 `CoUnmarshalInterface` 一样，`CoReleaseMarshalData` 以指向被列集之后对象引用的 `IStream` 接口指针作为参数。当一个表格列集不再有用的时候，我们应该调用 `CoReleaseMarshalData` 函数，以便注销掉这个表格列集。如果由于某些原因，一个经过常规列集（normal-marshaled）得到的对象引用没有被 `CoUnmarshalInterface` 散集的话，我们也应该调用 `CoReleaseMarshalData` 函数。

对象实现者也可以手工访问存根管理器的外部引用计数，以保证在对象生命周期的关键阶段存根管理器仍然存活。COM 提供了一个函数 `CoLockObjectExternal`，它可以增加或者减小存根管理器的外部引用计数，函数原型如下：

```
HRESULT CoLockObjectExternal([in] IUnknown *pUnkObject,
                           [in] BOOL block,
                           [in] BOOL bLastUnlockKillsStub);
```

`CoLockObjectExternal` 的第一个参数必须指向实际的对象，不能够指向代理。第二个参数 `bLock` 表示要增加或减小存根管理器的外部引用计数。第三个参数指示，如果这个调用减掉了最后一个外部引用的话，存根管理器是否要被删除。为了理解为什么 `CoLockObjectExternal` 是非常有必要的，请考虑这样一个对象：它要监视某个硬件设备，并且它已经被通过弱表格列集方式注册到 ROT 中了。当对象正在监视硬件设备的时候，它希望能够确保它的存根管理器仍然是有效的，以便新的客户可以与对象建立连接，然后检查硬件的状态。然而，如果对象当前并没有在监视硬件的状态，那么它会希望存根管理器从内存中消失，除非已经有未完结的代理与它建立了连接。为了实现这种功能，对象可能会有一个启动监视过程的方法：

```
STDMETHODIMP Monitor::StartMonitoring(void) {
    // 确保存根管理器/对象有效
    HRESULT hr = CoLockObjectExternal(this, TRUE, FALSE);
    // 开始硬件监视
```

```

if (SUCCEEDED(hr))
    hr = this->EnableHardwareProbe();
return hr;
}

```

以及另一个方法告诉对象可以结束监视过程了：

```

STDMETHODIMP Monitor::StopMonitoring(void) {
// 停止硬件监视
    this->DisableHardwareProbe();
// 允许存根管理器/对象在无客户时消失
    hr = CoLockObjectExternal(this, FALSE, TRUE);
    return hr;
}

```

假定此对象最初是用弱表格列集方式被列集的，上述代码保证，只要至少有一个未完结的代理对象，或者对象当前正在监视底层的硬件设备，那么存根管理器和对象就会存活在内存中。

除了允许对象实现者可以调整存根管理器的外部引用计数之外，COM 还允许对象实现者显式地删除存根管理器，而无须考虑未完结对象引用的数目。COM 提供了一个 API 函数 CoDisconnectObject，它会找到对象的存根管理器，然后把它销毁，断开所有的代理，函数原型如下：

```

HRESULT CoDisconnectObject(
    [in] IUnknown * pUnkObject,           // 对象指针
    [in] DWORD dwReserved);             // 保留，必须为 0

```

与 CoLockObjectExternal 一样，CoDisconnectObject 必须在对象所在的进程内被调用，并且不能在代理上被调用。为了使 CoDisconnectObject 也能够适用于前面展示的硬件监视对象，请考虑：如果对象的状态受到了破坏，那么事情会怎么样呢？为了防止对该对象的其他方法调用返回错误的结果，对象可以调用 CoDisconnectObject，强制性地断开它与所有未完结代理之间的连接，示例代码如下：

```

STDMETHODIMP Monitor::GetSample(/*[out]*/ SAMPLEDATA *ps) {
    HRESULT hr = this->GetSampleFromProbe(ps);
    if (FAILED(hr)) // 监视或对象可能损坏
        CoDisconnectObject(this, 0);
    return hr;
}

```

当一个进程希望终止，但是它的某些对象仍然有尚未完结的代理的时候，我们也可以使用 CoDisconnectObject 函数。对于可能具有尚未完结的代理的对象，在销毁这些对象之前先显式地调用 CoDisconnectObject，这样做可以避免“对象已经被销毁，却仍然

有针对该对象的入 ORPC 请求需要处理”的危险。如果在对象被销毁之后，但是存根管理器仍然存活在内存中，这时仍有入 ORPC 请求到达的话，接口存根会不加考虑到调用到“原先被认为是对象的内存”所在的方法，从而引起不必要的编程痛苦。

`CoLockObjectExternal` 和 `CoDisconnectObject` 都是对象实现者用来控制存根管理器的技术。如果能够知道存根管理器上是否存在未完结的代理，或者是否存在强类型列集，这往往会非常有用。为了告知对象“存根管理器上确实有未完结的外部引用”，COM 定义了一个接口 `IExternalConnection`，对象可以引出这个接口。接口的定义如下：

```
[ uuid(000001g-0000-0000-C000-00000000046), Object, local ]
interface IExternalConnection : IUnknown {
    DWORD AddConnection(
        [in] DWORD extconn,           // 引用类型
        [in] DWORD reserved          // 保留，必须为 0
    );
    DWORD ReleaseConnection(
        [in] DWORD extconn,           // 引用类型
        [in] DWORD reserved,          // 保留，必须为 0
        [in] BOOL fLastReleaseCloses // 销毁存根
    );
}
```

当存根管理器第一次与对象连接时，它询问对象是否愿意在“外部引用被创建或者被销毁”的时候接收到这样的通知。它通过 `QueryInterface` 请求 `IExternalConnection` 接口来实现这一点。如果对象没有实现 `IExternalConnection`，那么存根管理器将使用它自己的引用计数器来决定何时销毁存根管理器。如果对象实现了 `IExternalConnection` 接口，那么存根管理器会一直生存下去，直到对象调用 `CoDisconnectObject` 显式地销毁它为止。

对于实现了 `IExternalConnection` 接口的对象，它应该维护一个锁计数器，用来记录 `AddConnection` 和 `ReleaseConnection` 调用的数目。为考虑效率起见，COM 并不会在每次创建代理的时候都调用 `AddConnection`。这意味着，如果对象根据 `AddConnection` 和 `ReleaseConnection` 调用来维护锁计数器，那么对象的锁计数器并不能精确地反映出当前现有对象引用的数目。然而，COM 保证，只要锁计数值不为 0，那么至少会有一个未完结引用存在，而锁计数值为 0 表示不再存在未完结的引用。`CoLockObjectExternal` 调用也会影响到这个计数值。对于那些很关心“外部客户是否存在”的对象来说，这样的信息特别有用。例如，假定前面展示的硬件监视对象创建了一个线程，专门用于在后台记录采样数据。同时也假设，如果在“对象正在监视数据，或者正在被外部客户控制”的时候记录数据的话，采样数据就有可能发生错误。为了这个问题，记录数据的线程可以检查由“对象的 `IExternalConnection` 实现代码”维护的锁计数器，并且只有当不存在外部引用的时候，它才执行记录操作。所以，对象应该按下面的方式来实现 `IExternalConnection`：

```
class Monitor : public IExternalConnection,
```

```

public IMonitor {
    LONG m_cRef; // 常规 COM 引用计数
    LONG m_cExtRef; // 外部引用计数
    Monitor(void) : m_cRef(0), m_cExtRef(0) { ... }
    STDMETHODIMP_(DWORD) AddConnection(DWORD extconn, DWORD) {
        if (extconn & EXTCONN_STRONG) // 必须检测此位
            return InterlockedIncrement(&m_cExtRef);
    }

    STDMETHODIMP_(DWORD) ReleaseConnection(DWORD extconn, DWORD,
                                           BOOL bLastUnlockKillsStub) {
        DWORD res = 0;
        if (extconn & EXTCONN_STRONG){ // 必须检测此位
            res = InterlockedDecrement(&m_cExtRef);
            if (res == 0 && bLastUnlockKillsStub)
                CoDisconnectObject(this, 0);
        }
        return res;
    }
    :
    :
}

```

有了这样的实现之后，线程函数就可以检查对象的状态，根据对象的活动层次，来决定是否执行记录操作，代码如下：

```

DWORD WINAPI ThreadProc(void *pv) {
    // 假定实际对象指针传给了 CreateThread
    Monitor *pm = (Monitor*)pv;
    while (1) {
        // 休眠 10 秒
        Sleep(10000);
        // 如对象不再使用，执行记录操作
        if (pm->m_cExtRef == 0)
            pm->TryToLogSampleData();
    }
    return 0;
}

```

假定对象的 TryToLogSampleData 方法能够正确地处理并发过程，那么只有当对象没被外部客户使用，或者没有处于监测状态的时候（前面讲到过，当对象开始监视硬件设备的时候，它会调用 CoLockObjectExternal 提高外部引用计数），这个线程函数才会记录数据。尽管这个例子看起来有人为构造的痕迹，但是确实存在这样的情形，跟踪外部引用计数对于有些操作至关重要。第 6 章讨论了一个典型的例子，它与进程外服务器中类对象注册过程有关。

## 5.9 自定义列集

到目前为止，本章一直在讨论标准列集，以及基于 ORPC 的方法远程传递过程。对于大多数对象而言，我们所需要做的是如何在性能、语义的正确性以及实现的简单性之间做出平衡。但是有些对象，ORPC 远程方法提供的缺省行为不能达到它们所要求的效率，从而使它们很不实用。对于这样的对象，COM 支持自定义列集方式。正如本章前面所提到的，自定义列集使得对象实现者可以提供专门的代理实现，供引入套间创建这样的代理。只要对象引出 IMarshal 接口，这就表明它希望支持自定义列集方式，IMarshal 接口的定义如下：

```
[ uuid(0000003-0000-0000-C000-000000000046), local, object ]
interface IMarshal : IUnknown {
    // 获取自定义代理的 CLSID (CoMarshalInterface)
    HRESULT GetUnmarshalClass(
        [in] REFIID riid, [in, iid_is(riid)] void *pv,
        [in] DWORD dwDestCtx, [in] void *pvDestCtx,
        [in] DWORD mshlflags, [out] CLSID *pcclsid);

    // 获取自定义列集对象引用的大小 (CoGetMarshalSizeMax)
    HRESULT GetMarshalSizeMax(
        [in] REFIID riid, [in, iid_is(riid)] void *pv,
        [in] DWORD dwDestCtx, [in] void *pvDestCtx,
        [in] DWORD mshlflags, [out] DWORD *pSize);

    // 写自定义列集对象引用的大小 (CoMarshalInterface)
    HRESULT MarshalInterface([in] IStream *pStm,
        [in] REFIID riid, [in, iid_is(riid)] void *pv,
        [in] DWORD dwDestCtx, [in] void *pvDestCtx,
        [in] DWORD mshl flags);

    // 读对象引用并返回代理 (CoUnmarshalInterface)
    HRESULT UnmarshalInterface([in] IStream *pStm,
        [in] REFIID riid,
        [out, iid_is(riid)] void **ppv);

    // 释放列集 (CoReleaseMarshalData)
    HRESULT ReleaseMarshalData([in] IStream *pStm);
    // 关闭连接状态 (CoDisconnectObject)
    HRESULT DisconnectObject([in] DWORD dwReserved);
}
```

在上面的代码中，每个方法定义前面的注解说明了哪个 COM API 函数将会调用该

方法。

当针对一个支持自定义列集的对象调用 CoMarshalInterface 的时候，被列集之后的对象引用的格式有所不同（与标准列集方式比较），如图 5.7 所示（对照图 5.2）。注意，在标准的 MEOW 头部之后，该对象引用只是简单地包含一个 CLSID 和一个字节数组，此 CLSID 将被用于创建自定义代理，而字节数组则用来对它进行初始化。CoMarshalInterface 调用对象的 IMarshal::GetUnmarshalClass 方法，从而发现自定义代理的 CLSID。然后它调用对象的 IMarshal::MarshalInterface 方法，由它填充字节数组。在 MarshalInterface 方法中，这是对象唯一的机会——给新建的自定义代理发送初始消息。做法很简单，只要把消息写到 MarshalInterface 的字节流参数中。在 CoUnmarshalInterface 被调用的时候，这个消息将被交给新建的自定义代理（通过代理的 IMarshal::Unmarshal Interface 方法）。这意味着对象和自定义代理都必须要实现 IMarshal 接口。对象的 MarshalInterface 方法把初始消息写到字节流中；代理的 UnmarshalInterface 方法读入此初始消息。一旦 UnmarshalInterface 方法返回，COM 就不再介入到代理和对象之间的通信过程。自定义代理有责任按照正确的语义来实现 IMarshal 接口的方法。如果一个方法调用要被远程传递到对象上，那么代理有责任完成这项工作。如果该方法可以在客户的套间中完成，那么代理也同样要负起责任来。

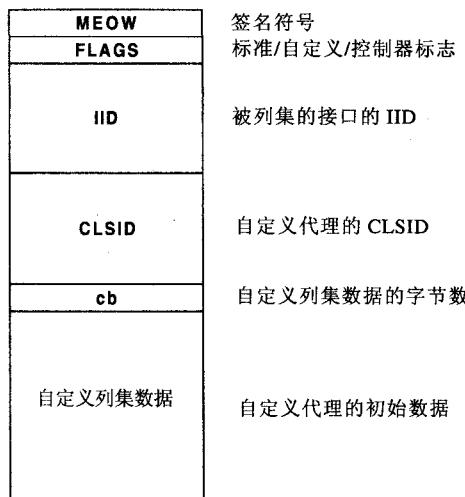


图 5.7 自定义列集对象引用

自定义列集的优势之一是，客户并不会注意到它正在使用自定义列集。实际上，客户并不能够很可靠地检测到某个接口引用到底是标准代理、还是自定义代理，或者是实际的对象。自定义列集是一个以对象为基本单位的策略（object-by-object decision）。同一个类的两个实例可以各自独立地选择使用标准列集或者自定义列集。当一个对象选择了实现自定义列集时，它必须对所有的接口都这样做。如果一个对象只希望对某些列集环境（比如，进程内环境、本地机器环境，或者远程主机环境）使用自定义列集方式，

那么对象可以先获得一个标准列集器实例，然后对于不支持自定义列集的环境，直接把标准列集器的 IMarshal 接口方法转发过去，于是所有的环境都得到了预期的支持。实际上，如果一个对象只是简单地、无条件地把所有 IMarshal 的方法都转发给标准列集器，那么这个对象其实就是在使用标准列集方式。为了得到指向标准列集器的指针，对象可以调用 CoGetStandardMarshal 方法：

```
HRESULT CoGetStandardMarshal(
    [in]REFIID riid,                      // itf 类型列集吗?
    [in, iid_is(riid)] IUnknown *pUnk,      // 列集的 itf
    [in] DWORD dwDestCtx,                  // MSHCTX
    [in] void *pvDestCtx,                  // 保留
    [in] DWORD mshlflags,                  // 常规还是表格
    [out] IMarshal **ppMarshal);          // 标准列集指针
```

假定一个对象只在本地主机上工作时才使用自定义列集技术，而在与远程套间进行通信时不使用自定义列集技术。那么对象的 GetMarshalSizeMax 方法实现会与下面的代码类似：

```
STDMETHODIMP CustStd::GetMarshalSizeMax(ULONG *pcb,
                                         REFIID riid, void *pv, DWORD dwDestCtx,
                                         void *pvDestCtx, DWORD mshlflags) {
    // 如环境支持，则执行
    if (dwDestCtx==MSHCTX_LOCAL || dwDestCtx==MSHCTX_INPROC)
        return this->MyCustomMarshalingRoutine(pcb);
    // 环境不支持，授权给标准列集
    IMarshal *pMsh = 0;
    HRESULT hr = CoGetStandardMarshal(riid, pv, dwDestCtx,
                                      pvDestCtx, mshlflags,
                                      &pMsh);
    if (SUCCEEDED(hr)) {
        hr = pMsh->GetMarshalSizeMax(pcb, riid, pv, dwDestCtx,
                                       pvDestCtx, mshlflags);
        pMsh->Release();
    }
    return hr;
}
```

上面的代码片断并没有显示出“当对象希望使用自定义列集时，它如何写初始消息”。这是因为 IMarshal 的所有方法都没有标准的实现模式（所以称之为自定义列集）。然而，在有些常见的情形下，自定义列集还是非常有意义的，而且这时候 IMarshal 接口的实现往往有一定的规则可以遵循。到目前为止，最常见的 IMarshal 应用是实现“按值列集（marshal-by-value）”。

“按值列集”对于那些“一旦初始化之后，其状态再也不会发生变化”的对象特别有用。对结构（structure）以 COM 对象的形式进行包装就是这样一个典型的例子，它

只需要被初始化，然后被传给另一个需要使用该结构中数据的对象，之后这个包装对象就被销毁掉。这样的对象就是自定义列集最主要候选。为了实现“按值列集”，对象实现几乎总是进程内服务器。这使得对象和代理可以共享同样的实现类。隐藏在“按值列集”背后的思想是，自定义代理变成原始对象的复制品。这隐含着被列集的对象引用必须包含原始对象的完整状态信息，而且为简化起见，自定义代理的 CLSID 必须与原来对象的 CLSID 相同。

假定下面的类定义是一个 COM 包装类，它封装了一个简单的二维点坐标：

```
class Point : public IPPoint, public IMarshal {
    long m_x; long m_y;
public:
    Point(void) : m_x(0), m_y(0) {}
    IMPLEMENTS_UNKNOWN(Point)
    BEGIN_INTERFACE_TABLE(Point)
        IMPLEMENTS_INTERFACE(IPPoint)
        IMPLEMENTS_INTERFACE(IMarshal)
    END_INTERFACE_TABLE()
    // IPPoint 方法
    // IMarshal 方法
};
```

为了支持“按值列集”，Point 类的 MarshalInterface 方法必须把对象的状态序列化，即写到流（IStream）中，作为代理的初始消息。代码如下：

```
STDMETHODIMP Point::MarshalInterface(IStream *pStm, REFIID,
                                      void *, DWORD, void *, DWORD) {
// 写出首部
    DWORD dw = 0xFF669900;
    HRESULT hr = pStm->Write(&dw, sizeof(DWORD), 0);
    if (FAILED(hr)) return hr;
    dw = m_x;
    hr = pStm->Write(&dw, sizeof(DWORD), 0);
    if (FAILED(hr)) return hr;
    dw = m_y;
    return pStm->Write(&dw, sizeof(DWORD), 0);
}
```

假设对象的类是以进程内服务器的形式被实现的，那么自定义代理可以是同一个类的第二个实例，这就隐含着 GetUnmarshalClass 的代码应该如下所示：

```
STDMETHODIMP Point::GetUnmarshalClass(REFIID, void *, DWORD,
                                      void *, DWORD, CLSID *pclsid) {
    *pclsid = CLSID_Point; // 此类的 CLSID
    return hr;
}
```

为了保证初始消息有足够的空间被分配，对象的 `GetMarshalSizeMax` 方法必须要返回正确的字节数：

```
STDMETHODIMP Point::GetMarshalSizeMax(REFIID, void *, DWORD,
                                      void *, DWORD, DWORD *pcb) {
    *pcb = 3 * sizeof(DWORD); // m_x + m_y + 首部
    return hr;
}
```

当列集之后的对象引用被 `CoUnmarshalInterface` 函数散集的时候，“此对象引用是一个自定义列集”这一事实将使得一个新的自定义代理被创建出来。此对象引用包含了自定义代理的 `CLSID`，也就是原始对象在 `GetUnmarshalClass` 方法中返回的 `CLSID`。新的自定义代理被创建起来之后，原始对象在 `MarshalInterface` 方法中写入的初始消息将被交给这个新代理的 `UnmarshalInterface` 方法：

```
// 读首部
DWORD dw; ULONG cbRead;
HRESULT hr = pStm->Read(&dw, sizeof(DWORD), &cbRead);
if (FAILED(hr) || cbRead != sizeof(DWORD))
    return RPC_E_INVALID_DATA;
bool bSwapEndian = dw == 0x009966FF;
// 读 m_x 和 m_y
hr = pStm->Read(&dw, sizeof(DWORD), &cbRead); m_x = dw;
if (FAILED(hr) || cbRead != sizeof(DWORD))
    return RPC_E_INVALID_DATA;
hr = pStm->Read(&dw, sizeof(DWORD), &cbRead); m_y = dw;
if (FAILED(hr) || cbRead != sizeof(DWORD))
    return RPC_E_INVALID_DATA;
// 如必要字节交换成员
if (bSwapEndian)
    byteswapdata(&m_x, &m_y);
// 返回对象指针
return this->QueryInterface(riid, ppv);
}
```

注意，`MarshalInterface` 和 `UnmarshalInterface` 的实现必须要保证，被列集的状态信息在任何平台上都是可读的。这意味着实现者必须要手工处理对齐方式、字节顺序，以及数据类型大小的差异等。

前面给出的 `UnmarshalInterface` 实现只是简单地返回一个指向新创建的自定义代理的指针。对于简单的“按值列集”对象，这可能已经很令人满意了。然而，更为一般的 `UnmarshalInterface` 实现可能还要检测出对应于同一个 COM 实体对象的多个散集，从而返回指向同一个代理实体的指针，以便维护代理与对象之间的实体对应关系。这不仅可以节省资源，同时也保持编程模型更加清晰。

## 5.10 自由线程列集器

当一个类被标记为 ThreadingModel=“Both”的时候，这就指明了它的实例和类对象既可以安全地驻留在 STA 中，也可以驻留在 MTA 中。然而，根据 COM 的规则，任何给定的实例都只能驻留在一个套间中。如果对象实现者能够保证某个对象可以安全地驻留在 MTA 中，那么有可能这个对象根本不需要关心套间。这样的对象既可以被 MTA 中多个线程并发访问，也可以被非 MTA 线程（也就是在 STA 中执行的线程）并发访问。然而，对于特定的对象，客户是不可能知道这样的访问是否是安全的，因此，所有跨套间的接口指针共享必须要用显式的列集技术建立起来。这意味着，对进程内对象的访问也要通过 ORPC 调用，除非调用者也在原先创建对象的套间中运行。

与客户不同的是，对象知道它们与套间之间的关系，也知道它们与并发性 (concurrency)、重入性 (reentrancy) 之间的关系。当对象被同一进程中多个套间访问的时候，凡是满足于 ORPC 调用的对象在缺省情况下都可以得到这种行为知识。然而，对于不满足于 ORPC 调用访问的对象，它们可以实现自定义列集，从而绕过这一过程。公平地讲，自定义列集很容易就能够绕过存根管理器，它只要直接把指向对象的原始指针写到经过列集之后的对象引用中。如果采用了这项技术，那么自定义代理只要简单地把原始指针从列集之后的对象引用中读出来，然后传给引入套间中的调用者即可。客户线程有可能会通过 CoMarshalInterface 和 CoUnmarshalInterface 函数（不管是显式地还是隐式地），在不同的套间之间传递此接口指针；然而，对象将与自定义代理合作，只是简单地把指向实际对象的原始指针传递出去。但是这项技术仅仅对于进程内部的列集过程能够很好地工作，无法用于进程之间的列集过程。幸运的是，对于除了 MSHCTX\_INPROC 之外的任何一种列集环境，对象实现都可以简单地把列集工作委托给标准列集器。

因为上面描述的行为对于绝大多数类型的对象都是有用的，所以 COM 提供了 IMarshal 接口的一个可聚合实现版本，它可以完成上面所讲述的功能。这份实现被称为自由线程列集器 (FTM, Free Threaded Marshaler)，我们可以使用 CoCreateFreeThreadedMarshaler API 函数来创建 FTM，CoCreateFreeThreadedMarshaler 函数的原型如下：

```
HRESULT CoCreateFreeThreadedMarshaler(
    [in] IUnknown *pUnkOuter,
    [out] IUnknown **ppUnkInner);
```

如果一个类要想使用 FTM，它只要简单地聚合一个 FTM 实例，既可以在初始化的时候，也可以在第一次 QueryInterface 请求 IMarshal 的时候根据需要完成这项工作。下面的类在构造时刻预先初始化一个 FTM：

```

class Point : public IPPoint {
    LONG m_cRef; IUnknown *m_pUnkFTM;
    long m_x; long m_y;
    Point(void) : m_cRef(0), m_x(0), m_y(0) {
        HRESULT hr = CoCreateFreeThreadedMarshaler(this,
            &m_pUnkFTM);
        assert(SUCCEEDED(hr));
    }
    virtual ~Point(void) { m_pUnkFTM->Release(); }
};

```

对应的 QueryInterface 实现只是简单地向 FTM 请求 IMarshal 接口：

```

STDMETHODIMP Point::QueryInterface(REFIID riid, void **ppv) {
    if (riid == IID_IUnknown || riid == IID_IPPoint)
        *ppv = static_cast<IPPoint*>(this);
    else if (riid == IID_IMarshal)
        return m_pUnkFTM->QueryInterface(riid, ppv);
    else
        return (*ppv = 0), E_NOINTERFACE;
    ((IUnknown*)*ppv)->AddRef();
    return S_OK;
}

```

一旦已经有了 FTM，不管什么时候指向 Point 对象的引用被汇集到其他套间中，它永远也不需要用到代理。这既适用于显式的 CoMarshalInterface/CoUnmarshalInterface 调用，也适用于当指向 Point 对象的引用被作为方法参数传递给“位于同一进程内的、指向非 Point 对象的代理”的时候。

FTM 至少要消耗 16 字节的内存。因为许多进程内对象永远也不会在它们的原始套间之外被使用，所以预先分配一个 FTM 并不是最佳的资源利用策略。而且很有可能对象将会有某种线程同步原语。如果确实是这样的话，那么 FTM 可以在第一次通过 QueryInterface 请求 IMarshal 接口的时候被“懒聚合（lazy-aggregated）”。为了做到这一点，不妨假设下面的类定义：

```

class LazyPoint : public IPPoint {
    LONG m_cRef; IUnknown *m_pUnkFTM;
    long m_x; long m_y;
    LazyPoint(void) : m_cRef(0), m_pUnkFTM(0), m_x(0), m_y(0) {}
    virtual ~LazyPoint(void)
    { if (m_pUnkFTM) m_pUnkFTM->Release(); }
    void Lock(void); // 获取对象专有锁
    void Unlock(void); // 释放对象专有锁
    :
    :
};

```

根据以上的类定义，下面的 `QueryInterface` 实现将可以正确地按需（on-demand）聚合 FTM：

```
STDMETHODIMP Point::QueryInterface(REFIID riid, void **ppv) {
    if (riid == IID_IUnknown || riid == IID_IPoint)
        *ppv = static_cast<IPoint*>(this);
    else if (riid == IID_IMarshal) {
        this->Lock();
        HRESULT hr = E_NOINTERFACE; *ppv = {};
        if (m_pUnkFTM == 0) // 第一次获取 FTM
            CoCreateFreeThreadedMarshaler(this, &m_pUnkFTM);
        if (m_pUnkFTM != 0) // 到此已获取了 FTM
            hr = m_pUnkFTM->QueryInterface(riid, ppv);
        this->Unlock();
        return hr;
    }
    else
        return (*ppv = 0), E_NOINTERFACE;
    ((IUnknown*)*ppv)->AddRef();
    return S_OK;
}
```

这种方法的缺点是，所有对 `IMarshal` 的 `QueryInterface` 请求都将被序列化（serialized）；然而，如果 `IMarshal` 接口从来也没有被请求的话，那么它几乎不会占用其他的资源。

以上介绍了 FTM 相对比较容易的用法之后，我们进一步来考虑什么时候不适合使用 FTM。很显然，只能生存在单线程套间中的对象不应该使用 FTM，因为它们几乎不可能期望会有并发性访问。然而，这并不意味着，所有能够运行在 MTA 中的对象都应该使用 FTM。如果一个对象具有接口指针类型的数据成员，那么它在使用 FTM 的时候必须要非常谨慎。考虑下面的类，它利用其他的 COM 对象来执行它自己的操作：

```
class Rect : public IRect {
    LONG m_cRef;
    IPoint *m_pPtTopLeft;
    IPoint *m_pPtBottomRight;
    Rect(void) : m_cRef(0) {
        HRESULT hr = CoCreateInstance(CLSID_Point, 0,
                                      CLSCTX_INPROC, IID_IPoint, (void**)&m_pPtTopLeft)
        assert(SUCCEEDED(hr));
        hr = CoCreateInstance(CLSID_Point, 0, CLSCTX_INPROC,
                             IID_IPoint, (void**)&m_pPtBottomRight);
        assert(SUCCEEDED(hr));
    }
    :
    :
}
```

假定类 Rect 是一个进程内的 COM 类，并且它被标记为 ThreadingModel=“Both”。对于给定的 Rect 对象，它的构造函数总是在调用 CoCreateInstance (CLSID\_Rect) 的线程所在的套间中执行。这意味着，两个 CoCreateInstance (CLSID\_Point) 调用也在客户的套间中执行。COM 的规则表明，只能执行 CoCreateInstance 调用的套间才可以访问数据成员 m\_pPtTopLeft 和 m\_pPtBottomRight。

很可能 Rect 至少有一个方法会利用这两个接口指针数据成员来执行其工作：

```
STDMETHODIMP Rect::get_Area(long *pn) {
    long top, left, bottom, right;
    HRESULT hr = m_pPtTopLeft->GetCoords(&left, &top);
    assert (SUCCEEDED(hr));
    hr = m_pPtBottomRight->GetCoords(&right, &bottom);
    assert (SUCCEEDED(hr));
    *pn = (right - left) * (bottom - top);
    return S_OK;
}
```

如果类 Rect 将使用 FTM，那么很有可能除了发出原始 CoCreateInstance 调用之外的其他套间也会调用这个方法。不幸的是，这将使 get\_Area 方法违反 COM 的规则，因为这两个接口指针只在原来的套间中才有效。如果 Point 类碰巧也使用 FTM 的话，那么从技术上讲，这将不再是个问题。然而，一般情况下，客户（在这里也就是 Rect 类）不应该对如此具体的实现细节做出这样的假设。实际上，如果 Point 对象没有使用 FTM，而且由于线程模型（Threading Model）不兼容的原因，它也是在不同的线程中被创建的，那么 Rect 对象所拥有的成员接口指针将指向 Point 对象的代理。代理当然会遵循 COM 的基本规则（代理是保证贯彻 COM 规则的有效手段），当客户在错误的套间中访问代理时，它会返回 RPC\_E\_WRONG\_THREAD。

这使 Rect 的实现者面临两个选择。第一个选择是不使用 FTM，并且接受这样的事实：当客户在套间之间传递 Rect 对象时，它将使用 ORPC 来访问类 Rect 的实例。这是非常容易的解决方案，因为这样做不会涉及到附加的代码，而且无须费神思索，它肯定可以正常工作。另一种选择方案是，不再保留未经处理的接口指针（raw interface pointer）作为数据成员，而是把接口指针被列集之后的形式作为数据成员保留下来。这正是 GIT 要做的事情。为了实现这种方法，Rect 类必须保留 DWORD 类型的 cookie 作为数据成员，代替未经处理的接口指针，类定义如下：

```
class SafeRect : public IRect {
    LONG m_cRef;           // COM 引用计数
    IUnknown *m_pUnkFTM;   // 缓存 FTM 懒聚合
    DWORD m_dwTopLeft;     // 顶/左 GIT cookie
    DWORD m_dwBottomRight; // 底/右 GIT cookie
    : : :
}
```

构造函数仍然创建两个 Point 实例，但是它不再直接记录这两个接口指针，而是把接口指针注册到 GIT 中：

```
SafeRect::SafeRect(void) : m_cRef(0), m_pUnkFTM(0) {
    // 假定 GIT 指针已初始化
    extern IGlobalInterfaceTable *g_pGIT; assert(g_pGIT != 0);
    IPoInt *pPoint = 0;
    // 实例化 Point 类
    HRESULT hr = CoCreateInstance(CLSID_Point, 0,
        CLSCTX_INPROC, IID_IPoInt, (void**)&pPoint);
    assert(SUCCEEDED(hr));
    // 在 GIT 中注册接口指针
    hr = g_pGIT->RegisterInterfaceInGlobal(pPoint, IID_IPoInt,
        &m_dwTopLeft);
    assert(SUCCEEDED(hr));
    pPoint->Release(); // 引用存于 GIT
    // 实例化 Point 类
    hr = CoCreateInstance(CLSID_Point, 0, CLSCTX_INPROC,
        IID_IPoInt, (void**)&pPoint);
    assert(SUCCEEDED(hr));
    // 在 GIT 中注册接口指针
    hr = g_pGIT->RegisterInterfaceInGlobal(pPoint, IID_IPoInt,
        &m_dwBottomRight);
    assert(SUCCEEDED(hr));
    pPoint->Release(); // 引用存于 GIT
}
```

注意，一旦接口指针被注册到 GIT 中之后，接口指针的用户就不必再保留其他的引用了。

当一个类已经用 GIT 方案代替了直接的接口指针方案之后，在每个“需要用到这些被注册的接口”的方法调用内部，它必须散集出一个新的代理以供使用，例如：

```
STDMETHODIMP SafeRect::get_Area(long *pn) {
    extern IGlobalInterfaceTable *g_pGIT; assert(g_pGIT != 0);
    // 从 GIT 散集两个接口指针
    IPoInt *ptl= 0, *pbr = 0;
    HRESULT hr = g_pGIT->GetInterfaceFromGlobal(m_dwPtTopLeft,
        IID_IPoInt, (void**)&ptl);
    assert(SUCCEEDED(hr));
    hr = g_pGIT->GetInterfaceFromGlobal(m_dwPtBottomRight,
        IID_IPoInt, (void**)&pbr);
    // 使用临时指针实现方法
    long top, left, bottom, right;
    hr = ptl->GetCoords(&left, &top);
    assert(SUCCEEDED(hr));
    hr = pbr->GetCoords(&right, &bottom);
    assert(SUCCEEDED(hr));
```

```

        *pn = (right - left) * (bottom - top);
    // 释放临时指针
    ptl->Release();
    pbr->Release();
    return S_OK;
}

```

由于 SafeRect 的实现使用了 FTM，所以在不同的方法调用之间，企图保留散集之后得到的接口指针是没有意义的，因为它无法知道下一个方法调用是否会在同一个套间中。

GIT 会使所有经过注册的接口指针保持存活状态，直到它们被显式地从 GIT 中删除为止。这意味着 SafeRect 必须要显式地注销掉对应于它的两个数据成员的 GIT 项：

```

SafeRect::~SafeRect(void) {
    extern IGlobalInterfaceTable *g_pGIT; assert(g_pGIT != 0);
    HRESULT hr=g_pGIT->RevokeInterfaceFromGlobal(m_dwTopLeft);
    assert(SUCCEEDED(hr));
    hr = g_pGIT->RevokeInterfaceFromGlobal(m_dwBottomRight);
    assert(SUCCEEDED(hr));
}

```

从 GIT 中删除一个接口指针，同时也会释放该对象上的引用。

注意，GIT 和 FTM 一起使用隐含着：为了要在每个独立的方法中使用被注册的对象，这些方法必须要频繁地调用 GIT，以便创建临时的接口指针。虽然针对这种使用模式，GIT 已经作了优化，但是程序代码仍然非常繁琐。下面的 C++类非常简单，它把 GIT 的 cookie 的用法包装成一个方便的、类型安全的接口：

```

template <class Itf, const IID* piid>
class GlobalInterfacePointer {
    DWORD m_dwCookie; // GIT cookie
// 防止误用
    GlobalInterfacePointer (const GlobalInterfacePointer&);
    void operator =(const GlobalInterfacePointer&);
public:
// 以无效 cookie 开始
    GlobalInterfacePointer(void) : m_dwCookie(0) { }
// 以自动全局化的局部指针开始
    GlobalInterfacePointer(Itf *pItf, HRESULT& hr)
        : m_dwCookie(0) { hr = Globalize(pItf); }

// 自动非全局化
    ~GlobalInterfacePointer (void)
    { if (m_dwCookie) Unglobalize(); }

// 在 GIT 中注册接口指针
    HRESULT Globalize(Itf *pItf) {
        assert(g_pGIT != 0 && m_dwCookie == 0);

```

```

    return g_pGIT->RegisterInterfaceInGlobal (pItf,
                                              *piid, &m_dwCookie);
}

// 销毁 GIT 中的一个接口指针
HRESULT Unglobalize(void) {
    assert(g_pGIT != 0 && m_dwCookie != 0);
    HRESULT hr=g_pGIT->RevokeInterfaceFromGlobal (m_dwCookie);
    m_dwCookie = 0;
    return hr;
}

// 从 GIT 得到一个局部接口指针
HRESULT Localize(Itf **ppItf) const {
    assert(g_pGIT != 0 && m_dwCookie != 0);
    return g_pGIT->GetInterfaceFromGlobal (m_dwCookie,
                                            *piid, (void**) ppItf);
}

// 方便的方法
bool IsOK(void) const { return m_dwCookie != 0; }
DWORD GetCookie(void) const { return m_dwCookie; }
};

#define GIP(Itf) GlobalInterfacePointer<Itf, &IID_##Itf>

```

有了这个类定义和宏之后，SafeRect 类就可以用 GlobalInterfacePointer 来代替直接的 DWORD 了：

```

class SafeRect : public IRect {
    LONG m_cRef;                      // COM 引用计数
    IUnknown *m_pUnkFTM;               // 缓存 FTM 懒聚合
    GIP(IPoint) m_gipTopLeft;         // 顶/左 GIT cookie
    GIP(IPoint) m_gipBottomRight;     // 底/右 GIT cookie
    :
    :
}

```

为了初始化 GlobalInterfacePointer 成员，构造函数（它在对象的套间内部执行）只要调用每个 GlobalInterfacePointer 成员的 Globalize 方法，就可以注册对应的、需要被管理的指针，代码如下：

```

SafeRect::SafeRect(void) : m_cRef(0), m_pUnkFTM(0) {
    IPoint *pPoint = 0;
// 实例化 Point 类
    HRESULT hr = CoCreateInstance(CLSID_Point, 0,
                                  CLSCTX_INPROC, IID_IPoint, (void**)&pPoint);
    assert(SUCCEEDED(hr));
// 在 GIT 中注册接口指针
    hr = m_gipTopLeft.Globalize(pPoint);
    assert(SUCCEEDED(hr));
    pPoint->Release(); // 引用存入 GIT
}

```

```

// 实例化 Point 类
hr = CoCreateInstance(CLSID_Point, 0, CLSCTX_INPROC,
                      IID_IPoint, (void**)&pPoint);
assert(SUCCEEDED(hr));
// 在 GIT 中注册接口指针
hr = m_gipBottomRight.Globalize(pPoint);
assert(SUCCEEDED(hr));
pPoint->Release(); // 引用存入 GIT
}

```

对于需要访问这些已经被全局化的指针（即已经被注册到 GIT 中的指针）的方法，它可以通过 GlobalInterfacePointer 的 Localize 方法引入一份本地拷贝：

```

STDMETHODIMP SafeRect::get_Top(long *pVal) {
    IPoint *pPoint = 0; // 局部引入指针
    HRESULT hr = m_gipTopLeft.Localize(&pPoint);
    if (SUCCEEDED(hr)) {
        long x;
        hr = pPoint->get_Coords(&x, pVal);
        pPoint->Release();
    }
    return hr;
}

```

注意，由于用到了 FTM，所以这些未经处理的接口指针不能被缓存起来，它必须在每次方法调用时被引入进来，以防止“在错误的套间中访问这些指针”的情况。

上面的代码片段可以被进一步自动化。因为 GlobalInterfacePointer 类的大多数方法调用都是“在一个方法调用内部产生一个本地临时指针”的操作，所以下面的类可以使“临时指针的引入操作”和“后续的释放操作”自动化，有点类似于智能指针，代码如下：

```

template <class Itf, const IID* piid>
class LocalInterfacePointer {
    Itf *m_pItf; // 临时引入指针
    // 防止误用
    LocalInterfacePointer(const LocalInterfacePointer&);
    operator = (const LocalInterfacePointer&);

public:
    LocalInterfacePointer(const GlobalInterfacePointer<Itf,
                          piid>& rhs, HRESULT& hr) {
        hr = rhs.Localize(&m_pItf);
    }

    LocalInterfacePointer(DWORD dwCookie, HRESULT& hr) {
        assert(g_pGIT != 0);
        hr = g_pGIT->GetInterfaceFromGlobal(dwCookie, *piid,
                                              (void**)&m_pItf);
    }
}

```

```

~LocalInterfacePointer(void)
{ if (m_pItf) m_pItf->Release(); }
class SafeItf : public Itf {
    STDMETHOD_(ULONG, AddRef)(void) = 0; // 隐藏
    STDMETHOD_(ULONG, Release)(void) = 0; // 隐藏
};
SafeItf *GetInterface(void) const
{ return (SafeItf*)m_pItf; }
SafeItf *operator -(void) const
{ assert(m_pItf != 0); return GetInterface(); }
#define LIP(Itf) LocalInterfacePointer<Itf, &IID_##Itf>

```

有了这第二个 C++ 类之后，引入指针的管理工作就显得更加简单了：

```

STDMETHODIMP SafeRect::get_Area(long *pn) {
    long top, left, bottom, right;
    HRESULT hr, hr2;
// 引入指针
    LIP(IPoint) lipTopLeft(m_gipTopLeft, hr);
    LIP(IPoint) lipBottomRight(m_gipBottomRight, hr2);
    assert(SUCCEEDED(hr) && SUCCEEDED(hr2));
// 使用临时局部指针
    hr = lipTopLeft->GetCoords(&left, &top);
    hr2 = lipBottomRight->GetCoords(&right, &bottom);
    assert(SUCCEEDED(hr) && SUCCEEDED(hr2));
    *pn = (right - left) * (bottom - top);
    return S_OK;
// LocalInterfacePointer 自动释放临时指针
}

```

GIP 和 LIP 宏使得 GIT 和 FTM 的结合使用不至于太麻烦。在 GIT 可供使用之前，要想在一个类中同时使用 FTM 和接口指针成员，这远比本节所显示的代码要困难得多。

## 5.11 我们走到哪儿了？

本章把套间概念描述成为一组“共享并发性和重入限制”的对象的逻辑组合。进程具有一个或者多个套间。任何一个线程在某一时刻只能在一个套间中执行。每个 COM 对象只能属于一个套间，为了允许不同套间之间能够进行通信，COM 支持跨套间边界的对象引用列集机制。代理是其他套间中的对象在当前套间中的一个本地代表。标准代理利用 ORPC 把方法请求传输到远程对象上。自定义代理则可以根据它们自己的意愿来提供正确的语义功能。套间是贯彻整个 COM 远程结构的基本抽象。

# 第 6 章

## 应 用

```
int process_id =fork();
if (process_id ==0)
exec("../bin/serverd");
Anonymous, 1981
```

上一章展示了 COM 套间的基础，并且通过较多的细节演示了 COM 的远程结构。上一章还讨论了在多线程环境中管理 COM 对象引用的规则，以及用于实现“线程可知（thread-aware）的 COM 类和对象”的众多技术。这一章将讨论与“COM 中进程和应用管理”有关的内容，焦点集中在套间与错误隔离（fault isolation）、信任（trust）和安全环境（security context）的关系上。

### 6.1 进程内激活的缺陷

到现在为止，COM 服务器一直被认为是进程内的代码单元，它被装载到激活者的进程内部，以便创建对象和执行其方法。对于许多类型的对象来说，这是一种非常合理的配置策略。然而，这种策略也不是没有缺点的。在客户的进程中运行一个对象，其中一个缺点就是缺乏错误隔离性。如果对象引发了一个访问违例错误，或者其他的运行时

严重错误，那么客户的进程将与对象一起被终止。同样地，如果客户程序由于某些原因发生了错误，那么所有在它的地址空间中创建的对象都将立即被销毁（毫无警告）。对于要正常退出的客户程序，这个问题也同样存在，比如最终用户关闭一个应用的时候。当客户进程退出的时候，在客户地址空间中创建的任何一个对象都将被销毁，即使这时进程之外的外部客户仍然拥有有效的引入引用（imported reference）。很显然，一旦对象被激活在客户的进程中，那么当客户进程退出的时候，对象的生命周期也就提前终止了。

对象在客户进程中运行的另一个潜在缺陷是安全环境共享问题。当客户在进程内部激活一个对象的时候，对象的方法在执行时用的是客户的安全凭证（security credential）。这意味着特权用户所创建的对象可以做出非常危险的行为。同时也意味着，具有相对较低权限的用户，他们所创建的对象可能没有足够的权限，无法访问“对于对象的某些操作至关重要”的资源。不幸的是，我们没有直接的办法可以让一个进程中对象拥有它自己的安全环境。

进程内激活的另一个缺陷是，它无法提供分布式计算的功能。如果一个对象必须要在客户的地址空间中激活，那么按照定义，它与客户共享 CPU 和其他的本地资源。进程内激活也使“多个客户进程共享同一个对象”非常困难。虽然套间的概念确实允许对象引用可从任何一个进程中引出去（也包括传统意义上被认为是客户的进程），但是激活语义对于“共享同一个进程内实例”来说还是难以想像的。

为了解决这些问题，COM 允许它的类可在其他的进程中被激活。当一个类（或者一组类）被其他的进程激活的时候，它可以有自己单独的安全环境。这意味着类的实现者可以控制哪些用户有权与它的对象进行通信。类的实现者也可以控制进程应该使用什么样的安全凭证。根据一个类被包装的情况，类的实现者还可以控制什么时候（如果确实要实施控制的话）它的宿主进程将会终止。最后，在不同的进程中激活一个类也提供了一定程度上的错误隔离，它可使客户和对象能够避免相互之间严重错误的影响。

## 6.2 激活和 SCM

COM 服务控制管理器（SCM，Service Control Manager）可以把 CLSID 与注册表中的服务器进程联系起来。这样 SCM 可以根据客户的激活请求，启动一个服务器进程。假定一个类的代码被包装成一个进程映像（一个 EXE 文件），而不是 DLL，那么我们所需要做的就是用 LocalServer32 注册表键代替 InprocServer32 键，如下面的例子所示：

```
[HKCR\CLSID\{27EE6A26-DF65-11d0-8C5F-0080C73925BA}]
@="Gorilla"

[HKCR\CLSID\{27EE6A26-DF65-11d0-8C5F-0080C73925BA}\LocalServer32]
@="C:\ServerOfTheApes.exe"
```

这个进程外服务器（out-of-process server）最好在安装的时候把这些键安装到注册表中。与进程内服务器不同的是，进程外服务器并不会引出 DllRegisterServer 和 DllUnregisterServer 函数。相反，一个进程外服务器必须检查命令行上是否有命令行开关/RegServer 和/UnregServer。<sup>1</sup> 给定了以上的注册表项之后，当 SCM 第一次接到针对 Gorilla 类的激活请求时，它会用 ServerOfTheApes.exe 文件来启动一个新的服务器进程。然后服务器进程有责任通知 SCM：新的进程实际可以提供哪些类的服务。

正如第 3 章中所讨论的，进程与 SCM 进行通信，以便把指针引用绑定到类对象，或者类实例、永久实例。COM 提供了三个激活函数来做到这一点：CoGetClassObject、CoCreateInstanceEx 和 CoGetInstanceFromFile，以及一个高层次的名字对象，它可以把绑定策略的实施细节隐藏起来。不论是哪一种激活策略，类对象都被用来使对象进入到活动状态（把对象带入内存）。正如第 3 章所讨论的，当 COM 在进程内部激活一个对象的时候，它要装载类的 DLL，然后使用统一的入口函数 DllGetClassObject 来获得适当的类对象。第 3 章还未讨论的就是如何跨越进程边界激活一个对象。

一个进程只要显式地把自己注册到 SCM 中，它就变成某个类的服务器进程了。在向 SCM 进行注册之后，针对这个类的所有进程外激活请求都将被分发到该服务器进程中。<sup>2</sup> 服务器进程利用 CoRegisterClassObject API 函数，把自己注册到 SCM 中，此函数的原型如下：

```
HRESULT CoRegisterClassObject(
    [in] REFCLSID rclsid,           // 属于哪个类?
    [in] IUnknown *pUnkClassObject, // 类对象指针
    [in] DWORD dwClsCtx,           // 局部特征
    [in] DWORD dwRegCls,           // 激活标志
    [out] DWORD *pdwReg);          // 关联 ID
```

在调用 CoRegisterClassObject 的时候，COM 库拥有一个指向类对象的引用（即第二个参数所提供的接口指针），并且在内部维护的表中，它把这个类对象与它的 CLSID 关联起来。根据这个调用中所用到的激活标志，COM 库也会通知本地 SCM：调用者的进程现在已经成为指定类的服务器进程了。CoRegisterClassObject 返回一个 DWORD 值，它代表了 CLSID 与类对象之间的关联。这个 DWORD 值也被用来终止此关联（也就是通知 SCM，调用者不再是这个 CLSID 的服务器进程了），服务器进程只需用此 DWORD 值调用 CoRevokeClassObject API 函数即可，函数的原型如下：

```
HRESULT CoRevokeClassObject(
    [in] DWORD dwReg); // 关联 ID
```

<sup>1</sup> 一个设计良好的服务器也应该检查-RegServer 和-UnregServer。而且这四个开关应该是大小写无关的。

<sup>2</sup> 根据一个类在本地注册表中的不同配置情况，服务器进程的注册有可能适用于所有的客户进程，也可能仅适用于具有相同安全凭证（或者在同一个窗口站中执行）的客户进程。

`CoRegisterClassObject` 的许多细微之处都围绕着两个 `DWORD` (即第三、第四个参数) 来展开, 它们可让调用者控制类对象如何被使用以及何时可被使用。

传递给 `CoRegisterClassObject` 的第四个参数是激活标志, 它使调用者可以决定它的类对象什么时候可被利用, 以及多长时间。COM 为这个参数提供了下面的常量:

```
typedef enum tagREGCLS {
    REGCLS_SINGLEUSE           = 0, // 类对象服务于一次
    REGCLS_MULTIPLEUSE         = 1, // 类对象服务于多次
    REGCLS_MULTI_SEPARATE      = 2, // 类对象服务于多次
    REGCLS_SUSPENDED           = 4, // 不通知 SCM (标志)
    REGCLS_SURROGATE           = 8 // 用于 DLL 代理
} REGCLS;
```

`REGCLS_SURROGATE` 值被用于 DLL 代理 (*surrogate*) 实现中, 本章后面将会讨论到这种实现。最主要的两个值是 `REGCLS_SINGLEUSE` 和 `REGCLS_MULTIPLEUSE`。前者告诉 COM 库, 类对象只能服务于一次激活请求。一旦第一次激活请求发生了, COM 就把这个已经被注册的类对象从公开的视野中删除掉。如果再有第二次激活请求的话, COM 必须使用其他已被注册的类对象。如果不存在其他具有相同 `CLSID` 的类对象, 那么 COM 将会创建另一个服务器进程来满足这个激活请求。

与此相反, `REGCLS_MULTIPLEUSE` 标志指示类对象可以被重复利用多次, 直到 `CoRevokeClassObject` 调用把它从 COM 库的类表中删除为止。`REGCLS_MULTI_SEPARATE` 标志针对可能发生在调用者进程中的后续进程内激活请求。如果调用者用 `REGCLS_MULTIPLEUSE` 标志注册了一个类对象, 那么 COM 假定任何来自于调用者进程的进程内激活请求不应该装载另外的进程内服务器, 而应该使用已经注册过的类对象。这意味着即使调用者只是用 `CLCTX_LOCAL_SERVER` 标志注册类对象, 被注册的类对象也可以用于来自同一进程的进程内激活请求。如果调用者不能接受这种行为的话, 那么它应该使用 `REGCLS_MULTI_SEPARATE` 标志来注册类对象, 这就告诉了 COM, 只有当该类被注册时使用了 `CLCTX_INPROC_SERVER` 标志, 被注册的类对象才可以用于进程内激活请求。这意味着下面的 `CoRegisterClassObject` 调用:

```
hr = CoRegisterClassObject(CLSID_Me, &g_coMe,
                           CLCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE, &dw);
```

等价于下面的 `CoRegisterClassObject` 调用:

```
hr = CoRegisterClassObject(CLSID_Me, &g_coMe,
                           CLCTX_LOCAL_SERVER|CLCTX_INPROC,
                           REGCLS_MULTI_SEPARATE, &dw);
```

无论在哪种情况下, 如果调用者进程发出了下面的调用:

```
hr = CoGetClassObject(CLSID_Me, CLSCTX_INPROC, 0,
                      IID_IUnknown, (void**)&pUnkCO);
```

那么，不会有新的 DLL 被装入进来；相反，COM 将使用被 `CoRegisterClassObject` 注册的类对象来满足调用者的请求。然而，如果服务器进程以下面的形式来调用 `CoRegisterClassObject`：

```
hr = CoRegisterClassObject(CLSID_Me, &g_coMe,
                           CLSCTX_LOCALSERVER,
                           REGCLS_MULTI_SEPARATE, &dw);
```

这样，所有针对 `CLSID_Me` 的进程内激活请求，如果它来自服务器进程内部，那么 COM 将强行装入另一个 DLL。

`CoRegisterClassObject` 也把被注册的类对象与调用者的套间关联起来。这意味着所有的入方法（*incoming method*）请求都将在调用者的套间内部执行。如果类对象引出 `IClassFactory` 接口，那么这就意味着 `CreateInstance` 方法将执行在调用者的套间中。`CreateInstance` 方法的结果将从类对象的套间中被列集出去，这也意味着对象的实例与类对象属于同一个套间。<sup>3</sup>

服务器进程也可以为多个不同的类注册相应的类对象。如果类对象被注册时运行在进程的 MTA 中，这就意味着一旦第一个 `CoRegisterClassObject` 调用完成之后，进来的入激活请求（*incoming activation request*）立即就可以得到服务了。在许多基于 MTA 的服务器进程中，这可能会产生问题，因为进程可能还会有进一步的初始化工作需要完成。为了避免这个问题，Windows NT 4.0 的 COM 版本引入了 `REGCLS_SUSPENDED` 标志。当服务器进程在 `CoRegisterClassObject` 调用中加入了这个标志之后，COM 库并不会通知 SCM 说这个类现在已经可以使用了。这样可以阻止入激活请求马上到达服务器进程内部。COM 库也会把 `CLSID` 和类对象关联起来；然而，COM 库把内部类表中对应的表项标记为“挂起”状态。为了改变这种“挂起”状态，COM 提供了一个 API 函数 `CoResumeClassObjects`：

```
HRESULT CoResumeClassObjects(void);
```

`CoResumeClassObjects` 完成两件事情。第一，它把所有被挂起的类对象标记为“可用”状态。第二，它给 SCM 发送一个通知，告诉 SCM 服务器进程中原先被挂起的类对象现在都可以使用了。这个通知是个原子（atomic）操作，也就是说，它针对被调用者注册的所有类，一次性更新机器范围内的 SCM 类表。

有了上面讲述的三个 API 函数之后，要创建一个服务器进程，并让它引出一个或者多个类就非常简单了。下面是一个简单程序，它从服务器的 MTA 中引出三个类，代码

---

<sup>3</sup> 从技术上讲，`CreateInstance` 方法也有可能使用标准的多线程程序设计技术，从而强迫对象被创建在其他的套间中。然而，实际上 `CreateInstance` 的实现往往只是简单地在当前套间中创建一个新的对象。

如下：

```

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int){
    // 为每个类定义一个类对象
    static GorillaClass s_gorillaClass;
    static OrangutanClass s_orangutanClass;
    static ChimpClass s_chimpClass;
    DWORD rgdwReg[3];
    const DWORD dwRegCls= REGCLS_MULTIPLEUSEJREGCLS_SUSPENDED;
    const DWORD dwClsCtx = CLSCTX_LOCAL_SERVER;
    // 进入 MTA
    HRESULT hr = CoInitializeEx(0, COINIT_MULTITHREADED);
    assert(SUCCEEDED(hr));

    // 在 COM 库的类表中注册类对象
    hr = CoRegisterClassObject(CLSID_Gorilla, &s_gorillaClass,
                                dwClsCtx, dwRegCls, rgdwReg);
    assert(SUCCEEDED(hr));
    hr = CoRegisterClassObject(CLSID_Orangutan,
                               &s_orangutanClass, dwClsCtx, dwRegCls, rgdwReg + 1);
    assert(SUCCEEDED(hr));
    hr = CoRegisterClassObject(CLSID_Chimp, &s_chimpClass,
                               dwClsCtx, dwRegCls, rgdwReg + assert(SUCCEEDED(hr)));
    // 通知 SCM
    hr = CoResumeClassObjects();
    assert(SUCCEEDED(hr));
    // 保持进程活动直至事件出现？
    extern HANDLE g_hEventShutdown;
    WaitForSingleObject(g_hEventShutdown, INFINITE);
    // 从 COM 库的类表中删除项
    for (int n = 0; n < 3; n++)
        CoRevokeClassObject(rgdwReg[n]);
    // 离开 MTA
    CoUninitialize();
    return 0;
}

```

以上代码片断假定在进程的某个地方，程序已经对 Win32 的事件对象（Event object）做了初始化，如下：

```
HANDLE g_hEventShutdown = CreateEvent(0, TRUE, FALSE, 0);
```

有了这样的事件对象之后，我们只须调用 SetEvent API 函数，就可以让服务器正常有序地终止，代码非常简单：

```
SetEvent(g_hEventShutdown);
```

上面的 SetEvent 调用会触发主线程的终止序列。如果这个服务器是以 STA 为基础

的话，那么主线程有必要运行一个 Windows 消息循环，而不是等待 Win32 同步事件。这样才能使入 ORPC 请求可以进入到主线程的套间中。

### 6.3 再谈服务器生命周期

上节显示的例子并没有考虑服务器进程如何终止以及何时终止的问题。一般来讲，服务器进程控制它自己的生命周期，可以在任何它愿意的时候终止自己。尽管服务器进程无限制地运行下去也是合法的，但是大多数服务器进程选择在“没有未完结的引用指向内部的对象或者类对象”的时候终止自己。这很类似于大多数进程内服务器在 DllCanUnloadNow 函数中所采用的策略。回顾前面第 3 章，一个服务器通常要实现两个函数，当外部客户获取或者释放接口指针的时候，这两个函数就会被调用到：

```
// 维持装载的原因
LONG g_cLocks = 0;
// 来自 AddRef + IClassFactory::LockServer(TRUE) 的调用
void LockModule(void)
{ InterlockedIncrement(&g_cLocks); }
// 来自 Release + IClassFactory::LockServer(FALSE) 的调用
void UnlockModule(void)
{ InterlockedDecrement(&g_cLocks); }
```

这使得 DllCanUnloadNow 的实现非常简单：

```
STDAPI DllCanUnloadNow() {return g_cLocks ? S_FALSE : S_OK;}
```

当客户调用 CoFreeUnusedLibraries 函数回收其地址空间的时候，DllCanUnloadNow 函数就会被调用到。

基于 EXE 的服务器的终止处理工作有一些不同。首先，激发终止过程是服务器进程自己的任务。不同于进程内服务器的情形，当进程外服务器愿意终止的时候，外界并不存在垃圾回收器会来询问它。相反，服务器进程必须在适当的时候显式地终止自己。如果一个 Win32 事件对象可用来终止服务器进程的话，那么这个进程必须调用 SetEvent API 函数：

```
void UnlockModule(void) {
    if (InterlockedDecrement(&g_cLocks) == 0) {
        extern HANDLE g_hEventShutdown;
        SetEvent(g_hEventShutdown);
    }
}
```

如果服务器的主线程正服务于一个 Windows MSG 队列，那么它必须要用某个 API 函数来终止循环过程。最为直接的技术是使用 PostThreadMessage，寄给（post）主线程一个 WM\_QUIT 消息：

```
void UnlockModule(void) {
    if (InterlockedDecrement(&g_cLocks) == 0) {
        extern DWORD g_dwMainThreadID; // 主线程设置
        PostThreadMessage(g_dwMainThreadID, WM_QUIT, 0, 0);
    }
}
```

如果一个以 STA 为基础的服务器进程知道它永远也不会创建其他的线程，那么它可以使用更为简单的 PostQuitMessage API 函数：

```
void UnlockModule(void) {
    if (InterlockedDecrement(&g_cLocks) == 0)
        PostQuitMessage(0);
}
```

只有当服务器进程的主线程调用这个函数时，这项技术才能起作用。

关于进程内服务器与进程外服务器生命周期管理，另一个区别关系到“维持服务器使它处于被装载或者正在运行状态的强制力量”。在进程内服务器中，存在两种这样的力量：未完结的对象引用和未完结的 IClassFactory::LockServer (TRUE) 调用。进程外服务器也要检查第一种力量。

无疑，当外部客户拥有指向服务器中类对象的未完结引用的时候，服务器应该留在内存中。对于进程内服务器，这可以用下面的代码来实现：

```
STDMETHODIMP_(ULONG) MyClassObject::AddRef(void) {
    LockModule();           // 注意未完结引用
    return 2;               // 不基于堆的对象
}
STDMETHODIMP_(ULONG) MyClassObject::Release(void) {
    UnlockModule();         // 注意已销毁引用
    return 1;               // 不基于堆的对象
}
```

这是一种强制行为，因为如果 DLL 被卸载的时候仍然存在指向类对象的未完结引用的话，那么后续的调用，即使是对 Release 方法的调用也足以使客户进程崩溃。

不幸的是，前面的 AddRef 和 Release 实现对于进程外服务器是不合适的。请回忆一下，在进入一个 COM 套间之后，进程外服务器通常做的第一件事情就是调用 CoRegisterClassObject，向 COM 库注册它的类对象。然而，当类表拥有一个类对象之后，对于这个类对象而言，至少会有一个未完结的 COM 引用。这意味着在注册了一个类对象之后，模块范围内的锁计数器就不再是 0 了。在服务器调用 CoRevokeClassObject 之

前，这些“自我强加的”引用将不会被释放。不幸的是，在模块范围内的锁计数器到达 0 之前，服务器进程通常不会调用 CoRevokeClassObject，这就意味着服务器进程永远也不会终止了。

为了打破类表（class table）和服务器生命周期之间的死循环，大多数进程外服务器的类对象实现只是简单地忽略掉对于 AddRef 和 Release 的未完结调用：

```
STDMETHODIMP_(ULONG) MyClassObject::AddRef(void) {
    // 忽略未完结引用
    return 2;           // 不基于堆的对象
}
STDMETHODIMP_(ULONG) MyClassObject::Release(void) {
    // 忽略已销毁引用
    return 1;           // 不基于堆的对象
}
```

这就意味着，服务器进程在注册了它的类对象之后，其模块范围内的锁计数器仍然为 0。

粗看起来，这种实现隐含着“服务器进程有可能在指向类对象的未完结引用仍然存在的情况下终止”。实际上这种行为要取决于类对象是如何实现的。请回忆以前提到过的，只要还存在指向类对象的外部引用，服务器就应该留在内存中继续运行。上面修改之后的 AddRef 和 Release 只是影响到内部的引用（为 COM 库的类表所拥有），因此会被忽略掉。当外部客户请求一个指向服务器进程中某个类对象的引用的时候，SCM 进入到类对象的套间中，并获取一个指向类对象的引用。这时候会发出 CoMarshalInterface 调用，以便把对象引用列集给客户。如果类对象实现了 IExternalConnection 接口，它可以知道什么时候外部引用尚未完结，并且用这一信息来控制服务器的生命周期。假定一个类对象实现了 IExternalConnection 接口，那么下面的代码就可以获得这种预期的效果：

```
STDMETHODIMP_(DWORD) MyClassObject::AddConnection(
    DWORD extconn, DWORD) {
    DWORD res = 0;
    if (extconn&EXTCONN_STRONG) {
        LockModule(); // 注意外部对象
        res = InterlockedIncrement(&m_cExtRef);
    }
    return res;
}
STDMETHODIMP_(DWORD) MyClassObject::ReleaseConnection(
    DWORD extconn, DWORD, BOOL bLastReleaseKillsStub) {
    DWORD res = 0;
    if (extconn&EXTCONN_STRONG) {
        UnlockModule(); // 注意外部对象
        res = InterlockedDecrement(&m_cExtRef);
        if (res == 0 & bLastReleaseKillsStub)
            CoDisconnectObject((IExternalConnection*)this, 0);
    }
}
```

```

    }
    return res;
}

```

注意，只要存在指向类对象的未完结外部引用，模块范围内的锁计数器就不会是0，但是COM库所拥有的内部引用则被忽略不计。

尽管自从COM早期开始，“在类对象上使用IExternalConnection接口”这项技术一直工作得很好，但是很少有实现者会真正使用这项技术。相反，大多数服务器往往会忽略掉针对类对象的未完结外部引用，从而有可能提前终止服务器进程。并且IClassFactory接口中LockServer方法的出现又加剧了这种状况，这个方法使开发人员真的认为“客户可以确保服务器留在内存中继续保持运行状态”。虽然大多数服务器的实现者将在LockServer方法中锁住进程模块，但是实际上客户并没有可靠的办法来调用这个方法。考虑下面的客户代码：

```

IClassFactory *pcf = 0;
HRESULT hr = CoGetClassObject(CLSID_Your, CLSCTX_LOCAL_SERVER,
                               0, IID_IClassFactory, (void**)&pcf);
if (SUCCEEDED(hr))
    hr = pcf->LockServer(TRUE); // 保持服务器运行吗?

```

在最初的COM版本中，这段代码有严重的竞争条件（race condition）。注意，在CoGetClassObject和IClassFactory::LockServer两个调用之间有一个时间间隔。在这段时间隔中，有可能存在另外一个客户销毁掉这个类的最后一个实例。因为指向类对象的未完结引用被服务器实现者忽略掉了，所以服务器进程将在客户调用LockServer之前就被终止了。理论上，这个问题可以用以下的代码来解决：

```

IClassFactory *pcf = 0;
HRESULT hr = S_OK;
do {
    if (pcf) pcf->Release();
    hr = CoGetClassObject(CLSID_Your, CLSCTX_LOCAL_SERVER,
                          0, IID_IClassFactory, (void**)&pcf);
    if (FAILED(hr))
        break;
    hr = pcf->LockServer(TRUE); // 保持服务器运行吗?
} while (FAILED(hr));

```

注意，这段代码片断不断地尝试“获取一个类对象并锁住它”，一直到LockServer调用成功为止。如果服务器在CoGetClassObject和LockServer调用之间提前结束的话，那么LockServer调用将会返回一个错误，该错误表明是一个断开连接的代理，并强制代码继续重复下去。在Windows NT 3.51或者更早的版本下，这段怪异的代码是唯一“能够可靠地获取指向类对象的引用”的办法。

真的要感谢“许多服务器的实现并没有使用 `IExternalConnection` 来管理服务器的生命周期”，所以 Windows NT 4.0 发放的 COM 版本引入了下面的增强功能来补偿这些不完善的服务器实现。当 SCM 列集一个指向类对象的引用，以作为对 `CoGetClassObject` 调用的响应时，它将调用类对象的 `IClassFactory::LockServer` 方法。由于大多数服务器都在类对象上实现 `IClassFactory`，所以 COM 库的这项增强功能修补了绝大多数缺陷。然而，如果一个类对象并没有引出 `IClassFactory` 接口，或者服务器必须要在 Windows NT 4.0 之前的 COM 环境中执行，则必须要使用 `IExternalConnection` 技术。

另一个与服务器生命周期有关的话题也有必要在这里进行讨论。注意，当一个服务器决定终止的时候，它发出信号，通知服务器应用的主线程应该启动它的终止序列，然后退出进程。服务器的终止序列包括“调用 `CoRevokeClassObject` 以注销类对象”。然而，如果服务器用到了前面显示的 `UnlockModule` 实现代码的话，那样就会存在严重的“竞争条件”。在“服务器调用 `SetEvent` 或者 `PostThreadMessage` 函数通知主线程”和“服务器调用 `CoRevokeClassObject` 注销它的类对象”两个时间点之间，有可能会有其他的激活请求到达服务器进程。如果在这段时间之内，新的对象被创建了的话，那就没有办法来告诉主线程“现在不应该终止”和“进程现在有了新的对象需要服务”。为了消除这个竞争条件，COM 提供了下面两个 API 函数：

```
ULONG CoAddRefServerProcess(void);
ULONG CoReleaseServerProcess(void);
```

这两个函数为调用者管理一个模块范围内的锁计数器。这些函数临时阻塞所有对 COM 库的访问，以确保在锁计数器被调整的过程中，不会有新的激活请求被处理。而且，如果 `CoReleaseServerProcess` 检测到它正在删除进程中最后一个锁计数时，它会在内部把进程的所有类对象标记为“挂起”状态，并通知 SCM“进程不再接受相应的 CLSID 的服务”。

下面的函数可以正确地实现进程外服务器的生命周期管理控制：

```
void LockModule(void) {
    CoAddRefServerProcess(); // COM 维持锁计数器
}
void UnlockModule(void) {
    if (CoReleaseServerProcess() == 0)
        SetEvent(g_hEventShutdown);
}
```

注意，使进程以一种很有序的方式终止，这仍然是调用者的责任。然而，一旦服务器已经决定要终止了，那么服务器进程就不会再为新的激活请求提供服务。

即使在使用了 `CoAddRefServerProcess/CoReleaseServerProcess` 之后，仍然有可能产生竞争条件。当 `CoReleaseServerProcess` 正在执行的时候，RPC 层仍然有可能接收到来自 SCM 的入激活请求。如果来自 SCM 的调用是在 `CoReleaseServerProcess` 释放了它对

COM 库的锁之后才被分发的，那么这个激活请求将注意到类对象已经被标记为“挂起”状态，于是一个错误码将被返回给 SCM (CO\_E\_SERVER\_STOPPING)。当 SCM 检测到这个错误码之后，它简单地启动一个新的服务器进程实例，一旦服务器进程完成自身注册工作之后，SCM 就会重试这个请求。尽管 COM 库已经用到了安全保护措施，但是入激活请求仍然有可能与最后的 CoReleaseServerProcess 调用同时执行。为了解决这个问题，服务器一旦检测到在发出终止请求之后又有一个请求被发出的话，它可以从 IClassFactory::CreateInstance 和 IPersistFile::Load 方法中显式地返回 CO\_E\_SERVER\_STOPPING。下面的代码演示了这项技术：

```
STDMETHODIMP MyClassObject::CreateInstance(IUnknown *puo,
                                         REFIID riid, void **ppv) {
    LockModule(); // 确保不在调用时关闭
    HRESULT hr; *ppv = 0;
    // 关闭初始化吗？
    DWORD dw : WaitForSingleObject(g_hEventShutdown, 0);
    if (dw == WAIT_OBJECT_0) hr = CO_E_SERVER_STOPPING;
    else {
        // 正常的 CreateInstance 实现
    }
    UnlockModule();
    return hr;
}
```

在本书写作的时候，还没有商业版本的 COM 类库实现了这项技术。

## 6.4 应用 ID

Windows NT 4.0 的 COM 版本引入了“COM 应用 (COM application)”的概念。COM 应用也用 GUID 来标识（在此情况下，这个 GUID 被称为 AppID），它代表了一个服务器进程（可以处理一个或者多个类）。每个 CLSID 只能与一个应用 ID 关联起来，这种关联关系通过本地注册表中的 AppID 名字值体现出来，例如：

```
[HKCR\CLSID\{27EE6A4E-DF65-11d0-8C5F-0080C73925BA}]
@="Gorilla"
AppID="{27EE6A40-DF65-11d0-8C5F-0080C73925BA}"
```

属于同一个 COM 应用的所有类具有同样的 AppID，并且它们共享同样的远程激活设置和安全性设置。这些设置被保存在本地注册表中 HKEY\_CLASSES\_ROOT\AppID 键的下面。

HKEY\_CLASSES\_ROOT\AppID

与 CLSID 一样，在 Windows NT 5.0（即 Windows 2000）或者以后的版本中，AppID 也可以在每个用户的基础上被注册。因为在 Windows NT 4.0 之前的版本中实现的服务器并不会显式地注册它们的 AppID，所以 COM 配置工具（例如 DCOMCNFG.EXE、OLEVIEW.EXE）将自动地为这些遗留下来的服务器程序创建一个新的 AppID。为了替这些遗留下来的服务器合成 AppID，这些工具自动地把 AppID 名字值加入到被特定的本地服务器引出的所有 CLSID 下。在加入这些名字值的时候，DCOMCNFG 或者 OLEVIEW 简单地使用它所碰到的第一个 CLSID（针对某个特定的本地服务器）作为 AppID。在 Windows NT 4.0 之后开发的应用可以（也应该）使用其他的 GUID 作为它们的 AppID。

大多数 AppID 设置可以使用 DCOMCNFG.EXE 来完成，它也是 Windows NT 4.0 或者更高版本的标准组件。DCOMCNFG.EXE 为管理员提供了非常易于使用的友好界面，用来控制远程设置和安全性设置。另一个功能更为强大的工具 OLEVIEW.EXE 包含了 DCOMCNFG.EXE 的绝大多数功能，同时还提供了一个以 COM 为中心的注册表视图。这两个工具的用法非常直观，对于 COM 开发工作来说都是非常基本的工具。

最简单的 AppID 设置是 RemoteServerName。这个名字值表明，如果一个远程激活请求没有通过 COSERVERINFO 显式地指定远程主机名字，那么 COM 应该使用哪台机器。考虑下面的注册表设置：

```
[HKCR\appid\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}]
@="Ape Server"
RemoteServerName= "www.apes.com"

[HKCR\CLSID\{27EE6A4E-DF65-11d0-8C5F-0080C73925BA}]
@="Gorilla"
AppID=" {27EE6A4D-DF65-11d0-8C5F-0080C73925BA} "

[HKCR\CLSID\{27EE6A4F-DF65-11d0-8C5F-0080C73925BA}]
@="Chimp"
AppID=" {27EE6A4D-DF65-11d0-8C5F-0080C73925BA} "
```

如果客户发出如下所示的激活请求：

```
IApeClass *pac = 0;
HRESULT hr = CoGetClassObject(CLSID_Chimp,
    CLSCTX_REMOTE_SERVER, 0, IID_IApeClass, (void**)&pac);
```

那么客户方的 SCM 将把这个请求转发给 www.apes.com 上的 SCM；在 www.apes.com 这台机器上，这个请求将被当作本地激活请求一样来对待。注意，如果客户提供了显式的主机名字：

```

IApeClass *pac = 0;
COSERVERINFO csi; ZeroMemory(&csi, sizeof(csi));
csi.pwszName = OLESTR("www.dogs.com");
HRESULT hr = CoGetClassObject(CLSID_Chimp,
    CLSCTX_REMOTE_SERVER, &csi, IID_IApeClass, (void**)&pac);

```

那么 RemoteServerName 设置就会被忽略掉，并且这个请求会被转发给 www.dogs.com。

更为常见的情形是，客户既没有显式地指定主机名字，也没有指定本地优先选择的标志。考虑下面的 CoGetClassObject 调用：

```

IApeClass *pac= 0;
HRESULT hr = CoGetClassObject(CLSID_Chimp,
    CLSCTX_ALL, 0, IID_IApeClass, (void**)&pac);

```

因为没有指定主机名字，所以 SCM 将首先在本地注册表中查找下面的键：

```
[HKCR\CLSID\{27EE6A4F-DF65-11d0-8C5F-0080C73925BA}]
```

如果找不到这个键的话，COM 就会与 Windows 2000 的类存储（class store）协商，看是否可以找到这个键的信息。如果这时候找到了该类的注册表键的话，然后 SCM 就会寻找 InprocServer32 子键：

```
[HKCR\CLSID\{27EE6A4F-DF65-11d0-8C5F-0080C73925BA}\InprocServer32]
@="C:\somefile.dll"
```

如果找到了这个键的话，那么 SCM 只需装载该键所指示的 DLL，就可以激活这个类。如果没有找到这个键的话，那么 SCM 会查找 InprocHandler32 子键：

```
[HKCR\CLSID\{27EE6A4F-DF65-11d0-8C5F-0080C73925BA}\InprocHandler32]
@="C:\somefile.dll"
```

如果这个类有上述控制器键（handler key）的话，那么同样地，SCM 只需装载该键所指示的 DLL，就可以激活这个类。如果这两个进程内子键都没有找到的话，SCM 就假定这个激活请求一定是进程外类型的。这时候，SCM 会检查是否有服务器进程已经注册了针对当前被请求的 CLSID 的类对象。<sup>4</sup> 如果确实有的话，SCM 就会到达服务器进程中，然后从适当的类对象上列集得到一个对象引用，并返回给调用者的套间，之后把对象引用散集出来，再把控制返回给调用者。如果服务器进程使用 REGCLS\_SINGLEUSE 标志来注册这个类对象，那么 SCM 会忘记掉“这个类在服务器进程中是可以使用的”，因此后续的激活请求就不会再使用这个类对象了。

---

<sup>4</sup> 从技术上讲，这个类对象在调用者安全环境中必须合法才可以。

如果服务器进程已经在运行了，那么上面所描述的情形就是正确的。然而，如果 SCM 收到一个进程外激活请求，但是并没有服务器进程为相应的 CLSID 注册了类对象，那么 SCM 会启动一个服务器进程（如果服务器进程没在运行的话）。对于创建类对象的服务器，COM 支持三种进程模型：NT 服务（NT Service）、普通的进程以及代理进程（surrogate process）。NT 服务和普通进程非常类似，可能有人喜欢某一种而不喜欢另一种，确切的理由将在本章后面进一步讨论。代理进程主要被用来容纳传统遗留下来的进程内服务器，使它们拥有独立的服务器进程。这为这些遗留下来的 DLL 或者“必须要被包装到 DLL 中的类”（比如 Java 虚拟机）提供了远程激活和错误隔离的好处。不管使用哪种模型来创建服务器进程，服务器进程可以有 120 秒时间（在 Windows NT Service Pack 2 或者之前的版本中为 30 秒），使用 CoRegisterClassObject 来注册被请求的类对象。如果服务器进程不能及时地注册自己的话，SCM 将认为调用者的激活请求已经失败了。

在创建服务器进程的时候，SCM 首先检查是否被请求的类所对应的 AppID 有 LocalService 名字值，例如：

```
[HKCR\AppID\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA} ]
LocalService="apesvc"
```

如果这个名字值出现的话，那么 SCM 会使用 NT 服务控制管理器（NT Service Control Manager）来启动注册表中指定的 NT 服务（比如 apesvc）。如果 LocalService 名字值没有出现的话，那么 SCM 会在指定的 CLSID 键下查找 LocalServer32 子键，例如：

```
[HKCR\CLSID\{27EE6A4F-DF65-11d0-8C5F-0080C73925BA}\LocalServer32]
@="C:\Xsomefile.exe"
```

如果这个键出现的话，那么 SCM 将会使用 CreateProcess（或者 CreateProcessAsUser）API 函数，以启动服务器进程。如果 LocalService 或者 LocalServer32 注册表项都没有出现的话，SCM 就检查是否有代理进程被分配给这个类的 AppID，例如：

```
[HKCR\AppID\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}]
DllSurrogate=""
```

如果 DllSurrogate 名字值出现了，但是其值为空，那么 SCM 将启动缺省的代理进程（dllhost.exe）。如果 DllSurrogate 名字值出现了，并且指向一个有效的文件名，例如：

```
[HKCR\AppID\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}]
DllSurrogate="C:\somefile.exe"
```

那么 SCM 将会启动该名字值指定的服务器进程。无论哪种情况下，代理进程都会使用 CoRegisterSurrogate API 函数，把自己注册到 COM 库中，成为 COM 代理进程。CoRegisterSurrogate 函数的原型如下：

```
HRESULT CoRegisterSurrogate([in] ISurrogate *psg);
```

这个 API 函数期望代理进程会提供一个 ISurrogate 接口实现，ISurrogate 接口的定义如下：

```
[ uuid(00000022-0000-0000-C000-000000000046), object ]
interface ISurrogate : IUnknown {
// SCM 要求代理装载进程内类对象
// 并用 REGCLS_SUSPENDED 调用 CoRegisterClassObject
    HRESULT LoadDllServer([in] REFLSID rclsid);
// SCM 要求代理关闭
    HRESULT FreeSurrogate();
}
```

ISurrogate 接口为 COM 提供了“使代理进程可以注册类对象以及终止进程”的机制。这种代理机制（surrogate mechanism）之所以存在，主要是为了使遗留下来的进程内服务器也能够支持远程激活。一般来讲，只有对于那些无法改建成进程外服务器的进程内服务器才会使用代理进程。

最后，如果所有这些注册表键或者名字值都没有出现的话，那么 SCM 将在该类对应的 AppID 下面查找 RemoteServerName 表项：

```
[HKCR\AppID\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}]
RemoteServerName="www.apes.com"
```

如果这个值出现了的话，那么激活请求就被转发给指定主机上的 SCM。注意，即使客户在最初的激活请求中仅仅指定了 CLSCTX\_LOCAL\_SERVER 标志，如果没有本地服务器进程被注册，那么这个请求也会被转发出去。

另外一个也能够导致激活请求转向的因素仅仅适用于 CoGetInstanceFromFile 请求（也包括调用文件名字对象的 BindToObject 方法）。缺省情况下，如果用于命名永久对象的文件名指向远程文件系统上的一个文件，则 COM 会使用上面刚刚讲述的算法来决定应该在哪里激活对象。然而，如果类的 AppID 有一个 ActivateAtStorage 名字值，并且它的值为“Y”或者“y”，那么虽然调用者并没有在 COSERVERINFO 结构中显式地提供主机名字，但是 COM 仍然把激活请求转发给该文件所在的机器。这样做可以保证在整个网络上只存在一个实例。

## 6.5 COM 和安全性

早期的 COM 版本并没有考虑安全性。这可以被看作是一种失误，因为许多非远程概念的 NT 语义（比如进程和线程），虽然它们不能被远程控制，但是也需要被保护起

来。Windows NT 4.0 强制性地把安全性加入到 COM 中，因为网络上的任何一台机器都有潜在的可能会访问某个服务器进程。幸运的是，因为 COM 使用 RPC 作为它的传输协议，所以 COM 安全性只要简单地让 RPC 现有的安全性基础设施发挥作用即可。

COM 安全性可以被分成三大类：认证（authentication）、访问控制（access control）和令牌管理（token management）。认证负责确保一条消息是可信的，也就是说，发送者确实是他所声称的人，并且给定的消息确实是他发送的。访问控制解决“允许谁访问服务器的对象，以及允许谁启动服务器进程”。令牌管理负责控制“在启动服务器进程的时候，以及在方法内部执行的时候，该使用哪个安全凭证”。COM 为这三方面安全性提供了某种合理的缺省设置，从而使我们有可能在编写 COM 应用的时候无须考虑安全性。这些缺省设置并没有任何令人惊讶的地方；也就是说，如果一个程序员并没有显式地处理安全性，NT 安全性模型中的任何漏洞都不太可能被引进来。然而，即使是建造一个简单的分布式 COM 应用，它也要求对安全性的各个方面都给予足够的重视。

COM 的大部分安全性只需要在注册表中放上正确的信息进行配置即可。DCOMCNFG.EXE 程序可让管理员调整绝大多数（不是全部）与 COM 安全性有关的设置。对于这大多数（不是全部）设置，应用开发人员也可以在程序中使用显式的 API 函数来替换掉这些注册表设置。一般来说，大多数应用会把 DCOMCNFG.EXE 和显式的 API 函数结合起来。前者使系统管理员很容易调试，后者提供了更好的灵活性，同时也可避免误用或者滥用 DCOMCNFG.EXE。

COM 安全性使用底层的 RPC 设施来提供认证和模仿（impersonation）功能。前面曾经提到过，RPC 使用可装载的传输模块，以便能够把新的网络协议加到系统中。这些传输模块用协议序列（比如“ncadg\_ip\_udp”）来命名，在注册表中这些协议序列被映射到特定的传输 DLL 上。这使得第三方厂商可以安装新的传输协议，而无须修改 COM 库。类似地，RPC 也支持可装载的安全性软件包（简称为安全包），从而允许新的安全协议可以被加入到系统中。这些安全包用整数来命名，在注册表中这些整数被映射到特定的安全包 DLL 上。这些 DLL 必须遵从安全性支持提供器接口（SSPI，Security Support Provider Interface），而 SSPI 是从 Internet 草案标准 GSSAPI 演变而来的。

系统头文件定义了几个常量，用来表示已知的安全包。在本书写作的时候，以下就是当前已知的安全包：

```
enum {
    RPC_C_AUTHN_NONE = 0,           // 无认证包
    RPC_C_AUTHN_DCE_PRIVATE = 1,     // DCE 私有密钥（未使用）
    RPC_C_AUTHN_DCE_PUBLIC = 2,      // DCE 公共密钥（未使用）
    RPC_C_AUTHN_DEC_PUBLIC = 4,      // DEC（未使用）
    RPC_C_AUTHN_WINNT = 10,          // NT Lan Manager
    RPC_C_AUTHN_GSS_KERBEROS,
    RPC_C_AUTHN_MQ = 100,            // MS 报文队列包
    RPC_C_AUTHN_DEFAULT = 0xFFFFFFFFL
};
```

RPC\_C\_AUTHN\_WINNT 表示应该使用 NT LAN Manager (NTLM) 认证协议。RPC\_C\_AUTH\_GSS\_KERBEROS 表示应该使用 Kerberos 认证协议。在 Windows NT 4.0 版本上，除非安装有第三方厂商的 SSP，否则就只有 NTLM 可以使用。Windows NT 5.0 (即 Windows 2000) 将至少会支持 NTLM 和 Kerberos。请参考最新的文档以了解到有关其认证包的可用信息。

每个接口代理可以被独立配置，以便使用不同的安全包。当一个接口代理被配置成用某一个安全协议的时候，对应的 SSP DLL 就会被装载到客户的进程中。为了接受所有的连接请求，服务器进程必须在接收到第一个来自客户的 ORPC 请求之前，就注册装载对应的 SSP DLL。当一个连接被配置成使用某个安全包的时候，对应的 SSP DLL 在 RPC 运行时层联合起来协同工作，而且对于特定连接上的每一个被传送和接收的数据包 (packet)，这个 SSP DLL 都有机会可以看到它们。SSP DLL 可以在每一个包中发附加的与安全有关的信息，也可以修改被列集之后的参数状态，以提供加密功能。DCE RPC (和 COM) 允许 6 个级别的认证保护，从没有保护一直到加密所有的参数状态，文件中的定义如下：

```
enum {
    RPC_C_AUTHN_LEVEL_DEFAULT,           // 使用缺省级
    RPC_C_AUTHN_LEVEL_NONE,              // 无认证
    RPC_C_AUTHN_LEVEL_CONNECT,           // 仅认证凭证
    RPC_C_AUTHN_LEVEL_CALL,              // 保护报文首部
    RPC_C_AUTHN_LEVEL_PKT,               // 保护数据包首部
    RPC_C_AUTHN_LEVEL_PKT_INTEGRITY,     // 保护参数状态
    RPC_C_AUTHN_LEVEL_PKT_PRIVACY,       // 加密参数状态
};
```

每一个后续的认证级别都包含了前一个级别的功能。RPC\_C\_AUTHN\_LEVEL\_NONE 表示没有认证。RPC\_C\_AUTHN\_LEVEL\_CONNECT 表示，在第一个方法调用时，客户的安全凭证必须要在服务器端进行认证。如果客户没有有效的安全凭证，那么 RPC 调用就会失败，返回值为 E\_ACCESSDENIED。如何检验这些安全凭证要取决于客户先前所使用的 SSP。在 NTLM 下，服务器进程给客户进程发送一个询问信息 (challenge)。询问信息只是一个简单的、不可预测的大随机数。客户使用经过编码之后的调用者口令来加密这个询问信息，然后被作为应答 (response) 送回给服务器。然后服务器使用它自己认为的口令（当然也要经过同样的编码方法）来加密原始的询问信息，并且用这个结果与它从客户处接收到的应答进行比较。如果客户的应答与服务器方加密之后的询问信息相互匹配的话，那么客户的身份就假定是经过认证了。因为 NTLMSSP 把“询问-应答的握手数据”附加在“RPC 运行时为了同步序列号而发送的初始数据包”上，所以它不会增加额外的“客户-服务器”之间的网络流量。根据帐号的类型（域帐号或者本地帐号），可能会有（也可能没有）额外的流量流向域控制器，以提供认证支持。

如果使用了 RPC\_C\_AUTHN\_LEVEL\_CONNECT 认证级别，那么一旦初始安全凭

证已经检验完成之后，就不会再交换额外的与安全有关的信息。这意味着，有些恶意的程序可以有意识地截取网络上的消息，然后修改包头中的 DCE 序列号就可以重放 RPC 调用。为了避免这种调用重放行为，我们应该使用 `RPC_C_AUTHN_LEVEL_CALL` 认证级别。它会告诉 SSP DLL，应该要保护每个 RPC 请求或者应答的第一个包的 RPC 首部，其做法是把被传送的包中的一个单向散列键连接起来。因为每个 RPC 请求或者应答有可能会被分割成多个网络数据包，所以 RPC API 也支持 `RPC_C_AUTHN_LEVEL_PKT` 认证级别。这个级别可以在每个网络数据包的层次上避免重放攻击，因为 RPC 消息有可能超过两个或者更多个数据包，所以它的保护能力超过 `RPC_C_AUTHN_LEVEL_CALL` 认证级别。

在 `RPC_C_AUTHN_LEVEL_PKT` 之前（也包括它在内）的认证级别中，SSP DLL 多多少少会忽略掉实际的 RPC 包净荷数据，而只是保护 RPC 头的完整性。为了确保被列集之后的参数状态不会被网络上的恶意程序修改，RPC 提供了 `RPC_C_AUTHN_LEVEL_PKT_INTEGRITY` 认证级别。此认证级别使得 SSP DLL 针对被列集之后的参数执行一个检验和，并检验包的内容是否在传输过程中被修改了。因为这个认证级别要求每个被传输的字节都要经过 SSP DLL 处理，所以它比 `RPC_C_AUTHN_LEVEL_PKT` 级别要慢得多，应该仅用于安全性要求很高的场合。

在 `RPC_C_AUTHN_LEVEL_PKT_INTEGRITY` 之前（也包括它在内）的认证级别中，RPC 包的实际净荷数据是以明文方式发送的（也即没有经过加密）。为了确保被列集之后的参数状态不会被网络上的恶意程序看到，RPC 提供了 `RPC_C_AUTHN_LEVEL_PKT_PRIVACY` 认证级别。此认证级别使得 SSP DLL 在传输被列集之后的参数状态之前，先对它们进行加密。如同其他的认证级别一样，`RPC_C_AUTHN_LEVEL_PKT_PRIVACY` 也包含在它之下的每一个安全级别所提供的保护。同 `RPC_C_AUTHN_LEVEL_PKT_INTEGRITY` 一样，每个被传输的字节都要经过 SSP DLL 处理，所以它应该仅用于安全性要求很高的场合，以避免过多的额外负担。

关于 COM 安全性，最重要的 API 函数是 `CoInitializeSecurity`。每一个使用 COM 的进程都会调用 `CoInitializeSecurity` 一次（可能是显式调用，也可能是隐式调用）。`CoInitializeSecurity` 建立起自动的安全设置。这些设置适用于所有被引入的（imported）和被引出的（exported）对象引用，除非使用其他的 API 调用显式地改变它们的设置。`CoInitializeSecurity` 也会配置底层的 RPC 运行时，使它使用一个或者多个安全包，同时还为进程设置缺省的认证级别。除此之外，`CoInitializeSecurity` 也允许调用者指定哪个用户可以向“当前进程所引出的对象”发出 ORPC 调用。`CoInitializeSecurity` 有大量的参数，其原型如下：

```
HRESULT CoInitializeSecurity(
    [in] PSECURITY_DESCRIPTOR pSecDesc, // 访问控制
    [in] LONG cAuthSvc, // 安全包号 (-1 == 使用缺省值)
    [in] SOLE_AUTHENTICATION_SERVICE *rgsAuthSvc, // SSP 数组
    [in] void *pReserved1, // 保留的 MBZ
```

```

[in] DWORD dwAuthnLevel,      // 自动 AUTHN_LEVEL
[in] DWORD dwImpLevel,       // 自动 IMP_LEVEL
[in] void *pReserved2,        // 保留的 MBZ
[in] DWORD dwCapabilities,   // 其他标志
[in] void *pReserved3,        // 保留
);

```

其中有的参数只有在当前进程被当作引出器/服务器的时候才有用；而另一些参数则只有在当前进程被当作引入器/客户的时候才有用；而其他的参数可同时适用于这两种情况。

`CoInitializeSecurity` 的第一个参数 `pSecDesc` 仅适用于当前进程为引出器的情形。它可用来控制哪些安全个体能够访问被当前进程引出的对象。本章后面将会进一步讨论这个参数的细节。`CoInitializeSecurity` 的第二和第三个参数，`cAuthSvc` 和 `rgsAuthSvc` 被用于当进程为引出器的情形，它们可用来把一个或者多个认证软件包注册到 COM 库中。这两个参数涉及到一个安全包描述（`SOLE_AUTHENTICATION_SERVICE`）数组，定义如下：

```

typedef struct tagSOLE_UTHENTICATION_SERVICE {
    DWORD    dwAuthnSvc;          // 哪个认证包?
    DWORD    dwAuthzSvc;          // 哪个认证服务?
    OLECHAR *pPrincipalName;     // 服务器主名?
    HRESULT hr;
} SOLE_AUTHENTICATION_SERVICE;

```

在 Windows NT 4.0 下面，只有 `RPC_C_AUTHN_WINNT` (NTLM) 认证服务被安装了。当使用 NTLM 认证的时候，授权服务 (authorization service) 必须是 `RPC_C_AUTHZ_NONE`，并且服务器个体名没有被用到，且必须为 `null`。<sup>5</sup> 如果有的进程只是希望使用当前机器上的缺省安全包，那么应该使用 -1 (`cAuthSvc`) 和 `null` (`rgsAuthSvc`)。

`CoInitializeSecurity` 的第五个参数 `dwAuthnLevel` 既适用于被引出的对象引用，也适用于被引入的对象引用。这个参数的值可以为当前进程所引出的对象引用设置最低水平的认证级别。这意味着进来的 ORPC 调用 (即入 ORPC 调用) 必须至少使用这个认证级别；否则这个调用会自动被拒绝。这个值也指定了“COM API 函数或者方法所返回的新接口代理”所使用的最小认证。当 COM 在散集时创建一个新的接口代理的时候，COM 找到引出器的最低水平认证级别 (作为 OXID 解析的一部分)。然后 COM 把新代理的认证级别设置为引出器的最低水平认证级别，或者当前进程的最低水平认证级别，到底设置成哪个取决于哪个更高。如果引入对象引用的进程，它的认证级别低于引出进程的认证级别，那么就使用引出器的最低水平认证级别作为接口代理的认证级别。这样做可以确保接口代理所发送的任何一个 ORPC 请求将可以通过引出器的最低水平。正如本章后面

<sup>5</sup> 这两个参数将来有可能被用于其他的认证包。

将要讨论到的，对于要求控制粒度更细的特殊接口代理，它有可能显式地改变认证级别。<sup>6</sup>

`CoInitializeSecurity` 的第六个参数 `dwImpLevel` 适用于被引入的对象引用。这个参数的值为所有被 `CoUnmarshalInterface` 返回的对象引用设置模仿级别（`impersonation level`）。模仿级别代表了客户对服务器的信任程度。这个参数必须是下面四个模仿级别之一：

```
enum {
    // 对对象隐藏调用者的凭证
    RPC_C_IMP_LEVEL_ANONYMOUS = 1,
    // 允许对象查询调用者凭证
    RPC_C_IMP_LEVEL_IDENTIFY = 2,
    // 允许在一跳(one-hop)内使用调用者的凭证
    RPC_C_IMP_LEVEL_IMPERSONATE = 3,
    // 允许跨过多跳使用调用者凭证
    RPC_C_IMP_LEVEL_DELEGATE = 4
};
```

`RPC_C_IMP_LEVEL_ANONYMOUS` 级别阻止对象实现代码发现调用者的安全标识符。<sup>7</sup> `RPC_C_IMP_LEVEL_IDENTIFY` 信任级别表示，对象可以在程序中确定调用者的安全标识符。`RPC_C_IMP_LEVEL_IMPERSONATE` 信任级别表示，服务器不仅可以确定调用者的安全标识符，还可以利用调用者的安全凭证执行操作系统层次（OS-level）上的操作。这个信任级别也有限制，当对象使用调用者的安全凭证的时候，它只能访问本地的资源。<sup>8</sup> 相反，`RPC_C_IMP_LEVEL_DELEGATE` 信任级别允许服务器利用调用者的安全凭证，既可以访问本地的资源，也可以访问远程资源。NTLM 认证协议不支持这种信任级别，但是 Kerberos 认证协议支持。

`CoInitializeSecurity` 的第八个参数 `dwCapabilities` 可同时适用于被引入的和被引出的对象引用。这个参数是一个位掩码，其中包含下面 0 个或者多个位：

```
typedef enum tagEOLE_AUTHENTICATION_CAPABILITIES {
    EOAC_NONE          = 0x0,
    EOAC_MUTUAL_AUTH   = 0x1,
    // 这些只对 CoInitializeSecurity 有效
    EOAC_SECURE_REFS    = 0x2,
    EOAC_ACCESS_CONTROL = 0x4,
```

<sup>6</sup> 某些安全包可能会根据所使用的传输协议，增加客户或者服务器指定的认证级别。特别地，对于在同一台机器上的所有调用，NTLM 都使用 `RPC_C_AUTHN_LEVEL_PRIVACY`。而且，对于数据报传输协议（例如 UDP），NTLM 将把 `RPC_AUTHN_LEVEL_CONNECT` 和 `RPC_AUTHN_LEVEL_CALL` 提升为 `RPC_C_AUTHN_LEVEL_PKT`；而对于面向连接的传输协议（例如 TCP），NTLM 将把 `RPC_C_AUTHN_LEVEL_CALL` 提升为 `RPC_C_AUTHN_LEVEL_PKT`。

<sup>7</sup> 在本书写作的时候，NTLM 和 Kerberos SSP 都能够接受这个值，但如果连接对象是远程机器的话，就自动将它提升为 `RPC_C_IMP_LEVEL_IDENTIFY`。

<sup>8</sup> 从技术上讲，`RPC_C_IMP_LEVEL_IMPERSONATE` 允许调用者的安全凭证至多在一个网络跳（`network hop`）上被传输。这就有效地限制了远程对象只能访问对象本机上的资源。

```

EOAC_APPID          = 0x8
} EOLE_AUTHENTICATION_CAPABILITIES;

```

NTLM 不支持双向认证 (EOAC\_MUTUAL\_AUTH)。它可用于验证服务器是否正以期望中的安全个体的身份在运行。安全的引用 (EOAC\_SECURE\_REFS) 指示 COM 的分布式引用计数调用将被认证，以确保不会有恶意的程序损害“OR 或者存根管理器为实现生命周期管理而用到的”引用计数。EOAC\_ACCESS\_CONTROL 和 EOAC\_APPID 被用于控制 CoInitializeSecurity 第一个参数的语义，将在本章后面进一步讨论。

正如本节前面所说的，CoInitializeSecurity 只被每个进程调用一次，或者是显式的，或者是隐式的。希望显式调用 CoInitializeSecurity 的应用必须要在第一个 CoInitializeEx 之后，但是在“第一个令人感兴趣的 COM 调用”之前调用这个函数。这里的“第一个令人感兴趣的 COM 调用”是指任何一个有可能要求 OXID 的 API 函数。这包括 CoMarshalInterface 和 CoUnmarshalInterface 调用，以及有可能隐式调用它们的函数。因为 CoRegisterClassObject 调用要把类对象与套间关联起来，所以 CoInitializeSecurity 必须要在注册类对象之前被调用。激活 API（比如 CoCreateInstanceEx）是一个有趣的例外。对于属于 COM API 本身的内部类（比如全局接口表、COM 的缺省访问控制对象）的激活 API，它们可以在调用 CoInitializeSecurity 之前被调用。但是，对于要实际查询注册表、装载其他 DLL 或者与其他服务器联系的激活调用，CoInitializeSecurity 必须要在这些调用之前先被调用。如果一个应用没有显式地调用 CoInitializeSecurity，那么 COM 会在第一个令人感兴趣的 COM 调用之前隐式地调用它。

当 COM 隐式地调用 CoInitializeSecurity 的时候，它从注册表读取到大多数参数的值；有的参数被保存在机器范围内的键中，而有的参数被保存在应用的 AppID 下面。为了获得应用的 AppID，COM 在 HKEY\_CLASSES\_ROOT\AppID 键下查找应用进程的文件名。如果 COM 在这里找到了文件名，它就从 AppID 名字值中获得 AppID，例如：

```
HKEY_CLASSES_ROOT\ApplD
```

如果没有找到匹配，则 COM 假定这个应用没有把特定的安全设置保存在注册表中。

隐式的 CoInitializeSecurity 调用在下面的名字值中找到被序列化之后的 NT SECURITY\_DESCRIPTOR，作为第一个参数 pSecDesc。名字值如下：

```
[HKCR\AppID\ServerOfTheApes.exe]
ApplD="{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}"
```

如果这个名字值没有被找到，那么 COM 就会查找机器范围内的表项：

```
[HKCR\AppID\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}]
AccessPermission=<serialized NT security descriptor>
```

使用 DCOMCNFG.EXE 可以很容易地修改这两个注册表项。如果这两个表项都没有被发现，那么 COM 将会创建一个安全描述符，只把访问权赋给调用者的安全个体和内

置的 SYSTEM 帐号。COM 使用这个安全描述符，通过 Win32 API 函数 AccessCheck，允许或者禁止访问被该进程引出的对象。

隐式的 CoInitializeSecurity 调用使用 -1 和 null 作为第二和第三个参数（cAuthSvc 和 rgsAuthSvc），表示使用缺省的安全包。隐式的 CoInitializeSecurity 调用在以下的（机器范围内的）注册表键中找到第五和第六个参数的值，注册表键如下：

```
[HKEY_LOCAL_MACHINE\Software\Microsoft\OLE]
LegacyAuthenticationLevel = 0x5
LegacyImpersonationLevel = 0x3
```

数值 5 和 3 分别对应于 RPC\_C\_AUTHN\_LEVEL\_PKT\_INTEGRITY 和 RPC\_C\_IMP\_LEVEL\_IMPERSONATE。如果这些名字值都没有出现，那么就使用 RPC\_C\_AUTHN\_LEVEL\_CONNECT 和 RPC\_C\_IMP\_LEVEL\_IDENTIFY。至于 CoInitializeSecurity 的第八个参数 dwCapabilities，目前只可能从下面的（机器范围内的）注册表键中读取到 EOAC\_SECURE\_REFS：

```
[HKEY_LOCAL_MACHINE\Software\Microsoft\OLE]
LegacySecureRefs = "Y"
```

如果这个名字值出现了并且包含“Y”或者“y”，那么 COM 将使用 EOAC\_SECURE\_REFS 标志；否则 COM 使用 EOAC\_NONE 标志。通过 DCOMCNFG.EXE 工具，这三个遗留下来的认证设置很容易可被修改掉。

## 6.6 通过编程实现安全性

通过 CoInitializeSecurity 获得的安全性设置被称为自动安全性设置，因为它们自动适用于所有被列集的对象引用。但是通常会有小部分的对象引用需要使用不同于进程缺省的安全性设置。最常见的情形是，为了考虑效率起见，进程只使用比较低的认证级别，但是某一个接口要求加密功能。与其强迫整个进程使用加密功能，不如简单地只让有必要的对象引用才使用加密功能。

为了允许开发人员可以在接口代理的基础上改写掉自动安全性设置，代理管理器暴露了 IClientSecurity 接口：

```
[ local,object,uuid(0000013D-0000-0000-C000-00000000046) ]
interface IClientSecurity : IUnknown {
    // 获得接口代理的安全设置 pProxy
    HRESULT QueryBlanket([in]    IUnknown *pProxy,
                        [out]   DWORD *pAuthnSvc,  [out]   DWORD *pAuthzSvc,
```

```

[out] OLECHAR **pServerPrincName,
[out] DWORD *pAuthnLevel, [out] DWORD *pImpLevel,
[out] void **pAuthInfo, [out] DWORD *pCapabilities
);
// 改变接口代理 pProxy 的安全设置
HRESULT SetBlanket([in] IUnknown *pProxy,
[in] DWORD AuthnSvc, [in] DWORD AuthzSvc,
[in] OLECHAR *pServerPrincName,
[in] DWORD AuthnLevel, [in] DWORD ImpLevel,
[in] void *pAuthInfo, [in] DWORD Capabilities
);
// 复制一个接口代理
HRESULT CopyProxy([in] IUnknown *pProxy,
[out] IUnknown **ppCopy
);
}
}

```

SetBlanket 和 QueryBlanket 的第二、第三和第四个参数分别对应于 SOLE\_AUTHENTICATION\_SERVICE 数据结构的三个成员。在 Windows NT 4.0 下面，唯一合法的值分别为 RPC\_C\_AUTHN\_WINNT、RPC\_C\_AUTHZ\_NONE 和 null。

正如图 6.1 所示，每个单独的接口代理可以有自己的认证设置。IClientSecurity::SetBlanket 方法允许调用者可以为了某个特殊的接口代理而改变这些设置。IClientSecurity::QueryBlanket 方法允许调用者读取某个接口代理的认证设置。如果调用

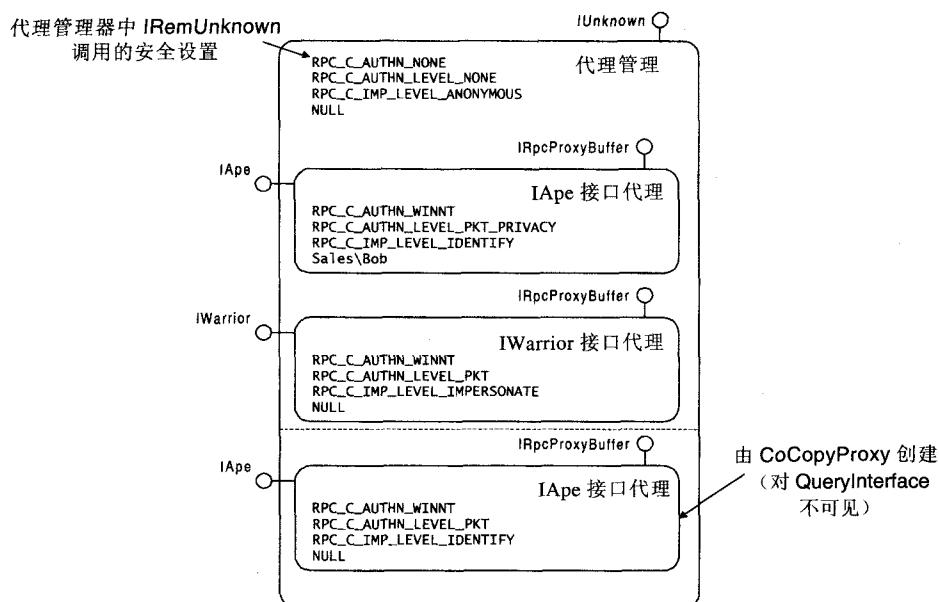


图 6.1 安全性和代理

者不关心某些参数，那么它可以传递空 (null) 指针。IClientSecurity::CopyProxy 方法允

许多调用者复制一个接口代理。这使得我们可以对一个接口的复制版本作修改，以后在代理管理器上调用 `QueryInterface` 不会返回复制版本的接口代理。理想的做法是，安全设置只应该作用在复制版本的接口代理上，从而把修改后的代理与代理管理器正常的 `QueryInterface` 实现隔离开来，也可让多个线程在方法调用之间相互独立地改变安全设置。

`IClientSecurity::SetBlanket` 和 `IClientSecurity::QueryBlanket` 的所有参数都与 `CoInitializeSecurity` 的参数相对应，只有一个显著的例外。第七个参数 `pAuthInfo` 指向一组客户安全凭证。这些安全凭证的确切格式与相应的安全包有关。对于 NTLM 安全包，这个参数指向一个 `COAUTHIDENTITY` 结构：

```
typedef struct _COAUTHIDENTITY {
    OLECHAR *User;           // 用户帐号名
    ULONG UserLength;        // wcslen(用户)
    OLECHAR *Domain;         // 机器名
    ULONG DomainLength;      // wcslen(域)
    OLECHAR *Password;       // 明文密码
    ULONG PasswordLength;   // wcslen(密码)
    ULONG Flags;             // 必须是 SEC_WINNT_AUTH_IDENTITY_UNICODE
} COAUTHIDENTITY;
```

这个结构允许客户以任意的安全个体身份来发出 COM 方法调用，只要客户知道该帐户明文形式的口令即可。<sup>9</sup> 如果这个参数为 null，而不是指向某个显式的 `COAUTHIDENTITY` 结构的指针，那么每个对外的调用都将使用调用进程的安全凭证。<sup>10</sup>

`IClientSecurity::SetBlanket` 较为常见的用法是提高一个代理的认证级别。下面的代码演示了这项技术：

```
HRESULT Encrypt(IApe *pApe) {
    IClientSecurity *pcs = 0;
    // 向代理管理器请求 IClientSecurity 接口
    HRESULT hr = pApe->QueryInterface(IID_IClientSecurity,
                                         (void**)&pcs);
    if (SUCCEEDED(hr)) {
        hr = pcs->SetBlanket(pApe, RPC_C_AUTHN_WINNT,
                               RPC_C_AUTHZ_NONE, 0,
                               RPC_C_AUTHN_LEVEL_PKT_PRIVACY,
                               RPC_C_IMP_LEVEL_IDENTIFY,
                               0, EOAC_NONE);
        pcs->Release();
    }
}
```

<sup>9</sup> 有一点很重要需要特别指出，在本书写作的时候，同一台机器上的通信不支持 `COAUTHIDENTITY`。它只适用于远程主机的通信。

<sup>10</sup> 在 Windows NT 5.0（即 Windows 2000）中，对于委托级别的模仿支持有可能会改变这种行为，改成使用调用线程的令牌。请查阅最新的文档以获得更多的信息。

```
    return hr;
}
```

理想的做法是，调用者把经过复制得到的接口代理传给这个函数。或者，我们可以修改这个函数，由它来执行复制操作，代码如下：

```
HRESULT DupeAndEncrypt(IApe *pApe, IAp * &rpSecretApe) {
    rpSecretApe = 0;
    IClientSecurity *pcs = 0;
    // 向代理管理器请求 IClientSecurity 接口
    HRESULT hr = pApe->QueryInterface(IID_IClientSecurity,
                                         (void**)&pcs);
    if (SUCCEEDED(hr)) {
        hr = pcs->CopyProxy(pApe, (IUnknown**)&rpSecretApe);
        if (SUCCEEDED(hr))
            hr = pcs->SetBlanket(rpSecretApe, RPC_C_AUTHN_WINNT,
                                   RPC_C_AUTHZ_NONE, 0,
                                   RPC_C_AUTHN_LEVEL_PKT_PRIVACY,
                                   RPC_C_IMP_LEVEL_IDENTIFY,
                                   0, EOAC_NONE);
        pcs->Release();
    }
    return hr;
}
```

为方便起见，COM API 为 IClientSecurity 的三个方法分别提供了包装函数，在包装函数的内部，它们调用 QueryInterface 得到对应的 IClientSecurity 接口，然后调用适当的方法，这三个包装函数的原型如下：

```
// 获取接口代理 pProxy 的安全设置
HRESULT CoQueryProxyBlanket([in] IUnknown *pProxy,
                            [out] DWORD *pAuthnSvc, [out] DWORD *pAuthzSvc,
                            [out] OLECHAR **pServerPrincName,
                            [out] DWORD *pAuthnLevel, [out] DWORD *pImpLevel,
                            [out] void **pAuthInfo, [out] DWORD *pCapabilities);

// 改变接口代理 pProxy 的安全设置
HRESULT CoSetProxyBlanket([in] IUnknown *pProxy,
                           [in] DWORD AuthnSvc, [in] DWORD AuthzSvc,
                           [in] OLECHAR *pServerPrincName,
                           [in] DWORD AuthnLevel, [in] DWORD ImpLevel,
                           [in] void *pAuthInfo, [in] DWORD Capabilities)

// 复制接口代理
HRESULT CoCopyProxy([in] IUnknown *pProxy,
                     [out] IUnknown **ppCopy);

HRESULT DupeAndEncrypt(IApe *pApe, IAp * &rpSecretApe) {
    rpSecretApe = 0;
```

```

HRESULT hr = CoCopyProxy(pApe, (IUnknown**)&rpSecretApe);
if (SUCCEEDED(hr))
    hr = CoSetProxyBlanket(rpSecretApe, RPC_C_AUTHN_WINNT,
                          RPC_C_AUTHZ_NONE, 0,
                          RPC_C_AUTHN_LEVEL_PKT_PRIVACY,
                          RPC_C_IMP_LEVEL_IDENTIFY,
                          0, EOAC_NONE);
return hr;

```

下面的代码是前面给出的 `DupeAndEncrypt` 函数修改之后的版本，它用到了其中一个包装函数，代码如下：

```

void TurnOffAllSecurity(IApe *pApe) {
    IUnknown *pUnkProxyManager = 0;
    // 获取代理管理器的指针
    HRESULT hr = pApe->QueryInterface(IID_IUnknown,
                                         (void**)&pUnkProxyManager);
    assert(SUCCEEDED(hr));
    // 全面设置代理管理器
    hr = CoSetProxyBlanket(pUnkProxyManager,
                          RPC_C_AUTHN_NONE, RPC_C_AUTHZ_NONE,
                          0, RPC_C_AUTHN_LEVEL_NONE,
                          RPC_C_IMP_LEVEL_ANONYMOUS,
                          0, EOAC_NONE);
    assert(SUCCEEDED(hr));
    // 全面设置接口代理
    hr = CoSetProxyBlanket(pApe,
                          RPC_C_AUTHN_NONE, RPC_C_AUTHZ_NONE,
                          0, RPC_C_AUTHNLEVEL_NONE,
                          RPC_C_IMP_LEVEL_ANONYMOUS,
                          0, EOAC_NONE);
    assert(SUCCEEDED(hr));
    // 释放代理管理器的临时指针
    pUnkProxyManager->Release();
}

```

前一个版本的效率比较高，因为只需一次调用 `QueryInterface` 请求 `IClientSecurity` 接口。后一个版本要求的代码少一些，因而代码出错率低一些。

有一点很重要，那就是 `IClientSecurity` 方法只能适用于使用接口代理的接口。这意味着，代理管理器本地实现的接口（比如 `IMultiQI`、`IClientSecurity`）不能够被用于 `IClientSecurity` 的方法。从技术上讲，`IUnknown` 是代理管理器实现的本地接口。然而，代理管理器往往要与服务器的套间进行通信，以请求新的接口，以及释放存根管理器所拥有的资源。这个通信过程发生在私有接口 `IRemUnknown` 上，这个接口由 COM 库内部实现（以每个套间作为基础）。开发人员只要把代理管理器的 `IUnknown` 传给 `IClientSecurity::SetBlanket`，就可以控制 `IRemUnknown` 调用所使用的安全设置。（与接

口代理一样，如果客户不针对代理管理器调用 SetBlanket 的话，代理管理器就使用进程范围内的自动安全性设置。)<sup>11</sup> 因为所有的接口代理都被代理管理器聚合了，所以，QueryInterface、AddRef 和 Release 实际上都不受任何一个接口代理上 IClientSecurity::SetBlanket 调用的影响。但是它们却受代理管理器的 IUnknown 的安全设置的影响。为了得到代理管理器的 IUnknown 接口指针，我们只要向任一个接口代理调用 QueryInterface，请求 IID\_IUnknown 即可。下面的代码演示了这项技术，它同时禁止接口代理和代理管理器的安全性：

```

void TurnOffAllSecurity(IApe *pApe) {
    IUnknown *pUnkProxyManager = 0;
    // 获取代理管理器的指针
    HRESULT hr = pApe->QueryInterface(IID_IUnknown,
                                         (void**)&pUnkProxyManager);
    assert(SUCCEEDED(hr));
    // 全面设置代理管理器
    hr = CoSetProxyBlanket(pUnkProxyManager,
                           RPC_C_AUTHN_NONE, RPC_C_AUTHZ_NONE,
                           0, RPC_C_AUTHN_LEVEL_NONE,
                           RPC_C_IMP_LEVEL_ANONYMOUS,
                           0, EOAC_NONE);
    assert(SUCCEEDED(hr));
    // 全面设置接口代理
    hr = CoSetProxyBlanket(pApe,
                           RPC_C_AUTHN_NONE, RPC_C_AUTHZ_NONE,
                           0, RPC_C_AUTHNLEVEL_NONE,
                           RPC_C_IMP_LEVEL_ANONYMOUS,
                           0, EOAC_NONE);
    assert(SUCCEEDED(hr));
    // 释放代理管理器的临时指针
    pUnkProxyManager->Release();
}

```

虽然我们可以设置和查询代理管理器的安全性，但是我们不能够使用 IClientSecurity::CopyProxy 来复制代理管理器，因为这样做违反了 COM 的身份一致性法则。

当一个 ORPC 请求被分发到某个接口存根的时候，COM 创建一个调用环境对象，它表达了这个调用各方面的状况，其中包括发出这个请求的接口代理的安全设置。COM 把这个环境对象与执行这个方法调用的线程联系起来。COM 库暴露了一个 API 函数 CoGetCallContext，它允许方法的实现代码访问当前方法调用的环境，此 API 函数的原

<sup>11</sup> 这句话有两点需要稍作修正。第一，如果客户进程在它的 CoInitializeSecurity 调用中被配置成使用安全的引用，那么 IRemUnknown::RemAddRef 和 IRemUnknown::RemRelease 调用将使用进程的安全个体身份，而不是在 IClientSecurity::SetBlanket 调用中指定的安全个体身份。第二，在 Windows NT 4.0 Service Pack 4 之前的版本中，所有的 IRemUnknown::RemAddRef 和 IRemUnknown::RemRelease 调用都使用进程的安全个体身份，而不管代理管理器的安全设置是什么。

型如下：

```
HRESULT CoGetCallContext([in] REFIID riid,
                         [out, iid_is(riid)] void **ppv);
```

在 Windows NT 4.0 中，这个调用环境对象唯一可供使用的接口就是 **IServerSecurity** 接口：

```
[ local, object, uuid(0000013E-0000-0000-C000-00000000046) ]
interface IServerSecurity : IUnknown {
// 获取调用者安全设置
    HRESULT QueryBlanket(
        [out] DWORD      *pAuthnSvc,          // 认证包
        [out] DWORD      *pAuthzSvc,         // 授权包
        [out] OLECHAR   **pServerName,       // 服务器主名字
        [out] DWORD      *pAuthnLevel,        // 认证级别
        [out] DWORD      *plmpLevel,         // 模仿级别
        [out] void       **pPrivils,          // 客户方主名字
        [out] DWORD      *pCaps             // EOAC 标志
    );
// 用调用者凭证开始运行
    HRESULT ImpersonateClient(void);
// 停止
    HRESULT RevertToSelf(void);
// 测试模仿?
    BOOL IsImpersonating(void);
}
```

**IServerSecurity::QueryBlanket** 返回用于发出当前 ORPC 调用的安全性设置（由于 SSP 可能会提升安全级别，所以对象得到的安全设置有可能与客户的设置不同）。如同 **IClientSecurity::QueryBlanket** 的情形一样，**IServerSecurity::QueryBlanket** 允许客户对不感兴趣的参数传递 **null**。以下是一个方法实现的例子代码，它可以确保调用者已经设置了加密选项，然后才作实际的处理：

```
STDMETHODIMP Gorilla::SwingFromTree(/*[in]*/ long nTreeID) {
// 获取当前调用环境
    IServerSecurity *pss = 0;
    HRESULT hr = CoGetCallContext(IID_IServerSecurity,
                                  (void**)&pss);
    DWORD dwAuthnLevel;
    if (SUCCEEDED(hr)) {
// 获取当前调用的认证级别
        hr = pss->QueryBlanket(0, 0, 0, &dwAuthnLevel, 0, 0, 0);
        pss->Release();
    }
// 确认认证级别正确
    if (FAILED(hr) ||
```

```

dwAuthnLevel != RPC_C_AUTHN_LEVEL_PKT_PRIVACY)
hr = APE_E_NOPUBLICTREE;
else
    hr = this->ActuallySwingFromTree(nTreeID);
return hr;
}

```

如同 IClientSecurity 的情形一样，IServerSecurity 的每个方法也都有一个对应的便利 API 函数可供使用。下面的方法实现代码使用了这样的包装函数，以替代显式的 IServerSecurity 接口调用：

```

STDMETHODIMP Gorilla::SwingFromTree(/*[in]*/ long nTreeID) {
    DWORD dwAuthnLevel;
    // 获取当前调用的认证级别
    HRESULT hr = CoQueryClientBlanket(0, 0, 0, &dwAuthnLevel,
                                      0, 0, 0);
    // 确认认证级别正确
    if (FAILED(hr) ||
        dwAuthnLevel != RPC_C_AUTHN_LEVEL_PKT_PRIVACY)
        hr = APE_E_NOPUBLICTREE;
    else
        hr = this->ActuallySwingFromTree(nTreeID);
    return hr;
}

```

再次说明，上面这个版本要求更少的代码，因而减少了代码出错的机会。

IServerSecurity::QueryBlanket 也允许对象实现者通过 pPrvs 参数找到调用者的安全标识符。如同安全凭证被传给 IClientSecurity::SetBlanket 的情形一样，此标识符确切的格式与特定的安全包有关。对于 NTLM，这个格式非常简单，为如下形式的字符串：

```
Authority\AccountName
```

下面的方法实现使用 CoQueryClientBlanket API 函数，获取到调用者的安全标识符：

```

STDMETHODIMP Gorilla::EatBanana() {
    OLECHAR *pwszClientPrincipal = 0,
    // 获取调用者安全标识符
    HRESULT hr = CoQueryClientBlanket(0, 0, 0, 0, 0,
                                      (void**)&pwszClientPrincipal, 0);
    // 记录用户名
    if (SUCCEEDED(hr)) {
        this->LogCaller1DToFile(pwszClientPrincipal );
        hr = this->ActuallyEatBanana();
    }
    return hr;
}

```

为了让 CoQueryClientBlanket 调用能成功地返回调用者的安全标识符，调用者必须

已经指定：（1）至少 `RPC_C_IMP_LEVEL_IDENTIFY` 作为自动的（或者显式的）模仿级别；（2）至少 `RPC_C_AUTHN_LEVEL_CONNECT` 作为自动的（或者显式的）认证级别。如果调用者在代理的安全设置中用 `COAUTHIDENTITY` 改变了调用安全个体，那么 `CoQueryClientBlanket` 将返回这个被显式指定的安全个体名。

正如客户在发出方法调用时，可以用 `IClientSecurity` 接口来完全控制安全性设置一样，客户在发出激活调用时控制安全性设置，这也是非常有用的。不幸的是，激活调用是全局 API 函数，没有相应的代理管理器可供获取 `IClientSecurity` 接口。为了允许调用者为激活调用指定安全设置，每个激活调用都接受一个 `COSERVERINFO` 结构：

```
typedef struct _COSERVERINFO {
    DWORD          dwReserved1;
    LPWSTR         pwszName;
    COAUTHINFO *   pAuthInfo;
    DWORD          dwReserved2;
} COSERVERINFO;
```

正如上一章所说明的，`pwszName` 数据成员允许调用者显式地控制“哪台机器将处理这个激活请求”。第三个数据成员 `pAuthInfo` 指向一个数据结构，它允许调用者控制这个激活调用的安全设置。这个参数是一个指针，指向 `COAUTHINFO` 结构，其定义如下：

```
typedef struct _COAUTHINFO {
    DWORD          dwAuthnSvc;
    DWORD          dwAuthzSvc;
    LPWSTR         pwszServerPrincName;
    DWORD          dwAuthnLevel;
    DWORD          dwImpersonationLevel;
    COAUTHIDENTITY * pAuthIdentityData;
    DWORD          dwCapabilities;
} COAUTHINFO;
```

这些数据成员对应于 `IClientSecurity::SetBlanket` 的参数，但是仅用于激活调用过程，它们不会影响到结果返回的接口代理。<sup>12</sup>

下面的代码片断在发出一个激活请求的时候，利用 `COAUTHINFO` 结构来发出激活调用，强迫 SCM 使用加密认证级别（`RPC_C_AUTHN_LEVEL_PKT_PRIVACY`）：

```
void CreateSecretChimp(IApe **&rpApe) {
    rpApe = 0;
    // 创建指定加密的 COAUTHINFO
```

<sup>12</sup> 很重要的一点是，由于激活调用的初始接收者是服务器端的 SCM，所以有些认证包可能并不被支持。Windows NT 4.0 SCM 只支持 NTLM。请参考最新的文档，以获取 Windows NT 5.0（即 Windows 2000）下认证支持的详细信息。

```

COAUTHINFO cai = {
    RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE, 0,
    RPC_C_AUTHN_LEVEL_PKT_PRIVACY,
    RPC_C_IMP_LEVEL_IDENTIFY,
    0, 0
};

// 使用 COAUTHINFO 发出一个激活调用
COSERVERINFO csi = { 0, 0, &cai, 0 };
IApeClass *pac = 0;
hr = CoGetClassObject(CLSID_Chimp, CLSCTX_ALL, &csi,
                      IID_IApeClass, (void**)&pac);
assert(SUCCEEDED(hr));
// 激活调用本身加密
// 而 pac 仍使用自动安全设置
hr = pac->CreateApe(&rpApe);
pac->Release();
return hr;
}

```

很重要的一点是，因为 **COAUTHINFO** 结构只影响到激活调用本身，所以结果得到的 **IApeClass** 接口代理将使用自动安全设置，即原先在 **CoInitializeSecurity** 调用中设置的安全参数。这意味着，**IApeClass::CreateApe** 将使用自动安全设置，而不是在 **COAUTHINFO** 结构中指定的安全设置。为了确保在创建和操纵新 **Chimp** 对象的过程中仍然使用加密设置，我们有必要对该函数作修改，以便对 **IApeClass** 和 **IApe** 接口代理都作安全性设置：

```

// 对 IApeClass 引用进行加密调用
CoSetProxyBlanket(pac, RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE,
                  0, RPC_C_AUTHN_LEVEL_PKT_PRIVACY,
                  RPC_C_IMP_LEVEL_ANONYMOUS, 0, EOAC_NONE);
// 发生创建对象的调用
pac->CreateApe(&rpApe);
// 对 IApe 引用进行加密调用
CoSetProxyBlanket(rpApe, RPC_C_AUTHN_WINNT, RPC_C_AUTHZ_NONE,
                  0, RPC_C_AUTHN_LEVEL_PKT_PRIVACY,
                  RPC_C_IMP_LEVEL_NONYMOUS, 0, EOAC_NONE);

```

在激活时刻使用显式的 **COAUTHIDENTITY**，这可使调用者在“调用进程本身无法访问的进程”中创建对象。然而，在释放接口指针的时候，调用者有责任保证代理管理器使用同样的安全凭证，否则服务器端的资源将会被泄漏。正如本章前面所提到的，我们可以通过在代理管理器的 **IUnknown** 接口上调用 **IClientSecurity::SetBlank**，以单独控制代理管理器的安全性。

## 6.7 访问控制

正如本章前面所提到的，每个 COM 进程都可以保护自己避免未经授权的访问。COM 在两个层次上考虑这个问题：激发许可（launch permission）和访问许可（access permission）。激发许可以用来决定“在向 SCM 发出激活调用的时候，哪些用户可以启动服务器进程”。访问许可以决定“一旦服务器被启动之后，哪些用户可以访问进程中的对象”。这两种类型的访问许可以够通过 DCOMCNFG.EXE 来设置，但是只有访问许可以在运行时由程序来指定（因为一旦服务器进程启动之后，再要拒绝用户的激发请求已经太迟了）。相反，激发许可以检查是由 SCM 在激活时刻强制实施的。

当 SCM 确定“必须要启动一个新的服务器进程”的时候，它企图获取一个 NT SECURITY\_DESCRIPTOR，该 SECURITY\_DESCRIPTOR 描述了哪些用户可以启动服务器进程。SCM 首先检查该类的 AppID，以求获得一个显式的激发许可以设置。这项设置是按照自相关的 NT 安全描述符经过序列化之后的形式，被保存在 AppID 的 LaunchPermission 名字值下面：

```
[HKCR\AppID\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}]
LaunchPermission=<serialized NT security descriptor>
```

如果这个名字值没有出现，那么 SCM 企图从下面的名字值中，读取机器范围内的激发许可以设置：

```
[HKEY_LOCAL_MACHINE\Software\Microsoft~OLE]
DefaultLaunchPermission=<serialized NT security descriptor>
```

这两个设置都可以用 DCOMCNFG.EXE 来修改。如果这两个注册表项都没有被找到的话，COM 将拒绝对每个人的激发请求。一旦 SECURITY\_DESCRIPTOR 已经被找到，SCM 就会检查发出激活请求的调用者（正式名称为激活者[activator]）的安全标识符，以及 SECURITY\_DESCRIPTOR 的任意访问控制列表（DACL，Discretionary Access Control List），并决定激活者是否有启动服务器的许可权。如果激活者没被赋予激发许可，那么激活调用失败，错误值 HRESULT 为 E\_ACCESSDENIED，而且不会启动进程。否则，SCM 启动服务器进程，并继续处理激活请求。

激发许可以解决了“哪些用户能够（或者不能够）在激活时刻启动服务器进程”。SCM 总是根据注册表中保存的信息来执行这项检查。访问许可以解决了“哪些用户可以与服务器进程的对象进行实际的通信”。这项检查是每当服务器接收到来自客户的连接请求时，由 COM 库来执行的。为了控制一个进程的访问许可以设置，开发人员也可以使

用 CoInitializeSecurity API 函数。

前面提到过，如果一个进程没有显式地调用 CoInitializeSecurity，那么它就自动使用保存在应用的 AppID 注册表键中的访问控制列表：

```
[HKCR\AppBar\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}]
AccessPermission=<serialized NT security descriptor>
```

正如前面所讲述的，如果此注册表键没有出现的话，COM 会查找机器范围内的缺省设置，如果这也没有找到的话，那么它就会创建一个新的访问控制列表，其中只包含服务器进程的安全个体和内置的 SYSTEM 帐号。

如果一个应用显式地调用了 CoInitializeSecurity，那么它可以手工地控制哪些调用者有权访问当前进程引出的对象。缺省情况下，CoInitializeSecurity 的第一个参数接受一个指向 NT SECURITY\_DESCRIPTOR 的指针，如果调用者在这个参数中传递一个 null 指针，那么 COM 将不对入调用执行任何访问检查。这就使得来自自己认证安全个体的任何调用都可以被接受。如果客户和服务器都指定了 RPC\_C\_AUTHN\_LEVEL\_NONE，那么 COM 就会允许来自任何人的调用，而不管他们是否已经被认证。如果调用者提供了一个有效的安全描述符指针，COM 就会用安全描述符的 DACL 来决定“哪些调用者有权访问进程的对象”。SDK 头文件定义了一个特殊的权限标志 (COM\_RIGHTS\_EXECUTE)，可以在创建 DACL 的时候用它来显式地赋予或者拒绝用户连接到进程的对象上。

虽然用 Win32 API 来创建一个 SECURITY\_DESCRIPTOR 并传给 CoInitializeSecurity，这样做是完全合法的，但是这并不是控制访问进程对象的首选办法，主要是由于原始的 Win32 安全性 API 的神秘本质。为了简化 COM 访问控制的程序设计，Windows NT 4.0 Service Pack 2 中的 COM 实现版本允许实现者指定一个 COM 对象，在建立新连接的时候，COM 将用它来执行访问检查。在 CoInitializeSecurity 时刻，这个对象被注册到 COM 库中，并且它必须实现 IAccessControl 控制：

```
[ object,uuid(EEDD23E0-8410-11CE-A1C3-08002B2B8D8F) ]
interface IAccessControl : IUnknown {
    // 增加一列用户的允许访问权限
    HRESULT GrantAccessRights(
        [in] PACTRL_ACCESSW           pAccessList
    );
    // 显示设定一列用户的访问权限
    HRESULT SetAccessRights(
        [in] PACTRL_ACCESSW   pAccessList           // 用户+权限
    );
    // 设置描述符的 owner/group ID
    HRESULT SetOwner(
        [in] PTRUSTEEW      pOwner,                // owner ID
        [in] PTRUSTEEW      pGroup                 // group ID
    );
    // 删除一列用户的访问权限
```

```

    HRESULT RevokeAccessRights(
        [in] LPWSTR      lpProperty,           // 未用
        [in] ULONG       cTrustees,            // 多少个用户
        [in, size_is(cTrustees)] TRUSTEEW prgTrustees[] // 用户
    );
    // 获取一列用户及其权限
    HRESULT GetAllAccessRights(
        [in] LPWSTR      lpProperty,           // 未用
        [out] PACTRL_ACCESSSW *ppAccessList,   // 用户+权限
        [out] PTRUSTEEW   *ppOwner,             // owner ID
        [out] PTRUSTEEW   *ppGroup              // group ID
    );
    // 由 COM 调用允许/禁止访问一个对象
    HRESULT IsAccessAllowed(
        [in] PTRUSTEEW   pTrustee,             // 调用者 ID
        [in] LPWSTR      lpProperty,           // 未用
        [in] ACCESS_RIGHTS Rights,            // COM_RIGHTS_EXECUTE
        [out] BOOL       *pbAllowed           // 是或否!
    );
}

```

设计这个接口的目的是为了允许开发人员根据静态的数据表格（把安全个体名字映射到访问控制权限）创建访问控制对象。这个接口以 Windows NT 4.0 中新的“基于被信任方（trustee-based）”的安全 API 为基础。此 API 用到的基本数据类型为 TRUSTEE:

```

typedef struct _TRUSTEE_W {
    struct _TRUSTEE_W      *pMultipleTrustee;
    MULTIPLE_TRUSTEE_OPERATION MultipleTrusteeOperation;
    TRUSTEE_FORM          TrusteeForm;
    TRUSTEE_TYPE           TrusteeType;
    [switch_is(TrusteeForm)]
    union {
        [case(TRUSTEE_ZS_NAME)]
        LPWSTR               ptstrName;
        [case(TRUSTEE_IS_SID)]
        SID                  *pSid;
    };
} TRUSTEE_W, *PTRUSTEE_W, TRUSTEEW, *PTRUSTEEW;

```

这个数据类型可用来描述一个安全个体。前两个参数（pMultipleTrustee 和 MultipleTrusteeOperation）使得调用者可以区分物理登录或者模仿企图。第五个参数（pstrName/pSid）或者包含 NT 安全标识符，或者包含文本形式的帐号名，并且第三个参数（TrusteeForm）指示使用哪个联合成员。第四个参数（TrusteeType）指示这里指定的安全个体是一个用户（user），还是一个组帐户（group account）。

为了把被信任方（trustee）与许可权限联系起来，Win32 API 也提供了 ACTRL\_ACCESS\_ENTRY 数据类型:

```

typedef struct _ACTRL_ACCESS_ENTRYW {
    TRUSTEE_W      Trustee;           // 谁?
    ULONG          fAccessFlags;      // 允许还是禁止?
    ACCESS_RIGHTS  Access;           // 哪些权限?
    ACCESS_RIGHTS  ProvSpecificAccess; // COM 未用
    INHERIT_FLAGS  Inheritance;      // COM 未用
    LPWSTR         lpInheritProperty; // COM 未用
} ACTRL_ACCESS_ENTRYW, *PACTRL_ACCESS_ENTRYW;

```

以及一个用来建立起“被信任方/许可”对照表模型的数据类型:

```

typedef struct _ACTRL_ACCESS_ENTRY_LISTW {
    ULONG          cEntries;
    [size_is(cEntries)] ACTRL_ACCESS_ENTRYW  *pAccessList;
} ACTRL_ACCESS_ENTRY_LISTW, *PACTRL_ACCESS_ENTRY_LISTW;

```

最后, Win32 还提供了其他两个数据类型, 它们把访问入口表与名字属性联系起来, 其定义如下:

```

typedef struct _ACTRL_PROPERTY_ENTRYW {
    LPWSTR          lpProperty;        // COM 未用
    ACTRLACCESS_ENTRY_LISTW *pAccessEntryList;
    ULONG           fList_aggs;        // COM 未用
} ACTRL_PROPERTY_ENTRYW, *PACTRL_PROPERTY_ENTRYW;
typedef struct __ACTRL_ALISTW {
    ULONG          cEntries;
    [size_is(cEntries)]
    ACTRL_PROPERTY_ENTRYW *pPropertyAccessList;
} ACTRL_ACCESSW, *PACTRL_ACCESSW;

```

虽然目前 COM 并没有使用这两个数据类型所暗示的基于属性的 (per-property) 能力特征, 但是 IAccessControl 仍然使用 ACTRL\_ACCESSW 数据类型来表示访问控制表。这是因为这个接口也被广泛用于 Windows NT 5.0 (即 Windows 2000) 的目录服务, 在目录服务中, 基于属性的访问控制是必需的。

COM 提供了一个 IAccessControl 实现 (CLSID\_DCOMAccessControl), 调用者可以利用 NT 4.0 的访问控制数据类型, 来指定显式的帐号名称和访问权限。<sup>13</sup> 下面的代码片段使用这个类来创建一个访问控制对象, 它允许内置帐号 SYSTEM 和 Sales\Managers 组中用户的访问, 但是禁止 Sales\Bob 用户的访问。代码如下:

```

HRESULT CreateAccessControl(IAccessControl * &rpac) {
    rpac = 0;
}

```

<sup>13</sup> 这个类也实现了 IPersistStream 接口, SCM 可以识别出它被序列化之后的格式, 因此它可以在自注册的时候被写到 AccessPermission 注册表项中。

```

// 创建缺省访问控制对象
HRESULT hr = CoCreateInstance(CLSID_DCOMAccessControl,
    0, CLSCTX_ALL, IID_IAccessControl,
    (void**)&rpac);

if (SUCCEEDED(hr)) {
// 用 NT4 安全数据类型创建用户/权限列表
    ACTRL_ACCESS_ENTRYW rgaae[] = {
        { {0, NO_MULTIPLE_TRUSTEE, TRUSTEE_IS_NAME,
            TRUSTEE_IS_USER, L"Sales\\Bob" },
          ACTRL_ACCESS_DENIED, COM_RIGHTS_EXECUTE, 0,
          NO_INHERITANCE, 0 },
        { {0, NO_MULTIPLE_TRUSTEE, TRUSTEE_IS_NAME,
            TRUSTEE_IS_GROUP, L"Sales\\Managers" },
          ACTRL_ACCESS_ALLOWED, COM_RIGHTS_EXECUTE, 0,
          NO_INHERITANCE, 0 },
        { {0, NO_MULTIPLE_TRUSTEE, TRUSTEE_IS_NAME,
            TRUSTEE_IS_USER, L"NT AUTHORITY\\SYSTEM" },
          ACTRL_ACCESS_ALLOWED, COM_RIGHTS_EXECUTE, 0,
          NO_INHERITANCE, 0 },
    };
    ACTRL_ACCESS_ENTRY_LISTW aael =
        { sizeof(rgaae)/sizeof(*rgaae), rgaae };
    ACTRL_PROPERTY_ENTRYW ape = { 0, &aael, 0 };
    ACTRL_ACCESSSW aa = { 1, &ape };

// 将列表提供给访问控制对象
    hr = rpac->SetAccessRights(&aa);
}
return hr;
}

```

有了这个对象之后，应用就可以把新建的访问控制对象与它的进程联系起来，代码如下：

```

IAccessControl *pac = 0;
HRESULT hr = CreateAccessControl(pac);
assert(SUCCEEDED(hr));
hr = CoInitializeSecurity(pac, -1, 0, 0,
    RPC_C_AUTHN_LEVEL_PKT, RPC_C_IMP_LEVEL_IDENTIFY, 0,
    EOAC_ACCESS_CONTROL,           // 使用 IAccessControl
    0);
assert(SUCCEEDED(hr));
pac->Release();                // COM 拥有引用直至最后一个 CoUninit

```

EOAC\_ACCESS\_CONTROL 标志表明 CoInitializeSecurity 的第一个参数是一个 IAccessControl 接口指针，而不是指向 NT SECURITY\_DESCRIPTOR 的指针。在每次接收到入连接请求时，COM 将使用这个对象的 IsAccessAllowed 方法来决定调用者是否有权访问进程的对象。有趣的是，虽然这段代码必须要在第一个“令人感兴趣的”COM 调

用”之前被执行到，但是调用 CoCreateInstance 以便获得 IAccessControl 缺省实现对象也是合法的，因为 COM 对这一动作并不感兴趣。

如果在进程启动的时候，它无法知道已授权用户的列表，那么我们可以注册一个自定义的 IAccessControl 实现，在自定义访问控制对象的 IsAccessAllowed 方法中执行某种运行时访问检查。因为 COM 本身只使用 IsAccessAllowed 方法，所以对于其他的 IsAccessAllowed 方法，自定义对象只要返回 E\_NOTIMPL 即可。下面是 IAccessControl 接口的一个简单实现，它只允许帐号名字中包含“x”的用户才可以访问进程的对象：

```

class XOnly : public IAccessControl {
// IUnknown 方法
    STDMETHODIMP QueryInterface(REFIID riid, void**ppv) {
        if (riid == IID_IAccessControl || riid == IID_IUnknown)
            *ppv = static_cast<IAccessControl*>(this);
        else
            return (*ppv = 0), E_NOINTERFACE;
        ((IUnknown*)*ppv)->AddRef();
        return S_OK;
    }
    STDMETHODIMP_(ULONG) AddRef(void) { return 2; }
    STDMETHODIMP_(ULONG) Release(void) { return 1; }
// IAccessControl 方法
    STDMETHODIMP GrantAccessRights(ACtrl_ACCESSW *)
    { return E_NOTIMPL; }
    STDMETHODIMP SetAccessRights(ACtrl_CCESSW *)
    { return E_NOTIMPL; }
    STDMETHODIMP SetOwner(PTRUSTEEW, PTRUSTEEW)
    { return E_NOTIMPL; }
    STDMETHODIMP RevokeAccessRights(LPWSTR, ULONG, TRUSTEEW [])
    { return E_NOTIMPL; }
    STDMETHODIMP GetAllAccessRights(LPWSTR,
        PACTRL_ACCESSW_LLOCATE_ALL_NODES *,
        PTRUSTEEW *, PTRUSTEEW *)
    { return E_NOTIMPL; }
// 唯一被 COM IAccessControl 调用的方法
    STDMETHODIMP IsAccessAllowed(
        PTRUSTEEW pTrustee, LPWSTR lpProperty,
        ACCESS_RIGHTS AccessRights, BOOL *pbIsAllowed) {
// 确保 Trustee 含有字符串
        if (pTrustee == 0
            || pTrustee->TrusteeForm != TRUSTEE_IS_NAME)
            return E_UNEXPECTED;
// 在当前寻找 X 或 x 以及授权/禁止
        *pbIsAllowed = wcsstr(pTrustee->ptstrName, L"x") != 0
            || wcsstr(pTrustee->ptstrName, L"X") != 0;
        return S_OK;
    };
}

```

如果上述 C++ 类的一个实例被注册到 CoInitializeSecurity 中：

```
XOnly xo; // 声明一个 C++ 类实例
Hr = CoInitializeSecurity(static_cast<IAccessControl *> (&xo),
    -1, 0, 0, RPC_C_AUTHN_LEVEL_PKT,
    RPC_C_IMP_LEVEL_IDENTIFY, 0,
    EOAC_ACCESS_CONTROL, // 使用 IAccessControl
    0);
assert(SUCCEEDED(hr));
```

那么对于所有帐号名字中不包含“x”的用户，它们发出的调用请求都会被拒绝。因为被信任方的名字也包含域名作为前缀，所以这样简单的测试也适用于来自域名中包含“x”的用户帐号。虽然这样的访问测试未必有实用价值，但是它演示了如何在 CoInitializeSecurity 中使用自定义 IAccessControl 对象的技术。

## 6.8 令牌管理

在 Windows NT 下面，每个进程都有一个访问令牌，它代表某个安全个体的安全凭证。这个访问令牌在进程初始化的时候被创建，它包含各种与用户有关的信息，包括用户的 NT 安全标识符（SID）、用户所属的组列表，以及用户的权限列表（比如用户是否可以关闭机器、是否可以改变系统时钟等）。当一个进程企图访问被保护的内核资源（比如文件、注册表键、信号量等）的时候，NT 安全引用监视器（NT Security Reference Monitor）以调用者的令牌作为审核和访问控制的对象。

当一个 ORPC 请求消息到达某个进程的时候，COM 负责安排对应的方法调用，或者在 RPC 线程中执行（如果是基于 MTA 的对象），或者在用户创建的线程中执行（如果是基于 STA 的对象）。无论哪种情况下，方法运行的时候都会使用进程的访问令牌。一般来讲，这正是我们所期望的，因为这样可使对象的实现者能够预知他们的对象将具有什么样的权限和优先级别，而无须考虑是哪个用户发出这个请求的。然而，方法在执行的时候，如果能使用发出该方法调用的客户的安全凭证，偶尔也是非常有用的，这样做可以限制或者增强对象的正常权限和优先级。为了支持这种风格的程序设计，Windows NT 允许访问令牌可以被赋给单独的线程。当一个线程有了自己的令牌的时候，安全引用监视器不再使用进程的令牌，而使用该线程的令牌来执行审核和访问控制。虽然在程序中创建令牌并分配给线程也是有可能的，但是 COM 提供了一种更为直接的机制，可根据当前线程正在处理的 ORPC 请求来创建一个令牌。通过调用环境对象，对象实现者可以使用这种机制。

前面曾经提到过，当 ORPC 请求被分发给某个接口存根的时候，调用环境对象与一个线程联系在一起。对象实现者通过 CoGetCallContext API 函数访问调用环境。调用环

境对象实现了 **I ServerSecurity** 接口：

```
[ local, object, uuid(0000013E-0000-0000-C000-00000000046) ]
interface I ServerSecurity : IUnknown {
// 获取调用者安全设置
    HRESULT QueryBlanket(
        [out] DWORD      *pAuthnSvc,           // 认证包
        [out] DWORD      *pAuthzSvc,          // 授权包
        [out] OLECHAR   **pServerName,        // 服务器主名字
        [out] DWORD      *pAuthnLevel,         // 认证级别
        [out] DWORD      *pImpLevel,          // 模仿级别
        [out] void       **pPrivs,             // 客户方主名字
        [out] DWORD      *pCaps               // EOAC 标志
    );
// 开始以调用者凭证运行
    HRESULT ImpersonateClient(void);
// 停止运行
    HRESULT RevertToSelf(void);
// 测试模仿
    BOOL IsImpersonating(void);
}
```

本章上一节已经讨论了 **QueryBlanket** 方法。剩下的三个方法被用于在方法执行过程中管理线程令牌。**ImpersonateClient** 方法根据客户的安全凭证，创建一个访问令牌，并把令牌分配给当前线程。一旦 **I ServerSecurity::ImpersonateClient** 返回，以后所有对 OS 资源的访问都将根据客户的安全凭证作出允许或者拒绝的决定，而不再根据对象的安全凭证作出决定。**RevertToSelf** 方法使当前线程恢复到使用进程访问令牌的状态。如果当前方法调用返回时它正处于模仿的状态，COM 将隐式地使线程恢复到使用进程令牌的状态。最后，**I ServerSecurity::IsImpersonating** 方法指明了当前线程正在使用客户的安全凭证，还是对像所在进程的令牌。如同 **QueryBlanket** 方法一样，这三个 **I ServerSecurity** 方法的前两个也有对应的便利包装函数，它们在内部先调用 **CoGetCallContext**，然后调用对应的方法，函数的原型如下：

```
HRESULT CoImpersonateClient(void);
HRESULT CoRevertToSelf(void);
```

一般来说，如果有多个 **I ServerSecurity** 方法要被用到的话，那么我们可以先调用 **CoGetCallContext**，然后使用结果得到的 **I ServerSecurity** 接口来调用每个方法，这样做效率略高一些。

下面的代码片断演示了如何使用调用环境对象，让方法的一部分代码在客户的安全凭证下运行：

```
STDMETHODIMP MyClass::ReadWrite(DWORD dwNew, DWORD *pdwOld) {
// 用服务器令牌执行，让所有人可以读取值
```

```

    ULONG cb;
    HANDLE hfile = CreateFile("C:\\\\file1.bin", GENERIC_READ,
                               0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if (hfile == INVALID_HANDLE_VALUE)
        return MAKE_HRESULT(SEVERITY_ERROR, FACILITY_WIN32,
                            GetLastError());
    ReadFile(hfile, pdwold, sizeof(DWORD), &cb, 0);
    CloseHandle(hfile);

    // 获取调用环境对象
    IServerSecurity *pss = 0;
    HRESULT hr = CoGetCallContext(IID_IServerSecurity,
                                  (void**)&pss);
    if (FAILED(hr)) return hr;

    // 使用调用者凭证设置线程令牌
    hr = pss->ImpersonateClient();
    assert(SUCCEEDED(hr));
    // 用客户令牌执行，仅让能够写文件的用户改变值
    hfile = CreateFile("C:\\\\file2.bin",
                       GENERIC_READ|GENERIC_WRITE, 0, 0,
                       OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    if (hfile == INVALID_HANDLE_VALUE)
        hr = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_WIN32,
                           GetLastError());
    else {
        WriteFile(hfile, &dwNew, sizeof(DWORD), &cb, 0);
        CloseHandle(hfile);
    }

    // 恢复线程为使用进程级令牌
    pss->RevertToSelf();
    // 释放调用环境
    pss->Release();
    return hr;
}

```

注意，第一个 `CreateFile` 调用是用对象进程的安全凭证来执行的，而第二个 `CreateFile` 调用是在客户的安全凭证下执行的。如果客户对于第二个底层文件具有读/写访问权限，那么即使对象的进程对该文件并没有正常的访问权限，第二个 `CreateFile` 也会成功。

很重要并且值得指出的一点是，当 `IIServerSecurity::ImpersonateClient` 成功的时候，除非是灾难性失败，否则对象的客户控制着结果令牌所允许的模仿级别。每个接口代理都有一个模仿级别，并且必须为四个常数之一 (`RPC_C_IMP_LEVEL_ANONYMOUS`、`RPC_C_IMP_LEVEL_IDENTIFY`、`RPC_C_IMP_LEVEL_IMPERSONATE` 或者 `RPC_C_IMP_LEVEL_DELEGATE`)。在散集的时候，COM 把这个级别设置为客户在 `CoInitializeSecurity`

调用中指定的值；然而，客户也可以利用 `IClientSecurity:: SetBlanket` 方法手工改写掉这个设置。当对象调用 `IServerSecurity::ImpersonateClient` 的时候，新的令牌被限制在发出这个调用的接口代理所指定的级别上。这意味着，如果客户只指定 `RPC_C_IMP_LEVEL_IDENTIFY`，那么对象在用客户的安全凭证执行方法的时候，不可能访问内核资源。然而，对象可以使用 Win32 API 函数 `OpenThreadToken/GetTokenInformation`，从模仿令牌中读取有关客户的信息（比如安全 ID、组成员关系）。另一点很重要的是，除非客户指定了 `RPC_C_IMP_LEVEL_DELEGATE`，否则对象不能使用客户的安全凭证访问任何受保护的远程资源。这包括打开远程文件系统中的文件，以及向远程对象发出经过认证的 COM 调用。不幸的是，NTLM 认证协议不支持 `RPC_C_IMP_LEVEL_DELEGATE`，所以在 Windows NT 4.0 下面，委托模仿是不可能的。

前面的讨论强调“正常情况下，对象的方法以对象进程的令牌来执行”。有一点我们一直没有讨论，那就是如何控制“服务器进程的初始令牌是用哪个安全个体的身份来创建的”。当 SCM 启动服务器进程的时候，它根据 AppID 中 RunAs 名字值的配置信息，为新的服务器进程分配一个令牌。如果 AppID 并没有 RunAs 值，那么它就假定此服务器还没有被配置成“可被分布式访问的”。为了避免这种类型的服务器进程把安全漏洞引入到系统中，SCM 使用发出激活请求的安全个体身份启动这些进程。这种激活类型通常被称为“以激活者身份（As Activator）”激活，因为服务器进程与发出激活请求的用户运行在同一个安全个体身份下。以激活者身份激活的用意是为了使传统遗留下来的服务器也能支持远程激活，但是它有一些缺陷。第一，为了保持“以激活者身份激活”的语义，COM 将为每个请求激活的用户帐号启动一个单独的服务器进程，而不管在 `CoRegisterClassObject` 中是否使用了 `REGCLS_MULTIPLEUSE`。这样就严重影响了系统的可伸缩性，而且不可能把类的所有实例都保持在同一个进程中。第二，每个服务器进程启动的时候，都被限制到 `RPC_C_IMP_LEVEL_IMPERSONATE`，这就意味着服务器进程不可能访问任何远程资源或者对象。<sup>14</sup>

理想情况下，服务器进程被配置成“在某个特定的安全个体身份下运行”。这可以由 AppID 的 RunAs 名字值中的帐号名称来控制：

```
[HKCR\AppID\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}]
RunAs="DomainX\UserY"
```

如果这个名字值出现了，那么 SCM 将使用指定的帐号名字创建一个新的登录令牌，然后把令牌赋给服务器进程。为了使这项工作能顺利进程，必须满足两个条件。第一，对应的口令必须要被保存在注册表中特殊的位置上，可以看作是 LSA（Local Security Authority，局部安全控制中心）的一个秘密。第二，被指定的用户帐号必须具有“`Logon as a batch job`”特权。在设置 RunAs 值的时候，`DCOMCNFG.EXE` 工具可以确保这两个

<sup>14</sup> 当然，这也暗示了调用者在发出激活请求的时候，必须至少指定 `RPC_C_IMP_LEVEL_IMPERSONATE` 模仿级别（可以通过 `CoInitializeSecurity` 隐式指定，或者使用 `COAUTHINFO` 结构显式指定）。

条件都可以满足。<sup>15</sup>

为了避免受到恶意程序的类的欺骗，CoRegisterClassObject 检查被注册类的 AppID。如果 AppID 具有 RunAs 设置，COM 要确保调用者的安全个体与注册表中保存的安全个体名一致。如果调用者不是该类的 AppID 中指定的 RunAs 帐号，那么 CoRegisterClassObject 将会失败，并返回一个可识别的 HRESULT CO\_E\_WRONG\_SERVER\_IDENTITY。因为 COM 的配置信息被保存在注册表中受保护的区域，所以只有特权用户才可以修改哪个类以哪个用户的身份运行。

很重要并且值得指出的一点是，当某个 AppID 有一个显式的 RunAs 用户帐号时，SCM 将总是在它自己特有的窗口站（window station）中启动服务器进程。<sup>16</sup> 这意味着服务器进程不能很容易地创建“对于当前交互用户可见的”窗口，服务器进程也不能很容易地读取来自键盘、鼠标或者剪贴板的输入。一般来说，这种保护是有好处的，因为这样可以使 COM 服务器不会干扰当前登录到机器上的用户的操作。<sup>17</sup> 不幸的是，有时候服务器进程仍然需要与当前登录用户进行通信。要做到这一点，一项技术是使用显式的 Win32 API 函数来管理窗口站和桌面，让一个线程临时在当前交互桌面中执行。当线程在当前交互桌面中执行时，它创建的所有窗口对于当前的交互用户来说都是可见的，而且该线程也能够接收到来自键盘和鼠标的硬件消息。另一项技术是，如果服务器所需要的仅仅是让用户给出一个简单的 yes/no 回答，那么 Win32 API 函数 MessageBox 支持 MB\_SERVICE\_NOTIFICATION，它可使消息框出现在当前交互桌面上，而无须进一步编写代码。

如果还需进一步与交互用户进行交互操作，则使用 Win32 窗口站 API 可能是非常麻烦的。更好的办法是把用户界面组件部分剥离到另一个进程外服务器中，让这个进程运行在另一个窗口站中（不同于主对象层次结构的窗口站）。为了强迫让一个具有用户界面的服务器进程运行在交互用户的窗口站中，COM 可以识别另一个 RunAs 值“Interactive User”：

```
[HKCR\AppID\{27EE6A4D-DF65-11d0-8C5F-0080C73925BA}]
RunAs="Interactive User"
```

当使用这个值的时候，COM 在当前登录用户的窗口站中启动新的服务器进程。为了得到新服务器进程的安全凭证，COM 在创建新的服务器进程的时候，只是简单地复

<sup>15</sup> 在自注册的时候执行这两个操作也是有可能的。请参考 Win32 SDK 中一个极好的例子程序 DCOMPPerm（作者 Miker Nelson）。

<sup>16</sup> 如果这个 AppID 并没有 RunAs 设置（也就是说该类被配置成“以激活者身份激活”），那么 SCM 在激活者的窗口站中启动服务器进程（如果激活者是一个远程客户，那么 SCM 在新的窗口站中启动服务器进程）。这意味着，只有当激活者碰巧是当前交互用户的时候，服务器才可以与交互用户进行交互。

<sup>17</sup> 这种隔离并非不需要付出性能上的代价。SCM 用 RunAs 帐号启动的每一个服务器进程都需要消耗一个窗口站和桌面。默认情况下，Windows NT 4.0 被配置成只允许约 14 个桌面总数。这意味着，在默认配置情况下只有 14（或者更少）个 RunAs 服务器可以同时运行。Microsoft 知识库（Knowledge Base）中文章 Q171890 解释了如何改变这项限制，使它达到一个比较合理的数字。

制了当前交互会话的令牌。这意味着我们不需要把口令保存在注册表中。不幸的是，这种激活模式也不是没有缺陷的。首先，如果一个激活到达某台远程服务器，但是当前并没有用户登录到该服务器上，那么激活调用将会失败，返回 E\_ACCESSDENIED。其次，如果交互用户注销当前会话时，服务器进程与客户的连接仍然活着，那么服务器进程将会提前终止，并且强行断开它与所有代理之间的连接。最后，在激活的时候，到底哪个用户已经登录到系统中了，这往往是不可预测的，从而使得开发人员难以保证对象有足够的权限去访问所要求的资源。这些限制使得这种激活模式仅适用于简单的用户界面组件。<sup>18</sup>

在控制服务器进程的令牌和窗口站方面，一个很有趣的变化情况与 NT 服务（NT Service）有关。前面曾经提到过，如果一个 CLSID 键中出现了 LocalService 名字值，这将使 SCM 利用 NT 服务控制管理器来启动一个服务器进程，而不是使用 CreateProcess 或者 CreateProcessAsUser。当以 NT 服务的方式来启动服务器进程的时候，COM 对于“该进程将以哪个安全个体身份来启动”并没有控制权，因为对于被启动的 NT 服务来说，在它的配置中这是固定的。然而，在这种情况下，COM 会忽略掉 RunAs 名字值，以保证任何进程不能够随便调用 CoRegisterClassObject；一旦出现了 LocalService 名字值，这就要求调用者以 NT 服务来运行。如果服务本身被配置成以内置帐号 SYSTEM 的身份来启动，那么服务器进程或者运行在交互窗口站中，或者运行在预先定义的、专门供这些“以 SYSTEM 身份运行的 NT 服务”共享的窗口站中，到底运行在哪个窗口站中要取决于该 NT 服务是如何被配置的。如果 NT 服务被配置成“以某个特定的帐号来运行”，那么 NT 服务控制管理器将总是在新的窗口站（特定于这个服务器进程）中启动这个服务。

以 NT 服务的形式来实现 COM 服务器，这种做法的一个常见的动机是：只有 NT 服务才可以使用内置帐号 SYSTEM 来执行。这个帐号对于本地资源（比如本地的文件和注册表键）往往有更大的访问权限。而且，这个帐号通常是唯一可以当作“可信任的计算基础”的帐号、唯一可以使用底层安全服务的帐号（如果普通用户帐号也可以访问底层安全服务的话，那就太危险了）。不幸的是，尽管 SYSTEM 帐号在本地系统上无所不能，但是对于受保护的远程资源（包括远程文件系统和远程 COM 对象），它也是无能为力的。因此，对于创建分布式系统而言，SYSTEM 帐号并不是非常有用。不管一个服务器被配置成 NT 服务，还是一个传统的 Win32 进程，为每一个“希望拥有完全网络安全凭证”的 COM 应用创建一个单独的用户帐号，这是比较实用的做法。

<sup>18</sup> 然而，在调试服务器进程的初始化过程中，只有用这种模式才可以避免 RPC\_E\_WRONG\_SERVER\_IDENTITY 错误。

## 6.9 我们走到哪儿了？

本章讨论了与“把 COM 类分离到不同服务器进程中”有关的事项。COM 可以根据激活请求来启动服务器进程。这些服务器进程必须用 `CoRegisterClassObject` 把自己注册到 COM 库中，以便使它们的类对象可被外部客户使用。COM 的安全性结构与 OS 的安全模型紧密集成在一起，它以三个不同的概念为基础。认证解决了客户与对象之间传送的 ORPC 消息的完整性和可信性。访问控制解决了哪些安全个体可以访问给定进程所引出的对象。令牌管理解决了“启动服务器进程”和“执行对象的方法”时使用什么样的安全凭证。

# 第 7 章

## 杂 项

```
Ichapter *pc = 0;
HRESULT hr = CoGetObject(OLESTR("Chapter:7"), 0;
                         IID_IChapter, (void**)&pc);
if (SUCCEEDED(hr)) {
    Hr = pc->IncludeAllTopicsNotCoveredYet();
    pc->Release();
}
```

本书作者, 1997

上一章展示了 COM 编程模型和远程结构的基础。尽管本书已经讨论了各种各样的 COM 接口和技术，但是还有许多话题不能够明确地归到前面任何特定的章节中，然而为了保持本书的完整性，仍有必要讨论这些话题。前面的每一章都已经有了合理的线索，或者说每一章的篇幅已经非常大了，所以我不再把这些待讨论的话题硬塞到前面的章节中，而是保留一章专门讨论这些“小一点”的话题。除了关于指针、内存管理和数组的介绍之外，其他的话题对于使用 COM 来建立有效的分布式系统并非至关重要。有了这样的认识之后，请你轻松一下，然后让你的眼睛快速而简略地浏览本章的内容，直到本章结束。

## 7.1 指针基础

与 DCE 一样，COM 的根源也是 C 程序设计语言。尽管很少有开发人员用 C 开发或者使用 COM 组件，但是 COM 确实继承了 C 的语法作为它的接口定义语言。关于接口的设计和使用，最容易混淆的一点是指针的管理。考虑下面非常简单的 IDL 方法定义：

```
HRESULT f([in] const short *ps);
```

如果调用者用下面的方式来调用这个方法：

```
short s = 10;
HRESULT hr = p->f(&s);
```

则数值 10 必须被发送给对象。如果这个方法跨越套间边界被远程调用，那么接口代理有责任把指针所指的内容(10)解析(dereference)出来，然后把数值 10 转送到 ORPC 请求消息中。

下面的客户代码是完全合法的 C 代码，但是比较有趣：

```
HRESULT hr = p->f(0); // 传递一个空指针
```

如果调用线程在对象的套间中执行，那么调用过程不会涉及到代理，空(null)指针被直接传递给对象。但是如果对象驻留在不同的套间中，客户与对象之间通过代理来传递调用，那又会怎么样呢？接口代理应该传递什么信息来表达这是一个 null 指针呢？而且，这是否意味着接口代理和存根必须要检查每个指针是否为 null 呢？这说明，有些情况下指针绝对不能为 null，而其他一些情况下 null 指针作为一个特殊的值非常有用，但是“null 指针被传给接口代理”的事实必须要由接口存根如实地复制到对象的套间中。

为了解决这些不同的要求，COM 允许接口设计者显式地指出每个指针参数的确切语义。为了指明一个指针永远也不能为 null，接口设计者可以使用[ref]属性：

```
HRESULT g([in,ref] short *ps); // ps 可以是一个空指针
```

使用[ref]属性的指针被称为引用指针(reference pointer)。给出了上面的 IDL 定义之后，下面的客户代码：

```
HRESULT hr = p->g(0); // 危险：传递空的[ref]指针
```

如果 p 指向一个接口代理，那么此语句为非法，并且接口代理将会检测到 null 指针，然后给调用者返回一个列集(marshaling)错误，而不会把方法调用传递给实际的远程对象。相反，如果 null 指针是一个合法的参数值，那么 IDL 定义应该使用[unique]属性：

```
HRESULT h([in, unique] short *ps); // ps 可以是空指针
```

使用[unique]属性的指针被称为单值指针（unique pointer）。有了上面的 IDL 定义之后，下面的客户代码是完全合法的：

```
HRESULT hr = p->h(0); // 放心，传递空的[unique]指针没问题。
```

这意味着接口代理在解析指针所指内容之前，必须显式地测试指针。更重要的是，这意味着接口代理不能够只是把解析出来的内容写到 ORPC 请求之中，而是要写入更多的内容。它必须要写入一个标记，来指示传递的参数是否为 null 指针。这使每个指针在 ORPC 消息中增加四个字节。在大多数应用中，附加的四个字节和检测 null 指针所需要的 CPU 周期<sup>1</sup>，相对于“能够使用 null 指针作为参数值”所获得的好处而言，这点代价几乎可以忽略。

在整个结构中，[ref]和[unique]之间的性能差异非常微小。然而，与指针有关的另一个问题还没有讨论。考虑下面的 IDL 片断：

```
HRESULT j([in] short *ps1, [in] short *ps2);
```

给出了这样的 IDL 定义之后，考虑下面的客户代码：

```
short x = 100;
HRESULT hr = p->j(&x, &x); // 注意：相同指针传递两次
```

问题是，当出现了重复指针的时候，接口代理该怎么办？如果接口指针什么也不做，那么数值 100 将在 ORPC 请求中被传输两次：一次是为了\*ps1，另一次是为了\*ps2。这意味着代理发送同样的信息两次，既浪费了网络带宽，又影响了性能。在这个例子中，数值 100 所消耗的字节数并不要紧，但是如果 ps1 和 ps2 指向巨大的数据结构，那么重复传输可能会带来严重的性能影响。不检测重复指针的另一个负面影响是，接口存根将把参数值散集（unmarshal）到不同的内存区域中。如果方法的语义依赖于两个指针是否相等而有所不同：

```
STDMETHODIMP MyClass::j(short *ps1, short *ps2) {
    if (ps1 == ps2)
        return this->OneKindOfBehavior(ps1);
    else
        return this->AnotherKindOfBehavior(ps1, ps2);
}
```

那么接口列集器将违反接口的语义约定，从而破坏了 COM 的远程透明性。

<sup>1</sup> MIDL 产生的接口代理和存根并不检查[ref]指针是否为 null。相反，它们盲目地把指针所指内容解析出来，并允许产生“访问违例(access violation)”。因为 MIDL 产生的列集器总是在一个异常处理函数的内部执行，所以“访问违例”可以在列集器的内部被捕捉到，并且被转译为列集错误，作为方法的 HRESULT 值被返回。

指针属性[ref]和[unique]都隐含了“指针所指向的内存并没有成为这个方法调用中另一个指针的别名”和“接口列集器不应该执行重复性测试”。为了表明一个指针所指向的内存可能是同一方法调用中另一个指针所指的内存，IDL 设计者应该使用[ptr]属性：

```
HRESULT k([in, ptr] short *ps1, [in, ptr] short *ps2);
```

使用[ptr]属性的指针被称为全指针（full pointers），因为它们最接近于 C 程序设计语言中指针的完全语义。有了这个 IDL 定义，下面的客户代码：

```
short x = 100;
HRESULT hr = p->k(&x, &x); // 注意：相同指针传递两次
```

将导致数值 100 只被传输一次，因为 ps1 参数中的[ptr]属性告诉接口列集器，应该针对所有其他的[ptr]作重复性检查。因为 ps2 参数也使用了[ptr]属性，所以接口列集器将检测重复的指针值，<sup>2</sup> 并且只解析和传输一个指针的值。接口存根也注意到，被传输过来的值既是 ps1 参数，也是 ps2 参数，这使得方法 k 在两个参数中接收到同样的指针。

虽然全指针可以解决各种问题，并且在特定的情况下也非常有用，但是它们并不是 COM 中首选的指针语义。这是因为多数情况下，接口设计者预先知道不会有重复指针被传送给方法。而且，虽然全指针会导致更小的 ORPC 消息（当确实发生重复指针的时候），但是运行库为找到重复指针而付出的代价也不容忽视（特别是当每个方法的指针数目较大的时候）。如果接口设计者确定不会发生重复指针，那么最好是指明这一点：使用单值指针或者引用指针。

## 7.2 指针和内存

到目前为止本章所给出的接口都非常简单，并且只使用基本的数据类型。很可能问题更多的一个方面是，在使用复杂数据类型的时候，方法参数的内存如何管理。考虑下面的 IDL 函数原型：

```
HRESULT f([out] short *ps);
```

有了这个原型之后，下面的 C 代码是完全合法的：

```
short s;
HRESULT hr = p->f(&s); // s 现在包含 f 的内容
```

对于像这样的一个简单函数，到底如何管理内存应该是非常清楚的。然而，初学者

<sup>2</sup> 接口列集器检测重复的指针值( $ps1 == ps2$ )，而不是解析出来的内容是否重复( $*ps1 == *ps2$ )；然而，前者也隐含了后者。

(甚至不是初学者) 往往会写出类似这样的代码:

```
short *ps; // 函数接收一个短整型*, 因此……
HRESULT hr = p->f(ps);
```

当我们考虑下面合法的函数实现代码时:

```
STDMETHODIMP MyClass::f(short *ps) {
    static short n = 0;
    *ps = n++;
    return S_OK;
}
```

很显然, 调用者应该负责分配短整型内存, 并把指向该内存的引用作为参数传给函数。注意, 从上面显示的实现中我们可以看出, 对于函数来说, 这块内存来自哪里(比如从堆中动态分配的、或者从栈中以 auto 变量方式声明的)并不重要, 只要实际的参数指向有效的内存即可。为了加强这项要求, COM 要求指针类型的[out]参数必须是引用指针(reference pointer)。

当我们使用自定义类型, 而不是简单的整数类型的时候, 事情就没有这么清楚了。考虑下面的 IDL:

```
typedef struct tagPoint {
    short x; short y;
} Point;
HRESULT g([out] Point *pPoint);
```

如同上一个例子一样, 正确的用法是: 调用者为 Point 值分配内存, 然后传递一个指向这块内存的引用:

```
Point pt;
HRESULT hr = p->g(&pt);
```

如果调用者传递一个无效的指针:

```
Point *ppt; // 随机未初始化指针
HRESULT hr = p->g(ppt); // 代理应该把 x & y 复制到哪里呢?
```

那么方法(或者接口代理)就得不到有效的内存可以写入 x 和 y 值。

当用户定义的类型变得再复杂一些时, 事情就更有趣了。考虑下面的 IDL:

```
[ uuid(E02E5345-1473-11d1-8C85-0080C73925BA), object ]
interface IDogManager : IUnknown {
    typedef struct tagHUMAN {
        long nHumanID;
```

```

} HUMAN;
typedef struct tagDOG {
    long nDogID;
    [unique] HUMAN *pOwner;
} DOG;
HRESULT GetFromPound([out] DOG **pDog);
HRESULT TakeToGroomer([in] const DOG *pDog);
HRESULT SendToVet([in, out] DOG *pDog);
}

```

这个接口的不同之处在于，现在调用者必须传递一个指向某块内存区域的指针，而该内存区域本身又包含一个指针。对于上面给出的方法定义，有人可能会争辩说下面的代码是正确的：

```

DOG fido;           // 参数是一个 DOG *, 所以调用者需要用一个 DOG
HUMAN dummy;        // DOG 指向一个主人，分配空间吗？
fido.pOwner = &dummy;
HRESULT hr = p->GetFromPound(&fido); // 这是正确的吗？

```

这段代码假定调用者负责分配 DOG 的内存，并以引用的方式传递给方法。从这个意义上讲，这段代码是正确的。然而，这段代码也假定了它有责任管理任何次级内存，这些次级内存有可能被更新之后的 DOG 对象所引用。很显然，这段代码不符合 COM 处理事情的风格。

COM 把方法调用中所涉及到的指针分为两类。一个方法中任何有名字的指针参数都被称为顶级指针（top-level pointer）。在顶级指针解析得到的数据中，其中隐含的任何次级指针被称为内嵌指针（embedded pointer）。在 GetFromPound 方法中，参数 pDog 被认为是一个顶级指针。次级指针 pDog->pOwner 被认为是一个内嵌指针。注意，DOG 结构定义用到了[unique]属性，这就显式地指定了结构成员 pOwner 的指针语义。如果指针的语义没有被显式指定的话，那么接口设计者也可以使用[pointer\_default]属性，为所有的内嵌指针指定一个默认指针属性（在整个接口范围内有效），示例如下：

```

[
    uuid(E02E5345-1473-11d1-8C85-0080C73925BA), object,
    pointer_default(ref) // [ref] 的缺省内嵌指针
]
interface IUseStructs : IUnknown {
    typedef struct tagNODE {
        long val;
        [unique] struct tagNODE *pNode; // 显式的 [unique]
    } NODE;
    typedef struct tagFOO {
        long val;
        long *pVal; // 隐式的 [ref]
    } FOO;
}

```

```
HRESULT Method([in] FOO *pFoo, [in, unique] NODE *pHead);
}
```

[pointer\_default]属性只适用于那些“指针语义没有被显式指定”的内嵌指针。在上面的接口定义中，唯一满足这一条件的只有FOO结构的pVal数据成员。NODE结构的pNode成员被显式指定为单值指针，所以不受[pointer\_default]设置的影响。方法参数pFoo和pHead也不受[pointer\_default]的影响，因为它们是顶级指针，其默认属性为[ref]，除非被显式指定为其他的属性（正如pHead的情形）。

COM中内嵌指针被给予不同的状态，其主要理由是它们具有不同的内存管理要求。关于内存管理，对于[in]参数来说，顶级指针和内嵌指针之间的差异并不重要，因为调用者要为方法提供所有的值，所以它必须为这些值分配相应的内存：

```
HUMAN bob = { 2231 };
DOG fido = { 12288, &bob }; // fido 由 bob 所拥有
HRESULT hr = p->TakeToGroomer(&fido); // 这是正确的!
```

然而，对于[out]参数和[in, out]参数的内存管理，顶级指针和内嵌指针之间的差异就非常重要。不论是[out]参数，还是[in, out]参数，顶级指针所指的内存是由调用者来管理的，如同[in]参数的情形一样。对于[out]或者[in, out]参数中出现的内嵌指针来说，内存是由被调用方（即方法）来管理的。这条规则的理由是“数据类型可以嵌套到任意深度”。例如，给定了下面的类型定义：

```
typedef struct tagNODE {
    short value;
    [unique] struct tagNODE *pNext;
} NODE;
```

调用者不可能预见到将有多少个子元素需要分配。然而，因为被调用方（方法）将为每个节点提供数据，所以它可以安全地为每个节点分配内存。

既然方法实现者必须分配一定的内存以便初始化任何内嵌指针，那么很自然就会引出这样一个问题：方法应该从哪里申请内存，才能让调用者知道如何释放这些内存（在用完这些内存之后）？答案是COM任务分配器（COM task allocator）。COM任务分配器是一个以进程为基础（per-process）的内存分配器，专门用于为[out]和[in, out]内嵌指针分配内存。使用COM任务分配器最简单的方法莫过于通过以下三个COM API函数：

```
void *CoTaskMemAlloc(DWORD cb); // 分配 cb 个字节
void CoTaskMemFree(void *pv); // 释放*pv 处的内存
void *CoTaskMemRealloc(void *pv, DWORD cb); // 增大/减少 *pv
```

这三个函数的语义等同于C运行库中对应的malloc、free和realloc函数。不同之处在于它们专门供[out]和[in, out]内嵌指针分配内存。另一个重要的区别是，C运行库函数

不能够用于“在一个模块中分配内存，而在另一个模块中释放它”。因为每一个 C 运行库的实现细节是私有的，不同编译器之间可以有所不同。如果让所有参与各方都同意使用同一个由 COM 提供的分配器，则虽然对象是在单独的 DLL 中实现的，但是它分配的内存可以由客户来释放。

为了理解在方法中被分配的内存是如何被使用的，考虑前面显示的 `GetFromPound` 方法：

```
HRESULT GetFromPound([out] DOG *pDog);
```

这里，`DOG` 对象的内存一定是由调用方来分配的（`pDog` 是一个顶级指针），而 `HUMAN` 对象的内存则必须由方法实现代码使用任务分配器来分配（`pDog->pOwner` 是 `[out]` 参数中的内嵌指针）。所以，这个方法的实现应该与下面的代码相似：

```
STDMETHODIMP GetFromPound(/* [out] */ DOG *pDog) {
    short did = LookupNewDogId;
    short hid = LookupHumanId(did);
    pDog->nDogID = did;
    // 为内嵌指针分配内存
    pDog->pOwner = (HUMAN*)CoTaskMemAlloc(sizeof(HUMAN));
    if (pDog->pOwner == 0) // 内存不够
        return E_OUTOFMEMORY;
    pDog->pOwner->nHumanID = hid;
    return S_OK;
}
```

注意，方法返回一个特殊的 `HRESULT E_OUTOFMEMORY`，以表示由于内存不足而操作失败。

`GetFromPound` 方法的调用者在用完了这些数据之后负责释放由该方法分配的内存：

```
DOG fido;
HRESULT hr = p->GetFromPound(&fido);
if (SUCCEEDED(hr)) {
    printf("The dog %h is owned by %h", fido.nDogID,
           fido.pOwner->nHumanID);
    // 数据已用完，释放内存
    CoTaskMemFree(fido.pOwner);
}
```

除非专门有文档说明，否则当方法失败的时候，客户可以假定该方法没有分配任何内存。

上面显示的例子使用一个纯粹的 `[out]` 参数。管理 `[in, out]` 参数要更加复杂一些。`[in, out]` 参数的内嵌指针必须由调用者使用任务分配器来分配。如果方法有必要对客户传递进来的内存重新分配，那么它必须使用 `CoTaskMemRealloc` 对它重新分配。如果调用者没有信息要传给方法，它可以在输入时传递一个 `null` 指针，而方法仍然可以使用

**CoTaskMemRealloc**（这个函数可以接受 **null** 指针，而且能够正常工作）。另一方面，如果方法没有信息要传回给调用者，那么它只需简单地释放掉由内嵌指针所指的内存即可。考虑下面的 IDL 方法定义：

```
HRESULT SendToVet([in, out] DOG *pDog);
```

假定调用者有一个有效的 **HUMAN**，并且希望把它作为参数传递进去，那么客户代码有可能是这样的：

```
HUMAN *pHuman = (HUMAN*) CoTaskMemAlloc(sizeof(HUMAN));
pHuman->nHumanID = 1522;
DOG fido = { 4111, pHuman };
if (SUCCEEDED(hr)) {
    if (fido.pOwner)
        printf("Dog is now owned by %h", fido.pOwner->nHumanID);
    CoTaskMemFree(fido.pOwner); // 释放空指针，没问题
}
```

方法实现可以重用由调用者提供的缓冲区，或者分配一个新的缓冲区（如果调用者传递一个 **null** 内嵌指针的话）：

```
STDMETHODIMP MyClass::SendToVet(/*[in, out]*/ DOG *pDog) {
    if (fido.pOwner == 0)
        fido.pOwner = (HUMAN*) CoTaskMemAlloc(sizeof(HUMAN));
    if (fido.pOwner == 0) // 分配失败
        return E_OUTOFMEMORY;
    fido.pOwner->nHumanID = 22;
    return S_OK;
}
```

由于处理这种包含内嵌指针的 [**in, out**] 参数有诸多微妙之处，所以通常的做法是，在接口文档中反复说明内嵌指针的内存管理规则。

前面的代码片断用到了最为便利的 COM 任务分配器接口。在 Windows NT 之前的 COM 版本中，任务分配器的主要操纵方式是通过 **IMalloc** 接口：

```
[ uuid(00000002-0000-0000-C000-00000000046), local, object ]
interface IMalloc : IUnknown {
    void *Alloc([in] ULONG cb);
    void *Realloc ([in, unique] void *pv, [in] ULONG cb);
    void Free([in, unique] void *pv);
    ULONG GetSize([in, unique] void *pv);
    int DidAlloc([in, unique] void *pv);
    void HeapMinimize (void);
}
```

为了得到任务分配器的 IMalloc 接口，COM 提供了 API 函数 CoGetMalloc：

```
HRESULT CoGetMalloc(
    [in] DWORD dwMemCtx, // 保留，但必须有
    [out] IMalloc **ppMalloc); // 存于此！
```

这意味着，如果不调用便利的 CoTaskMemAlloc 函数：

```
HUMAN *pHuman = (HUMAN*)CoTaskMemAlloc(sizeof(HUMAN));
```

那么，我们可以使用下面较为不便利的形式：

```
IMalloc *pMalloc = 0;
pHuman = 0;
HRESULT hr = CoGetMalloc(1, &pMalloc);
if (SUCCEEDED(hr)) {
    PHuman = (HUMAN*)pMalloc->Alloc(sizeof(HUMAN));
    pMalloc->Release();
}
```

后面这种形式的好处在于，它可以与 Windows NT 之前的 COM 版本保持兼容。一般来说，使用 CoTaskMemAlloc 这组函数的做法更为可取一些，因为它要求较少的代码，从而减少代码出错的机会。

到现在为止，我们对任务分配器的讨论一直集中在“对象如何以及何时分配内存、客户如何以及何时释放内存”。我们还没有讨论当对象与客户驻留在不同地址空间的时候，它们又是如何工作的。这主要是因为，在使用了接口列集器之后，客户和对象的实现方式应该不受到“是否位于同一地址空间”的影响。由于事实上 COM 任务分配器是从进程的私有地址空间中获取内存的，所以隐藏“任务分配器无法跨越两个地址空间”的事实，就成了接口存根和接口代理的任务。当接口存根调用某个对象的方法时，它把所有的[out]和[in,out]参数都列集到 ORPC 应答消息中。如图 7.1 所示，一旦此列集工作完成之后，接口存根（它其实就是对象套间内部的客户）使用 CoTaskMemFree 释放掉任何由方法分配的内存。这就等于释放了“在方法调用中通过任务分配器分配的，并且位于对象地址空间中的”所有内存。接口代理在接收到了 ORPC 应答消息之后，它使用 CoTaskMemAlloc 为所有“该由方法分配内存的参数”分配空间。当这些内存块被实际的客户通过 CoTaskMemFree 释放的时候，实际上这就释放了“在方法调用中通过任务分配器分配的，但是位于客户地址空间中的”所有内存。

程序员很容易忘记释放内存，这是众所周知的，所以有时候在一个进程中监视任务分配器的行为（或者漏掉哪些行为）是非常有用的功能。为了提供这样的功能，COM 允许用户自定义的探查（spy）对象钩住（hook）任务分配器的行为，任务管理器在每次操作之前或者之后都会通知探查对象。这个用户自定义的探查对象必须实现 IMallocSpy 接口：

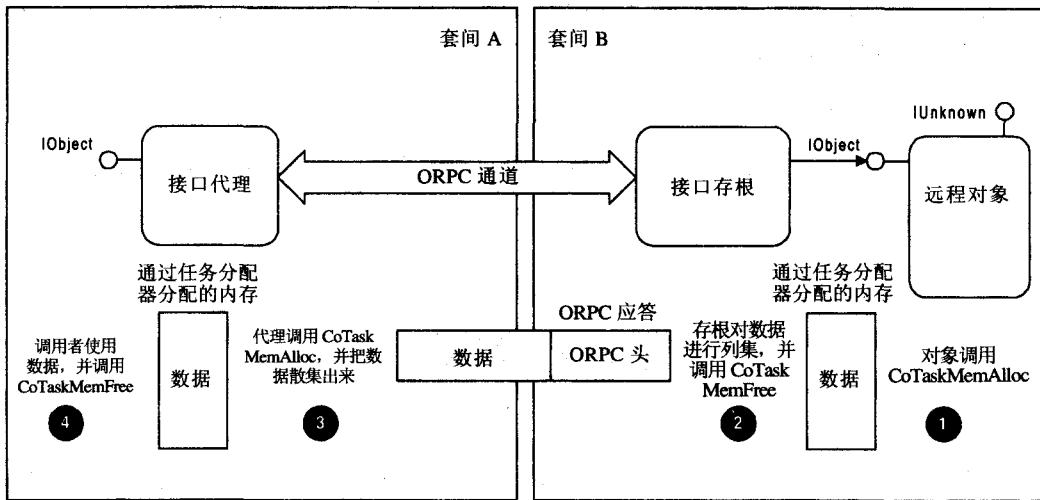


图 7.1 跨越进程边界使用任务分配器

```
[ uuid (0000001d-0000-0000-C000-00000000046), local, object ]
interface IMallocSpy : IUnknown {
    ULONG PreAlloc([in] ULONG cbRequest);
    void *PostAlloc( [in] void *pActual);
    void *PreFree([in] void *pRequest,[in] BOOL fSpayed);
    void PostFree([in] BOOL fSpayed);
    ULONG PreRealloc([in] void *pRequest, [in] ULONG cbRequest,
                    [out] void **ppNewRequest, [in] BOOL fSpayed);
    void *PostRealloc([in] void *pActual, [in] BOOL fSpayed);
    void *PreGetSize([in] void *pRequest, [in] BOOL fSpayed);
    ULONG PostGetSize([in] ULONG cbActual,[in] BOOL fSpayed);
    void *PreDidAlloc([in] void *pRequest,[in] BOOL fSpayed);
    int PostDidAlloc([in] void *pRequest,
                     [in] BOOL fSpayed, [in] int factual);
    void PreHeapMinimize (void);
    void PostHeapMinimize (void);
}
```

注意，对于 IMalloc 的每个方法，IMallocSpy 都有两个方法：一个被 COM “在分配器做实际工作之前” 调用，另一个被 COM “在分配器做完工作之后” 调用。在每个“前”方法中，用户自定义的探查对象可以修改“用户传给任务分配器的参数”。在每个“后”方法中，探查对象可以修改“分配器返回给调用者的结果”。这使得探查对象可以分配额外的内存，以便在每块内存中加入调试信息。COM 提供了一个 API 函数，可用于注册一个全进程范围内的 malloc 探查对象：

```
HRESULT CoRegisterMallocSpy([in] IMallocSpy *pms);
```

每个进程只能注册一个 malloc 探查对象（如果另一个探查对象已经被注册了，那么

CoRegisterMallocSpy 将返回 CO\_E\_OBJISREG)。为了注销一个 malloc 探查对象，COM 提供了下面的 API 函数 CoRevokeMallocSpy:

```
HRESULT CoRevokeMallocSpy(void);
```

如果使用当前探查对象分配的内存还没有被释放，那么 COM 不允许注销当前的探查对象。

### 7.3 数组

缺省情况下，指针参数总是被假设为指向单个实例的指针，而不是数组。为了在参数中传递一个数组，我们可以使用 C 语言的数组语法，或者使用特殊的 IDL 属性以指明数组的维数信息。传递数组最简单的技术是在编译时指定维数：

```
HRESULT Method1([in] short rgs[8]);
```

这种形式的数组被称为固定数组，它既是 IDL 中最简单的表示形式，也是运行时最简单、最紧凑的数据表示。对于上面例子中的数组来说，接口代理总是在 ORPC 请求消息中分配 16 字节 (8\*sizeof(short)) 的空间，然后把 8 个元素一起拷贝到消息中。一旦服务器接收到此 ORPC 请求消息，接口存根直接利用接收到的缓冲区作为函数的参数，如图 7.2 所示。因为数组的大小是固定的，数组所有的内容都在接收到的缓冲区中，所以接口存根就可以智能地重用缓冲区的内存，作为方法的实际参数。

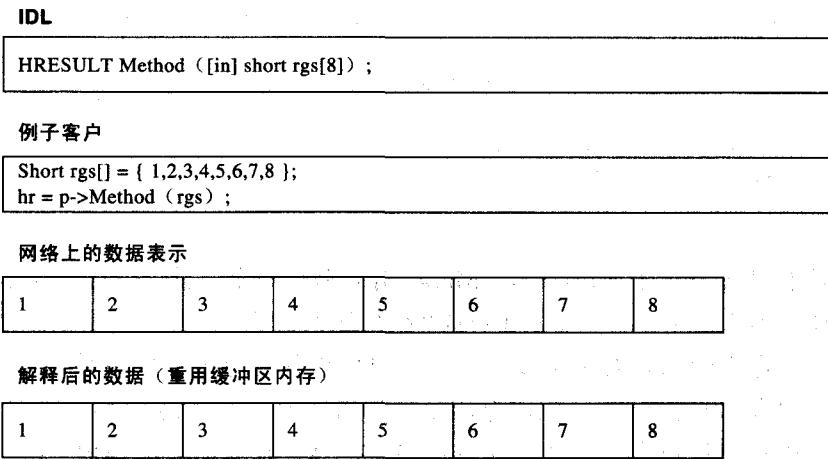


图 7.2 固定数组

如果在所有的情况下数组的长度都是 8，那么上面给出的方法就会很有用。它使得

调用者可以传递任何一个短整型数组，只要数组只有 8 个元素即可，例如：

```
void f(IFoo *pFoo) {
    short rgs[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
    pFoo->Method1(rgs);
}
```

在实践中，预测数组的长度往往是不太可能的，因为猜测得太小则意味着无法传递所有的元素，猜测得太大则使被传递的消息过于庞大。而且，如果数组是由复杂数据类型组成的，则对超过实际大小的数组进行列集可能会非常昂贵，甚至会引起列集错误。不过，如果数组的大小确实是个常数，并且在设计接口的时候其大小就是已知的话，那么固定数组就会非常有用。

为了允许在运行时确定数组的维数，IDL（和底层的有线网络协议 NDR）允许调用者在运行时指定数组的容纳能力。这种类型的数组被称为适应性数组（conformant arrays）。我们既可以在运行时，也可以在编译时指定适应性数组的最大合法索引值，数组的能力（也就是数组的适应能力[conformance]）在实际的元素之前被传输，如图 7.3。与固定数组一样，适应性数组也可以直接把接收到的缓冲区用作参数，被传给方法实现，而无须附加的拷贝操作，因为数组的整个内容都已经出现在接收到的消息中了。

#### IDL

```
HRESULT Method ([in] long cElems,[in,size_is(cElems)] short *prgs);
```

#### 例子客户

```
Short rgs[] = { 1,2,3,4,5,6,7,8 };
hr = p->Method ( sizeof ( rgs ) /sizeof ( *rgs ),rgs );
```

#### 网络上的数据表示

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

#### 解释后的数据（重用缓冲区内存）

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

图 7.3 适应性数组

IDL 使用 [size\_is] 属性，允许调用者指定数组的适应能力：

```
HRESULT Method2([in] long cElems,
               [in, size_is(cElems)] short rgs[*]);
```

或者

```
HRESULT Method3([in] long cElems,
               [in, size_is(cElems)] short rgs[]);
```

或者

```
HRESULT Method4([in] long cElems,
                [in, size_is(cElems)] short *rgs);
```

这三种风格对于底层包的格式而言都是等价的。因此，这些方法都允许调用者用下面的方式来指定数组的大小：

```
void f(IFoo *pFoo) {
    short rgs[] : { 1, 2, 3, 4, 5, 6, 7, 8 };
    pFoo->Method2(8, rgs);
}
```

作为一个参数属性，[size\_is]属性所使用的表达式可以使用同一个方法中任何其他的参数，可以使用算术、逻辑和条件运算符。例如，下面的 IDL 描述虽然不太容易被理解，但它是合法的：

```
HRESULT Method5([in] long arg1, [in] long arg2,
                [in] long arg3,
                [in, size_is(arg1 ? (arg3+1) : (arg1&arg2))]
                short *rgs);
```

函数调用或者其他可能会引起副作用的语言基本元素（比如++或者--操作符）都禁止在[size\_is]表达式中使用。

如果一个适应性数组被嵌在一个结构中，那么在描述这个数组的时候，[size\_is]属性可以使用同一个结构中任何其他成员，示例如下：

```
typedef struct tagCOUNTED_SHORTS {
    long cElems;
    [size_is(cElems)] short rgs[];
} COUNTED_SHORTS;

HRESULT Method6([in] COUNTED_SHORTS *pcs);
```

我们可以假设调用者会写出这样的代码来：

```
void SendFiveShorts (IFoo *pFoo) {
    char buffer [sizeof (COUNTED_SHORTS) + 4 * sizeof (short)];
    COUNTED_SHORTS& rcs = *reinterpret_cast<COUNTED_SHORTS*>(buffer);
    rcs.cElems = 5; rcs.rgs[0] = 0; rcs.rgs[1] = 1;
    rcs.rgs[2] = 2; rcs.rgs[3] = 3; rcs.rgs[4] = 4;
    pFoo->Method6(&rcs);
}
```

IDL 也支持[max\_is]属性，这是[size\_is]在形式上的一种变化。[size\_is]属性表示一

个数组中元素的总数目；[max\_is]属性表示一个数组中最大的合法索引值（它比数组中元素总数目少1）。这意味着下面两个声明是等价的：

```
HRESULT Method7([in, size_is(10)] short *rgs);
HRESULT Method8([in, max_is(9)] short *rgs);
```

有趣的是，虽然我们可以像上面的声明那样在[size\_is]属性中使用常数，但是使用固定数组的效率会略高一点。在上述两个例子中，它们都使用了适应性数组，虽然数组的维数是静态的，并且接口代理和接口存根在编译时就已经知道了它们的值，但是数组的维数仍然要随着数据一起被传输。

如果数组的内容只是从调用者一端被传输到方法实现中，那么适应性数组能够满足绝大多数的用途。然而，在许多情况下，调用者希望把一个空数组传给对象，然后让对象来填充数组中的值。使用适应性数组作为输出参数当然也是可能的，如下所示：

```
HRESULT Method9([in] long cMax,
                [out, size_is(cMax)] short *rgs);
```

这就暗指了下面的客户代码用法：

```
void f(IFoo *pFoo) {
    short rgs[100];
    pFoo->Method9(100, rgs);
}
```

以及下面的服务器实现代码：

```
HRESULT CFoo::Method9(long cMax, short *rgs) {
    for (long n = 0 ; n < cMax; n++)
        rgs[n] = n * n;
    return S_OK;
}
```

但是，如果方法实现不能正确地以有效元素填满整个数组，那又该怎么办呢？在前述的代码片断中，即使方法只对数组的前cMax/2个元素作了初始化，服务器端的存根仍将传输整个数组的cMax个元素。这显然是很不合理的，为了解决这个问题，IDL和NDR提供了第三种类型的数组：可变数组（varying array）。

可变数组是指这样的数组，它具有固定的能力（即总维数），但是它所包含的有效元素的个数可以少于数组的实际维数。使用了可变数组之后，只有一部分连续的内容会被传输，而不是传输整个数组。为了指定待传输的元素子集，IDL使用了[length\_is]属性。我们知道，[size\_is]属性描述了数组的容纳能力，与此不同的是，[length\_is]属性描述了数组的实际内容。考虑下面的IDL：

```
HRESULT Method10([in] long cActual,
                 [in, length_is(cActual)] short rgs[1024]);
```

在传输的时候，`cActual` 的值（也就是数组的变化特性`[variance]`）将位于被传输的数组元素之前。为了使被传输的数据区域可以出现在数组中的任何地方，而不一定是数组的起始处，IDL 和 NDR 也支持`[first_is]`属性，它指示被传输部分从哪里开始。此偏移值也要与数组内容一起被传输，因而散集器（unmarshaler）可以知道数组的哪部分子集要被初始化。我们知道`[max_is]`是`[size_is]`的变化形式，与此相仿，`[length_is]`也有一种变化形式，为`[last_is]`，它使用索引值来代替计数值。下面的两个定义是等价的：

```
HRESULT Method11([in, first_is(2), length_is(5)]
                  short rgs[8]);
HRESULT Method12([in, first_is(2), last_is(6)]
                  short rgs[8]);
```

这两个方法都指示列集器只传输 5 个数组元素，但是在散集的一端，它要分配 8 个元素的空间，并且传输进来的值被拷贝到适当的位置上。在接收缓冲区中不出现的元素都被初始化为 0。

可变数组可以降低网络的流量，因为只有真正必要的元素才被传输。然而，如图 7.4 所示，从内存拷贝的角度来看，可变数组的效率比适应性数组要低一些，因为服务器端存根传给方法实现的数组是从堆中分配的、单独的一块内存。这块内存首先被填充为 0，然后接收缓冲区中的内容被拷贝到适当的位置上。这使得在进入到方法之前，先要进行一次或者两次内存传送。对于大的数组就会明显地影响性能。我并不是说可变数组就没有用了，而是说，当可变数组被用作`[in]`参数的时候，它比一个逻辑上等价的适应性数组，效率要低一些。

**IDL**

HRESULT Method ([in] long cActual,[in] long iFirst, [in,length_is (cActual) ,first_is (iFirst) ] short prgs[8];
--

**例子客户**

Short rgs[8]; rgs[4] = 5; rgs[5] = 6 hr = p->Method ( 2,4,rgs ) ;
--

**网络上的数据表示**

4	2	5	6
---	---	---	---

**解释后的数据（独立的内存块）**

0	0	0	0	5	6	0	0
---	---	---	---	---	---	---	---

**图 7.4 可变数组**

与固定数组一样，可变数组要求接口设计者在编译时指定数组的适应能力。这使得可变数组非常受限制，因为在实践中，要预测一个接口所用到的数组的最佳缓冲区大小

是非常困难的（例如，有的客户对于内存的使用有非常严格的限制，而其他的客户则喜欢放入额外的数据，从而要求更大的缓冲区）。幸运的是，IDL 和 NDR 允许我们把 [size\_is] 和 [length\_is] 结合起来，从而既可以指定数组的内容（即变化），又可以指定数组的适应能力。当一个数组同时使用了这两个属性的时候，它被称为适应性可变数组，或者更为简捷一点，称为开放数组（open array）。为了指定一个开放数组，只要让调用者有办法通过参数来指定适应能力和内容即可，示例如下：

```
HRESULT Method13([in] cMax, [in] cActual,
                  [in,
                   size_is (cMax),
                   length_is(cActual)] short rgs[]);
```

或者：

```
HRESULT Method14([in] cMax, [in] cActual,
                  [in,
                   size_is (cMax),
                   length_is(cActual)] short rgs[*]);
```

或者：

```
HRESULT Method15([in] cMax, [in] cActual,
                  [in,
                   size_is (cMax),
                   length_is(cActual)] short *rgs);
```

对于以上任一种情形，客户端的代码都可以是：

```
void f(IFoo *pFoo) {
    short rgs[8];
    rgs[0] = 1; rgs[1] = 2;
    pFoo->Method13(8, 2, rgs);
}
```

如图 7.5 所示，在传输开放数组的时候，列集器首先写入数组的容纳能力，以及实际内容的偏移和长度。与可变数组的情形一样，数组的能力可能会大于被传输的元素个数。这意味着接收缓冲区中的内容不能直接被传给调用者，所以要用到第二块内存，从而增加了内存的开销。

对于输入参数来说，适应性数组是最有用的数组类型；对于输出参数或者输入/输出参数来说，开放数组是最有用的数组类型，因为它们允许调用者分配任意大小的缓冲区，而只有被用到的元素才被传输。为了允许这种用法，典型的 IDL 示意如下：

```
HRESULT Method16([in] long cMax,
                  [out] long *pcActual,
```

```
[out, size_is(cMax), length_is(*pcActual)]
    short *rgs);
```

**IDL**

<b>HRESULT Method ([in] long cMax,[in] long cActual,[in] long iFirst,</b>
<b>[in, size_is (cMax) ,length_is (cActual) ],first_is[ (iFirst) ] short *rgs) ;</b>

**例子客户**

Short rgs[8]; rgs[4] = 5; rgs[5] = 6;     hr = p->Method ( 8,2,4, rgs );
--

**网络上的数据表示**

8	4	2	5	6
---	---	---	---	---

**解释后的数据（独立的内存块）**

0	0	0	0	5	6	0	0
---	---	---	---	---	---	---	---

**图 7.5 开放数组**

此方法暗示了下面的客户代码：

```
void f(IFoo *pFoo) {
    short rgs[8];
    long cActual;
    pFoo->Method16(8, &cActual, rgs);
    // .. 首先处理 rgs 的 cActual 元素
}
```

以及与下面的代码类似的服务器端实现：

```
HRESULT CFoo::Method16(long cMax, long *pcActual,
    short *rgs) {
    *pcActual = min(cMax,5); // 首先只写 5 个元素
    for (long n = 0; n < *pcActual; n++)
        rgs[n] = n * n;
    return S_OK;
}
```

这使得调用者可以控制缓冲区的大小，而方法实现控制被传输元素的实际个数。

如果一个开放数组将被用作输入-输出参数，那么数组的变化信息需要在每个方向上被指定。如果输入时的元素个数与输出时的元素个数有可能不一致，那么变化参数必须也是一个输入 - 输出参数。示例如下：

```
HRESULT Method17([in] long cMax,
    [in, out] long *pcActual,
```

```
[in, out, size_is(cMax), length_is(*pcActual)] short *rgs;
```

此方法暗示了下面的客户代码：

```
void f(IFoo *pFoo) {
    short rgs[8];
    rgs[0] = 0; rgs[1] = 1;
    long cActual = 2;
    pFoo->Method17(8, &cActual, rgs);
    // 首先处理 rgs 的 cActual 元素
}
```

如果输入时的元素个数与输出时的元素个数一致，则适应性数组就可以满足要求，例如：

```
HRESULT Method18([in] long cElems,
    [in, out, size_is(cElems)] short *rgs);
```

这个方法既可以得益于适应性数组的高效率，又非常容易使用。

以上给出的例子都是处理一维数组的。考虑下面的 C 原型：

```
void g(short **arg1);
```

在 C 语言中，这个原型函数可以有多种含义。有可能这个函数期望得到一个指向单个短整数的指针的指针：

```
void g(short **arg1) { // 返回静态指针
    static short s;
    *arg1 = &s;
}
```

也有可能，函数期望得到一个内含 100 个短整型指针的数组：

```
void g(short **arg1) { // 100 个短整型指针数组
    for (int n = 0; n < 100; n++)
        *(arg1[n]) *= *(arg1[n]);
```

或者，函数期望得到一个指向短整型数组的指针：

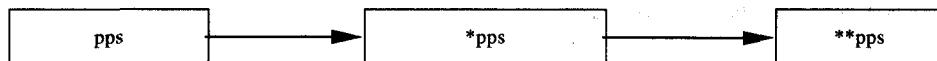
```
void g(short **arg1) { // 100 个短整型数组
    for (int n = 0; n < 100; n++)
        (*arg1)[n] *= (*arg1)[n];
```

IDL 使用特殊的语法来解决这些语法梦魇，初学者可以查看 IDL 的文档，以避免这些复杂语法带来的迷惑。

IDL 的 [size\_is] 和 [length\_is] 属性可以接受可变数目的变量，相互之间以逗号作为分割，每一层间接性都对应一个变量。如果某一个参数漏掉了，那么当前的间接层被假定为一个实例指针，而不是数组。要想指定某个参数是一个指向单个实例的指针的指针，并不需要任何专门的属性：

```
HRESULT Method19([in] short **pps);
```

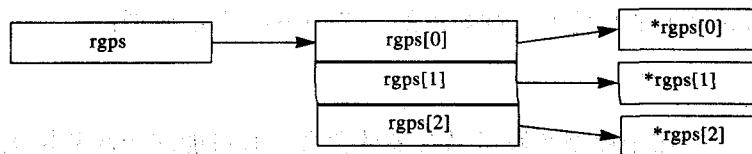
这意味着下面的内存布局结构：



若要表示一个参数是一个数组指针，且数组的元素为指向单个实例的指针，则可以采用下面的 IDL：

```
HRESULT Method20([in, size_is(3)] short **rgps);
```

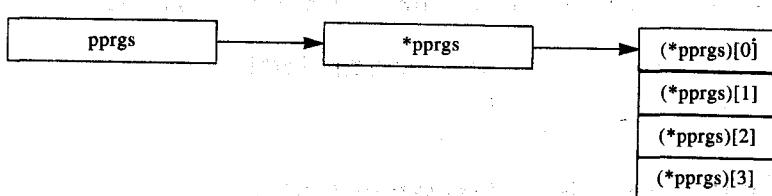
内存中的布局结构如下所示：



若要表示一个参数是一个指向实例数组的指针的指针，则可以使用下面的 IDL：

```
HRESULT Method21([in, size_is(,4)] short ***pprgs);
```

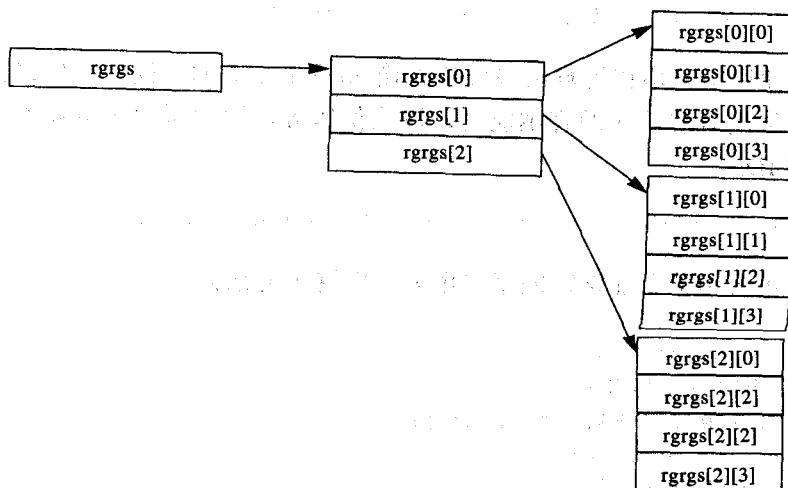
内存中的布局结构如下所示：



若要表示一个参数是一个指针数组，而数组的每个元素又是一个实例数组，则可以使用下面的 IDL：

```
HRESULT Method22([in, size_is(3,4)] short ***rgrrgs);
```

内存中的布局结构如下所示：



尽管这样的语法并没有覆盖所有可能的情况，但是它比 C 语言更加灵活，并且减少了二义性。

很重要的一点是，前面给出的 IDL 方法确实指定了一个多维数组；从技术上来看，这是一个指向指针的数组，而指针元素又指向一个实例数组。这与 C 语言的多维数组不同，在 IDL 中也可以使用标准的 C 语法：

```
HRESULT Method23([in] short rgrrgs[3][4]);
```

这里的语法假定数组的所有元素在内存中是连续的，而前面的例子则没有这样的假设条件。

使用 [size\_is] 属性来指定多维数组的第一维也是合法的，譬如：

```
HRESULT Method24([in, size_is(3)] short rgrrgs[] [4]);
```

但除了最左边的第一维之外，要想指定其他的维数则是不可能的。

在 [size\_is]、[length\_is] 或者其他与维数有关的属性中，它们所使用的表达式不能够包含函数调用。这将使字符串很难列集，因为字符串的适应能力（即字符串长度）或者变化部分是以 wcslen 或者 strlen 调用为基础的。这意味着下面的 IDL 是非法的：

```
HRESULT Method24([in, size_is(wcslen(wsz) + 1)]
                  const OLECHAR *wz);
```

因为这项限制使得客户程序使用字符串非常不方便，所以 IDL 特别支持 string 属性，它告诉列集层：调用适当的 xxxlen 函数来计算数组的适应能力。下面的方法指定一个字符串作为输入参数：

```
HRESULT Method25([in, string] const OLECHAR *wz);
```

或者

```
HRESULT Method25([in, string] const OLECHAR wsz[]);
```

当我们使用字符串作为输出参数或者输入/输出参数的时候，一种比较好的办法是，显式地指定调用者缓冲区的容纳能力，以确保服务器端的缓冲区足够大。考虑下面这个不太合适的 IDL：

```
HRESULT Method27([in, out, string] OLECHAR *pwsz);
```

如果调用者用某个非常短的字符串来调用这个方法：

```
void f(IFoo *pFoo) {
    OLECHAR wsz[1024];
    wcscpy(wsz, OLESTR("Hello"));
    pFoo->Method27(wsz);
    // ...处理已更新字符串
}
```

则在服务器端分配的字符数组的大小将根据输入字符串的长度来计算（应该为 6，包括结尾的 null 字符）。考虑下面服务器端的方法实现：

```
HRESULT CFoo::Method27(OLECHAR *wsz) {
    DisplayString(wsz);
    // wsz 只能有 6 个字符!
    wcscpy(wsz, OLESTR("Goodbye"));
    return S_OK;
}
```

因为数组的适应能力是以 `wcslen(OLESTR("Hello")) + 1` 为基础的，所以当方法实现在原来的缓冲区中写入一段比它更长的字符串信息时，字符串的尾端将会写到不该写入的随机内存中，极有可能在软件结束之前引起严重错误。这意味着，即使调用者已经预先分配了充足的存储空间来容纳结果字符串，服务器端的列集层并不知道这些看起来毫无关系的内存，它只分配仅够容纳 6 个 Unicode 字符串的空间。IDL 应该写成这样：

```
HRESULT Method28([in] long cchMax,
                 [in, out, string, size_is(cchMax)] OLECHAR *wsz);
```

调用者可以这样来使用该方法：

```
void f(IFoo *pFoo) {
    OLECHAR wsz[1024];
    wcscpy(wsz, OLESTR("Hello"));
    pFoo->Method28(1024, wsz);
    // ...处理已更新字符串
```

在[in, out, string]例子中最不幸的情形是，当输入字符串至少与输出字符串一样长的时候，它仍然可以有效地运行。与这个方法有关的错误将是间断性的（不确定的），有可能在工程的测试阶段永远也不会发生。

在大多数传统的 API 中，当函数向调用者返回一个可变长度的数据结构时，调用者要预先分配一个缓冲区来容纳函数的结果，而函数的实现代码填充这个由调用者提供的缓冲区。也就是说，调用者决定缓冲区的确切大小。用调用者指定大小的缓冲区来返回可变长度的数据结构存在一些问题，比如方法实现代码有可能希望返回更多的数据，其长度超过了调用者预先指定的值。考虑下面的 Windows SDK 代码，它显示编辑控制中的文本信息：

```
void Show(HWND hwndEdit) {
    TCHAR sz[1024];
    GetWindowText(hwndEdit, sz, 1024);
    MessageBox(0, sz, _TEXT("Hi!"), MB_OK);
}
```

注意，Show 的实现者猜测这个编辑控制所包含的字符数永远也不会超过 1024 个。他或者她如何知道这一点呢？实际上，我们可以想象下面的实现代码更加安全：

```
void Show(HWND hwndEdit) {
    int cch = GetWindowTextLength(hwndEdit);
    TCHAR *psz = new TCHAR[cch + 1];
    GetWindowText(hwndEdit, psz, cch);
    MessageBox(0, sz, _TEXT("Hi!"), MB_OK);
    delete[] psz;
}
```

但是在这个例子中，调用者如何断定：在 GetWindowTextLength 之后，但是在 GetWindowText 之前，这一段时间内用户不会输入一个字符呢？实际上，“根据可能过时的信息来分配空间”这样的事实使得这种用法容易受到“竞争条件（race condition）”的影响。

上面所提的编程习惯，对于 HWND 或许是可以接受的，但是对于 COM 对象则不能容忍的。与 HWND 不同的是，COM 对象很有可能同时被多个参与方访问。而且，像上面这种做法，为了执行一个操作而发出两个方法调用，代价太大，很容易导致性能下降，特别是在分布式环境中，包的传输和接收所带来的时延，将对调用者和方法之间的来回通信造成巨大的影响。由于这两个原因，当可变长度的数据类型以[out]参数的形式，被从方法实现一端传递给调用者的时候，设计良好的 COM 接口应该强迫方法实现代码使用 COM 任务分配器，为结果数据分配空间。这样做是必要的，因为只有在方法实现的内部它才知道结果数据的实际尺寸。这个被动态分配得到的缓冲区被返回给方法的调用者，当它不再有用的时候，调用者有责任释放此缓冲区（在调用者进程中通过 COM 任务分配器完成释放工作）。为了针对字符串参数表达这种习惯用法，下面的 IDL 可以

正确地工作：

```
HRESULT Method29([out, string]OLECHAR **ppwsz);
```

它意味着下面的服务器端实现代码：

```
HRESULT CFoo::Method29(OLECHAR **ppwsz) {
    const OLECHAR wsz[] = OLESTR("Goodbye");
    int cb = (wcslen(wsz) + 1) * sizeof(OLECHAR);
    *ppwsz = (OLECHAR*)CoTaskMemAlloc(cb);
    if (*ppwsz == 0) return E_OUTOFMEMORY;
    wcscpy(*ppwsz, wsz);
    return S_OK;
}
```

为了正确地使用这个方法，要求使用类似下面的客户端代码：

```
void f(IFoo *pFoo) {
    OLECHAR *pwsz = 0;
    if SUCCEEDED(pFoo->Method29(&pwsz)) {
        DisplayString(pwsz);
        CoTaskMemFree(pwsz);
    }
}
```

尽管这种用法会导致额外的内存拷贝代价，但是相对于降低了通信代价、以及“保证任何长度的字符串都可以被返回给调用者，并且不需要调用者为大字符串预先准备额外的缓冲区空间”而言，我们可以权衡分析这种做法是否值得。

本节以上给出的数组语法对于 C 或者 C++ 开发人员都是非常合理的。不幸的是，在本书写作的时候，Visual Basic 还不能处理任何可变尺寸的数组，只能识别固定数组。为了让 Visual Basic 也能够发送和接收可变尺寸的数组，COM IDL 文件定义了一个被称为 SAFEARRAY 的复合类型。SAFEARRAY 是一种很少见的数据结构，它允许 VARIANT 兼容类型的多维数组可以作为参数来传递。为了表达 SAFEARRAY 的维数，COM 提供了 SAFEARRAYBOUND 数据类型：

```
typedef struct tagSAFEARRAYBOUND {
    ULONG cElements;           // size_is 维数
    LONG lLbound;              // 维数的最小索引(通常为 0)
} SAFEARRAYBOUND;
```

SAFEARRAY 数据类型的内部使用了 SAFEARRAYBOUND 的适应性数组，把一些描述信息放在数组的内容中，定义如下：

```
typedef struct tagSAFEARRAY {
    USHORT cDims;               // 维数
```

```

USHORT fFeatures;           // 内容描述标志
ULONG cbElements;          // 每元素的字节数
ULONG clocks;              // 用于跟踪内存使用
void* pvData;              // 实际元素
[size_is(cDims)] SAFEARRAYBOUND rgsabound[];
} SAFEARRAY;

```

上面的 IDL 并不真正被用来描述 SAFEARRAY 的网络传输格式，而是用于在程序中描述 SAFEARRAY。

为了让用户在内存管理上具有最大的灵活性，COM 定义了下面的标志，用于 fFeatures 域：

```

typedef struct tagSAFEARRAY {
    USHORT cDims;           // 维数
    USHORT fFeatures;        // 内容描述标志
    ULONG cbElements;        // 每元素的字节数
    ULONG clocks;            // 用于跟踪内存使用
    void* pvData;            // 实际元素
    [size_is(cDims)] SAFEARRAYBOUND rgsabound[];
} SAFEARRAY;

```

为了指定 SAFEARRAY 数组元素的数据类型，IDL 编译器可以识别一个专门用于 SAFEARRAY 的语法：

```
HRESULT Method([in] SAFEARRAY(type) *ppsa);
```

这里 *type* 是 SAFEARRAY 中元素的类型。对应的 C++ 方法原型应该如下：

```
HRESULT Method(SAFEARRAY **psa);
```

注意，IDL 定义只用到了一层间接性（即只用到了一个\*号）；而在对应的 C++ 定义中，却用到了两层间接性。考虑下面的 IDL 定义，它指定了一个短整型 SAFEARRAY：

```
HRESULT Method([in] SAFEARRAY(short) *psa);
```

对应的 Visual Basic 定义将如下所示：

```
Sub Method( ByVal psa As Integer())
```

注意，在 Visual Basic 原型中，没有显式的数组维数。然而，前面提到过，Visual Basic 只支持固定数组。

有一组十分丰富的 API 函数可支持 SAFEARRAY 数据类型，它们可以重新指定 SAFEARRAY 数组的维数，也可以在不同的平台上传递 SAFEARRAY 数组。为了访问 SAFEARRAY 的元素，COM 提供了下面的 API 调用：

```
// 获得一个指向实际数组元素的指针
HRESULT SafeArrayAccessData([in] SAFEARRAY *psa,
```

```

        [out] void ** ppv);
// 释放 SafeArrayAccessData 返回的指针
HRESULT SafeArrayUnaccessData([in] SAFEARRAY *psa);
// 获得维数
ULONG SafeArrayGetDim([in] SAFEARRAY *psa);
// 获得一维的上界
HRESULT SafeArrayGetUBound([in] SAFEARRAY *psa,
                           [in] UINT nDim, [out] long *pUBound);
// 获得一维的下界
HRESULT SafeArrayGetLBound([in] SAFEARRAY *psa,
                           [in] UINT nDim, [out] long *pLBound);

```

这些方法提供了一种可移植而又安全的方法，用来访问数组中实际的数据。假设下面的 IDL：

```

HRESULT Sum([in] SAFEARRAY(long) *ppsa,
            [out, retval] long *pSum);

```

则下面的方法实现可以计算出长整型 SAFEARRAY 数组中所有元素的和：

```

STDMETHODIMP MyClass::Sum(SAFEARRAY **ppsa, long *pnSum) {
    assert(ppsa && *ppsa && pnSum);
    assert(SafeArrayGetDim(*ppsa) == 1);
    long iUBound, iLBbound;
    // 注意维索引是以 1 为基的
    HRESULT hr = SafeArrayGetUBound(*ppsa, 1, &iUBound);
    assert(SUCCEEDED(hr));
    hr = SafeArrayGetLBound(*ppsa, 1, &iLBbound);
    assert(SUCCEEDED(hr));
    long *prgn = 0;
    hr = SafeArrayAccessData(*ppsa, (void**)&prgn);
    *pnSum = 0;
    for (long i = 0; i < iUBound - iLBbound; i++)
        *pnSum += prgn[i];
    SafeArrayUnaccessData(*ppsa);
    return S_OK;
}

```

注意，任何与数组维数有关的 API 函数调用都使用以 1 为基的索引。

前面的代码片断只是简单地操作一个现有的 SAFEARRAY 数组的内容。为了创建一个一维的 SAFEARRAY，并作为参数传递给方法，COM 提供了下面的 API 函数 SafeArrayCreateVector，为 SAFEARRAY 结构和数组元素分配连续的内存块，原型如下：

```

SAFEARRAY *SafeArrayCreateVector(
    [in] VARTYPE vt,                      // 元素类型
    [in] long iLBbound,                    // 下界索引
    [in] unsigned iht cElems);           // 元素数

```

COM 也提供了各种函数用来分配多维数组，但对这些函数的讨论超出了本书的范围。给定下面的 IDL 方法定义：

```
HRESULT GetPrimes([in] long nStart, [in] long nEnd,
                   [out] SAFEARRAY(long) *ppsa);
```

下面的 C++ 方法定义给调用者返回一个由被调用方分配的 SAFEARRAY 数组：

```
STDMETHODIMP MyClass::GetPrimes(long nMin, long nMax,
                                 SAFEARRAY **ppsa) {
    assert (ppsa);
    UINT cElems = GetNumberOfPrimes(nMin, nMax);
    *ppsa = SafeArrayCreateVector(VT_I4, 0, cElems);
    assert (*ppsa);
    long *prgn = 0;
    HRESULT hr = SafeArrayAccessData(*ppsa, (void**)&prgn);
    assert(SUCCEEDED(hr));
    for (UINT i = 0; i < cElems; i++)
        prgn[i] = GetNextPrime(i ? prgn[i-1] : nMin);
    SafeArrayUnaccessData(*ppsa);
    return S_OK;
}
```

对应的 Visual Basic 客户代码应该看起来如下所示：

```
Function GetSumOfPrimes(ByVal nMin as Long,
                        ByVal nMax as Long) as Long
    Dim arr0 as Long
    Dim n as Variant
    Objref. GetPrimes nMin, nMax, arr
    GetSumOfPrimes = 0
    for each n in arr
        GetSumOfPrimes = GetSumOfPrimes + n
    Next n
End Function
```

可以被转译成下面的 C++ 代码：

```
long GetSumOfPrimes(long nMin, long nMax) {
    SAFEARRAY *pArray = 0;
    HRESULT hr = g_pObjRef->GetPrimes(nMin, nMax, &pArray);
    assert(SUCCEEDED(hr) && SafeArrayGetDim(pArray) == 1);
    long *prgn = 0;
    hr = SafeArrayAccessData(pArray, (void**)&prgn);
    long iUBound, iLBbound, result = 0;
    SafeArrayGetUBound(pArray, 1, &iUBound);
    SafeArrayGetLBound(pArray, 1, &iLBbound);
    for (long n = iLBbound; n <= iUBound; n++)
```

```

        result += prgn[n];
SafeArrayUnaccessData(pArray);
SafeArrayDestroy(pArray);
return n;
}

```

注意，在[out]参数中被返回的 SAFEARRAY 数组，其释放工作应该由调用者负责。调用 SafeArrayDestroy 函数可以正确地释放所有被 SAFEARRAY 拥有的内存和资源。

## 7.4 流程控制

注意，在前面那些使用数组和 SAFEARRAY 的例子中，数据的发送者可以决定 ORPC 消息中将有多少数据要被传送。假设下面一个很简单的 IDL 方法定义：

```

HRESULT Sum([in] long cElems,
           [in, size_is(cElems)] double *prgd,
           [out, retval] double *pResult);

```

如果调用者用下面的形式来调用这个方法：

```

double rgd[1024 * 1024 * 16];
HRESULT hr = p->Sum(sizeof(rgd)/sizeof(*rgd), rgd);

```

那么结果 ORPC 消息将至少达到 128MB。尽管底层 RPC 协议有很强的能力，可以把巨大的消息分割成许多个网络数据包，但是使用这么大的数组仍然存在一些问题。一个很显然的问题是，除了当前数组所占有的内存之外，调用者必须另有一块 128MB 以上的可用内存。这块重复的内存之所以需要，是因为接口代理要建立 ORPC 请求消息，数组的内容最终将被拷贝到该消息中。与之相关的另一个问题是，对象的进程必须也要有 128MB 以上的可用内存，才能把接收到的 RPC 包重构到一个连续的 ORPC 消息中。如果数组使用了[length\_is] 属性，那么还需要另一块 128MB 内存用来复制数组并传给方法。这个问题对于[in] 和 [out] 参数两种情形都存在。无论哪种情况下，数组的发送者都要有足够的缓冲区空间，才能建立起 ORPC 消息，但是数组的接收者可能不需要。问题发生的原因是，接收者没有应用层上的流程控制机制。

在前面的方法定义中，一个更为微妙的问题是时延问题。ORPC 调用的语义要求：在调用对象的方法之前，RPC/ORPC 层必须重构完整的 ORPC 消息。这意味着在最后一个数据包到达之前，对象还不能处理已经到达的数据。由于大数组的传输时间将会相当长，因此对象在这一段时间中将会空闲，等待最后一个包的到达。在这段空闲的过程中，可能有许多元素陆陆续续成功地到达对象的地址空间；然而，COM 方法调用的语义要

求“所有的元素都到达之后才执行实际的调用”。当数组被作为[out]参数传递的时候，同样的问题也存在，因为客户不可能在只接收到部分结果的情况下就开始处理某个操作。

为了解决有关“以方法参数的形式传递大数组”的问题，COM 有一个标准的接口设计习惯，它允许数据的接收者可以显式地完成数据元素的流程控制。这个设计习惯要求传递一个特殊的 COM 接口指针，而不是传递实际的数组。这个特殊的接口被称为枚举器（enumerator），它使接收者可以主动地、按照它所希望的速度从发送者处“拉（pull）”元素过来。为了把这种设计习惯应用到前面的方法定义上，下面的接口定义是必需的：

```
interface IEnumDouble : IUnknown {
    // 从发送者处拉多个元素
    HRESULT Next([in] ULONG cElems,
                [out; size_is(cElems), length_is(*pcFetched)]
                    double *prgElems,
                [out] ULONG *pcFetched);
    // 游标前进 cElems 个元素
    HRESULT Skip([in] cElems);
    // 将游标重置为第一个元素
    HRESULT Reset(void);
    // 复制枚举器的当前游标
    HRESULT Clone([out] IEnumDouble **pped);
}
```

有一点很重要并值得注意的是，IEnum 接口只是简单地建立了“游标（cursor）”的模型，而不是实际数组的模型。有了这个接口定义之后，原来的 IDL 方法定义：

```
HRESULT Sum([in] long cElems,
            [in, size_is(cElems)] double *prgd,
            [out, retval] double *pResult);
```

就变成了下面的定义：

```
HRESULT Sum([in] IEnumDouble *ped,
            [out, retval] double *pResult);
```

注意，元素的计数总数不再是必要的了，因为当 IEnumDouble::Next 方法返回一个可被识别的 HRESULT (S\_FALSE) 的时候，数据的接收者就会知道数组已经结束了。

有了前面的接口定义之后，下面是正确的方法实现：

```
STDMETHODIMP MyClass::Sum(IEnumDouble *ped, double *psum) {
    assert(ped && psum);
    *psum = 0;
    HRESULT hr;
    do {
```

```

// 声明接收元素的缓冲区
enum { CHUNKSIZE = 2048 };
double rgd[CHUNKSIZE];
// 要求数据生成者发送 CHUNKSIZE 个元素
ULONG cFetched;
hr = ped->Next(CHUNKSIZE, rgd, &cFetched);
// 调整 cFetched 以处理松散对象
if (hr == S_OK) cFetched = CHUNKSIZE;
if (SUCCEEDED(hr))           // S_OK 或 S_FALSE
// 使用接收到的元素
for (ULONG n = 0; n < cFetched; n++)
    *psum += rgd[n];
} while (hr == S_OK);        // S_FALSE 或错误终止
}

```

注意，如果发送者还有其他元素的话，`Next` 函数将返回 `S_OK`；如果发送者已经完成了数据传送，则 `Next` 函数将返回 `S_FALSE`。而且请注意，以上的代码也做了一些特殊的处理，以补偿“万一 `Next` 函数在返回 `S_OK` 的时候忘记了设置 `cFetched` 变量”的疏忽（`Next` 函数返回 `S_OK` 表示所有请求的元素都被取到了）。

使用 `IEnum` 的一个优点是，它允许发送者延迟产生数组元素。考虑下面的 IDL 方法定义：

```

HRESULT GetPrimes([in] long nMin, [in] long nMax,
                   [out] IEnumLong **ppe);

```

对象的实现者可以创建一个专门的类来产生所要求的素数（这个类被称为 `PrimeGenerator`，即素数发生器），它实现了 `IEnumLong` 接口：

```

class PrimeGenerator : public IEnumLong {
    LONG m_cRef;           // COM 引用计数
    long m_nCurrentPrime;  // 游标
    long m_nMin;           // 最小素数值
    long m_nMax;           // 最大素数值
public:
    PrimeGenerator(long nMin, long nMax, long nCurrentPrime)
        : m_cRef(0), m_nMin(nMin), m_nMax(nMax),
          m_nCurrentPrime(nCurrentPrime) {
    }
    // IUnknown 方法
    STDMETHODIMP QueryInterface(REFIID riid, void**ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
    // IEnumLong 方法
    STDMETHODIMP Next(ULONG, long *, ULONG *);
    STDMETHODIMP Skip(ULONG);
    STDMETHODIMP Reset(void);
    STDMETHODIMP Clone(IEnumLong **ppe);
};

```

发生器的 Next 实现只是简单地产生所要求的素数：

```
STDMETHODIMP PrimeGenerator::Next(ULONG cElems,
                                  long *prgElems, ULONG *pcFetched) {
    // 确保 pcFetched 有效如果 cElems > 1
    if (cElems > 1 && pcFetched == 0)
        return E_INVALIDARG;
    // 填充缓冲区
    ULONG cFetched = 0;
    while (cFetched < cElems && m_nCurrentPrime <= m_nMax) {
        prgElems[cFetched] = GetNextPrime(m_nCurrentPrime);
        m_nCurrentPrime = prgElems[cFetched]++;
    }
    if (pcFetched) // 有些调用者会传递 NULL
        *pcFetched = cFetched;
    return cFetched == cElems ? S_OK : S_FALSE;
}
```

注意，即使有几百万个可能的数值，也不会有太多的数值同时驻留在内存中。  
发生器的 Skip 方法只是简单地产生并丢弃掉指定数目的素数即可：

```
STDMETHODIMP PrimeGenerator::Skip(ULONG cElems) {
    ULONG cEaten = 0;
    while (cEaten < cElems && m_nCurrentPrime <= m_nMax)
        m_nCurrentPrime = GetNextPrime(m_nCurrentPrime);
    cEaten++;
}
return cEaten == cElems ? S_OK : S_FALSE;
}
```

Reset 方法把游标重置为初始值：

```
STDMETHODIMP PrimeGenerator::Reset(void) {
    m_nCurrentPrime = m_nMin;
    return S_OK;
}
```

Clone 方法根据当前发生器的最小值、最大值和当前值，创建一个新的素数发生器：

```
STDMETHODIMP PrimeGenerator::Clone(IEnumLong **ppe) {
    assert(ppe);
    *ppe = new PrimeGenerator(m_nMin, m_nMax, m_nCurrent);
    if (*ppe)
        (*ppe)->AddRef();
    return S_OK;
}
```

有了素数发生器（PrimeGenerator）的实现之后，实际对象的 GetPrimes 方法实现就非常简单了：

```
STDMETHODIMP MyClass::GetPrimes(long nMin, long nMax,
                                 IEnumLong **ppe) {
    assert(ppe);
    *ppe = new PrimeGenerator(nMin, nMax, nMin);
    if (*ppe)
        (*ppe)->AddRef();
    return S_OK;
}
```

注意，现在 GetPrimes 方法的大部分实现代码位于 PrimeGenerator 中，而不是在对象的类中。

## 7.5 动态与静态调用

正如到现在为止我们所讨论的，COM 建立在“在开发客户程序的时候我们必须要有关于接口定义的先验知识”的基础上。这既可以通过 C++ 头文件（对于 C++ 客户）获得，也可以通过类型库（对于 Java 和 Visual Basic 客户）获得。一般来讲，这并不成问题，因为用这些语言编写的程序在发布之前，往往要经过编译阶段。有的语言在开发过程中并没有经过这样的编译阶段，而是以源代码的形式被配置发布，在运行时才被解释和执行。可能最普遍和流行的语是以 HTML 为基础的脚本语言（例如 VBScript、JavaScript），它们在 Web 浏览器或者 Web 服务器的环境中执行。在这两种情况下，脚本代码都是以纯文本的格式被嵌在 HTML 文件中，当 HTML 文件被解析的时候，运行时环境会根据需要执行脚本代码。为了提供一个丰富的程序设计环境，这些环境都允许脚本代码可以调用 COM 对象的方法，这些 COM 对象既可以是由这些脚本代码本身创建的，也可能由其他的 HTML 流创建的（例如，ActiveX 控制也可以是 Web 页面的一部分）。在这些环境中，目前还不可能使用类型库或者其他先验知识，把接口的描述信息提供给运行时引擎。这意味着对象自身必须要帮助脚本解释器，将文本形式的脚本代码翻译为有意义的方法调用。

为了允许解释执行的环境（比如 VBScript 和 JavaScript）也能够使用 COM 对象，COM 定义了一个接口，用来表达这种翻译机制。这个接口被称为 IDispatch，定义如下：

```
// 为一组已命名参数建模的结构
typedef struct tagDISPPARAMS {
    [size_is(cArgs)] VARIANTARG * rgvarg;
    [size_is(cNamedArgs)] DISPID * rgdispidNamedArgs;
```

```

    UINT cArgs;
    UINT cNamedArgs;
} DISPPARAMS;

// 对象能描述这个接口吗?
HRESULT GetTypeInfoCount([out] UINT * pctinfo);

// 返回此接口与地区有关的描述
HRESULT GetTypeInfo (
    [in] UINT iTInfo,           // 保留
    [Ein] LCID lcid,           // 地区 ID
    [out] ITypeinfo ** ppTInfo // 存于此!
);

// 将成员/参数的名字解析为 DISPID
HRESULT GetIDsOfNames (
    [in] REFIID riid,          // 保留, 必须为 IID_NULL
    [in, size_is(cNames)] LPOLESTR * rgszNames, // methoU+params
    [in] UINT cNames,           // 名字数量
    [in] LCID lcid,             // 地区 ID
    [out, size_is(cNames)] DISPID * rgid // 名字的令牌符号
);

// 通过 DISPID 访问成员
HRESULT Invoke(
    [in] DISPID id,             // 名字的令牌符号
    [in] REFIID riid,           // 保留, 必须是 IID_NULL
    [in] LCID lcid,              // 地区 ID
    [in] WORD wFlags,            // 方法, propput, 还是 propget?
    [in,out] DISPPARAMS * pDispParams, // 逻辑参数
    [out] VARIANT * pVarResult,   // 逻辑结果
    [out] EXCEPINFO * pExcepInfo, // IErrorInfo 参数
    [out] UINT * puArgErr        // 用于类型错误
);

```

当一个脚本引擎首次尝试要访问一个对象的时候, 它使用 `QueryInterface` 向对象请求 `IDispatch` 接口。如果对象不能满足这个 `QueryInterface` 请求, 那么脚本引擎就无法使用这个对象。如果对象成功地把它的 `IDispatch` 接口返回给脚本引擎, 则引擎将使用对象的 `GetIDsOfNames` 方法, 把方法和属性的名字翻译为令牌符号。这些令牌符号的正式名称为 `DISPID`, 它们实际上只是简单有效的、可被解析的整数而已, 这些整数唯一地标识了一个属性或者方法。在得到了方法或者属性的 `DISPID` 之后, 引擎只要调用对象的 `IDispatch::Invoke` 方法, 就可以请求原先名字所命名的方法或者属性。注意, 因为 `IDispatch::Invoke` 通过 `DISPPARAMS` 结构来接受操作的参数值, 而 `DISPPARAMS` 结构包含 `VARIANT` 数组, 所以它所支持的参数类型局限于 `VARIANT` 所能存储的类型范围。

以 `IDispatch` 为基的接口(通常被称为 `dispinterface`)逻辑上等价于一个普通的 COM 接口。主要的区别在于接口的逻辑操作实际是如何被调用的。对于普通的 COM 接口,

方法的调用是以“该方法原型的静态先验知识”为基础。对于 `dispinterface`，方法的调用是以“该方法调用的预期原型的文字表示”为基础。如果调用者正确地猜测出方法的原型，那么这个调用就可以被正确地分发。如果调用者不能正确地猜测出方法的原型，那么这个调用就不可能被分发。如果方法参数中使用了不正确的数据类型，那么（在可能的情况下）把它们转换成正确的类型是对象的任务。

在 IDL 中，描述一个 `dispinterface` 最简单的方法是使用 `dispinterface` 关键字：

```
[ uuid(75DA6450-DD0F-11d0-8C58-0080C73925BA) ]
dispinterface DPrimeManager {
properties:
    [id(1), readonly] long MinPrimeOnMachine;
    [id(2)] long MinPrime;
methods:
    [id(3)] long GetNextPrime([in] long n);
}
```

这里的语法可读性很强，然而，它假设调用者将总是通过 `IDispatch` 来访问对象的属性和方法。历史表明，随着开发环境和运行时环境的不断发展，它们总是能够适应普通 COM 接口的使用。为了确保 `dispinterface` 也能够有效地被将来的脚本环境访问，通常更好的做法是把接口设计成双接口（`dual` 接口）。

双接口非常简单，它是从 `IDispatch` 接口派生的普通 COM 接口。因为 `IDispatch` 接口是基接口，所以双接口完全兼容于解释执行的脚本客户。然而，双接口也向上兼容于那些“可直接绑定到一个静态定义的 COM 接口”的环境。以下的 IDL 定义是 `DIPrimeManager` 的双接口版本：

```
[ object, dual, uuid(75DA6450-DD0F-11d0-8C58-0080C73925BA) ]
interface DIPrimeManager : IDispatch {
    [id(1), propget] HRESULT MinPrimeOnMachine(
        [out, retval] long *pval);
    [id(2), propput] HRESULT MinPrime([in] long val);
    [id(2), propget] HRESULT MinPrime(
        [out, retval] long *pval);
    [id(3)] long GetNextPrime([in] long n);
}
```

注意，接口 `DIPrimeManager` 是从 `IDispatch` 继承的，而不是从 `IUnknown` 接口继承的。也请注意该接口有 `[dual]` 属性。这个属性使得被产生的类型库包含该接口的 `dispinterface` 版本，可以与“非双接口感知（non-dual-interface-aware）”的环境相兼容。`[dual]` 属性包括 `[oleautomation]` 属性，并且使类型库在 `RegisterTypeLib` 时刻为通用列集器（universal marshaler）加入相应的注册表键。

如果一个接口被定义成双接口，则 `IDispatch` 方法的实现就不是很重要了。这是因为类型库解析器实现了 `IDispatch` 接口四个方法中的两个方法。假设我们已经有了前面

定义的双接口，对象只需在初始化的时候装入类型库：

```
class PrimeManager : DIPrimeManager {
    LONG m_cRef;           // COM 引用计数
    ITypelInfo *m_pTypeInfo; // 类型的指针
    // IUnknown 方法...
    // IDispatch 方法...
    // IPrimeManager 方法...
    PrimeManager(void) : m_cRef(0)
        ITypelib *ptl = 0;
        HRESULT hr = LoadRegTypeLib(LIBID_PrimeLib, 1, 0, 0, &ptl);
        assert(SUCCEEDED(hr));
        hr = ptl->GetTypeInfoOfGuid(IID_DIPrimeManager,
                                       &m_pTypeInfo);
        ptl->Release();
    }
    virtual ~PrimeManager(void) {
        m_pTypeInfo->Release();
    }
};
```

给出了上面的类定义之后，`GetTypeInfo` 方法只是返回接口的描述：

```
STDMETHODIMP PrimeManager::GetTypeInfo(UINT it, LCID lcid,
                                         ITypelInfo **ppti) {
    assert(it == 0 && ppti != 0);
    (*ppti = m_pTypeInfo)->AddRef();
    return S_OK;
}
```

如果对象支持多种语言的类型库，则 `GetTypeInfo` 实现可以利用 `LCID` 参数来决定返回哪个类型描述。对应的 `GetTypeInfoCount` 实现更加简单：

```
STDMETHODIMP PrimeManager::GetTypeInfoCount(UINT *pit) {
    assert(pit != 0);
    *pit = 1; // 仅允许 0 或 1
    return S_OK;
}
```

唯一合法的计数值为 0（这表明对象不提供有关它的接口的描述）和 1（表示对象提供有关它的接口的描述）。即使对象支持多种语言的类型描述，结果计数值仍然为 1。

从技术上讲，`GetTypeInfo` 和 `GetTypeInfoCount` 方法是可选的。`IDispatch` 接口的真正核心在于 `GetIDsOfNames` 和 `Invoke`。`GetIDsOfNames` 方法只是简单地把调用转给类型库的解析引擎（它被内置在 COM 中）：

```
STDMETHODIMP PrimeManager::GetIDsOfNames(REFIID riid,
                                         OLECHAR **pNames, UINT cNames,
```

```

    LCID lcid, DISPID *pdispids) {
    assert(riid == IID_NULL);
    return m_pTypeInfo->GetIDsOfNames(pNames, cNames, pdispids);
}

```

因为类型库包含了所有的方法名字和对应的 DISPID，所以对于解析器来说这是非常简单不过的。Invoke 方法用类似下面代码的方式来实现：

```

STDMETHODIMP PrimeManager::Invoke(DISPID id,
    REFIID riid, LCID lcid, WORD wFlags,
    DISPPARAMS *pd, VARIANT *pVarResult,
    EXCEPINFO *pe, UINT *pu) {
    assert(riid == IID_NULL);
    void *pvThis = static_cast<DIPrimeManager*>(this);
    return m_pTypeInfo->Invoke(pvThis, id, wFlags,
        pd, pVarResult, pe, pu);
}

```

注意，`ITypeInfo::Invoke` 的第一个参数是一个接口指针，它的类型必须与类型信息中描述的接口完全一致。当传递进来的参数已经被正确地解析到调用栈上，解析器将通过这个接口指针调用实际的方法。图 7.6 演示了脚本环境通过双接口调用对象方法的调用顺序。

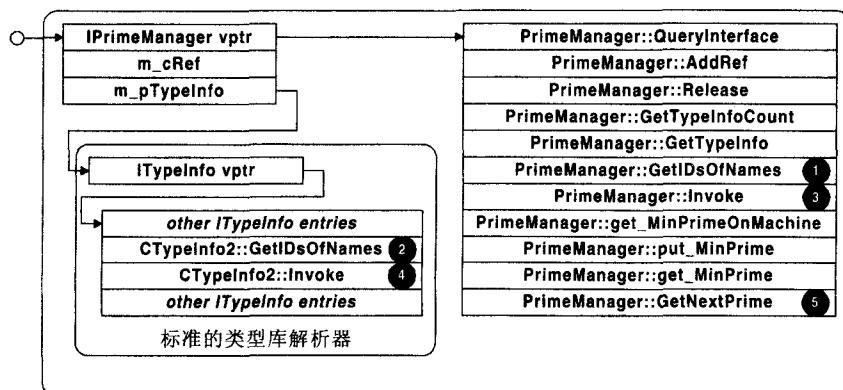


图 7.6 动态调用和双接口

## 7.6 双向接口协议

正如第 5 章所说的，驻留在不同套间中的对象可以相互之间利用其他对象的服务，而无需考虑其他的对象到底驻留在哪个套间中。因为 COM 远程结构是以套间概念为基

础的，所以开发人员不应该把进程看作是纯粹的客户或者服务器，而应看作是一个或者多个套间的集合，它们能够同时引出或者引入接口。

两个对象之间如何协商“用哪个接口来协作通信”？很大程度上讲，这是一个与领域相关的问题。例如，下面的接口针对程序员建立了一个模型：

```
[uuid(75DA6457-DD0F-11d0-8C58-0080C73925BA), object]
interface IProgrammer : IUnknown {
    HRESULT StartHacking(void);
    HRESULT IsProductDone([out, retval] BOOL *pbIsDone);
}
```

这个接口意味着下面的客户使用模式：

```
HRESULT ShipSoftware(void) {
    IProgrammer *pp = 0;
    HRESULT hr = CoGetObject(OLESTR("programmer:Bob"), 0,
                           IID_IProgrammer, (void**)&pp);
    if (SUCCEEDED(hr)) {
        hr = pp->StartHacking();
        BOOL blsDone = FALSE;
        while (!blsDone && SUCCEEDED(hr)) {
            Sleep(15000); // 等待 15 秒
            hr = pp->IsProductDone(&blsDone); // 检测状态
        }
        pp->Release();
    }
}
```

很显然，这段代码是非常低效的，因为客户循环地每隔 15 分钟检查一次对象的状态。一种更有效的办法是，客户提供第二个对象，一旦程序员对象（programmer object）到达了预期的状态之后，它就可以通知这第二个对象。这个由客户提供的对象必须要引出一个接口，这个接口提供了一个可供程序员对象进行操作的支持环境：

```
[uuid(75DA6458-DD0F-11d0-8C58-0080C73925BA), object]
interface ISoftwareConsumer : IUnknown {
    HRESULT OnProductIsDone(void);
    HRESULT OnProductWillBeLate([in] hyper nMonths);
}
```

有了这个配套的接口定义之后，还需要有一种机制来告诉程序员对象：客户已经实现了一个 ISoftwareConsumer 接口，可以接收来自程序员对象的状态变化通知。一种很常见的技术是，为 IProgrammer 接口定义一个显式的方法，通过这个方法客户可以把程序员对象与消费者对象（consumer object）关联起来。这种技术典型的做法是提供一个 Advise 方法：

```
interface IProgrammer : IUnknown {
    HRESULT Advise([in] ISoftwareConsumer *psc,
                  [out] DWORD *pdwCookie);
    :
    :
}
```

客户通过 `Advise` 方法提供配套的消费者对象，而程序员对象返回一个代表这种关联的 `DWORD` 值。以后，这个 `DWORD` 可以被用在对应的 `Unadvise` 方法中：

```
interface IProgrammer : IUnknown {
    :
    :
    HRESULT Unadvise([in] DWORD dwCookie);
}
```

`Unadvise` 方法通知程序员对象终止它与消费者对象之间的关联。用一个唯一的 `DWORD` 值来代表程序员对象和消费者对象之间的关联，这样的接口设计允许任意个数的消费者对象与程序员对象连接起来（或者断开连接），而且相互之间保持独立。

如果 `IProgrammer` 接口包含这两个方法，那么程序员对象可以在 `Advise` 方法中获取消费者对象，并一直拥有该消费者对象：

```
STDMETHODIMP Programmer::Advise(ISoftwareConsumer *pcs,
                                 DWORD *pdwCookie) {
    assert(pcs);
    if (m_pConsumer != 0)           // 已经有消费者了吗?
        return E_UNEXPECTED;
    (m_pConsumer = pcs)->AddRef(); // 新的消费者
    *pdwCookie = DWORD(pcs);       // 产生合理的 Cookie
    return S_OK;
}
```

对应的 `Unadvise` 实现应该与下面的代码类似：

```
STDMETHODIMP Programmer::Unadvise(DWORD dwCookie) {
// cookie 与当前消费者对应吗?
    if (DWORD(m_pConsumer) != dwCookie)
        return E_UNEXPECTED;
    (m_pConsumer)->Release();      // 释放当前消费者
    m_pConsumer = 0;
    return S_OK;
}
```

程序员和消费者之间的关系如图 7.7 所示。尽管这种实现方式在同一时刻只允许一个消费者，但是更加复杂一点的程序员对象可以管理一个动态的 `ISoftwareConsumer` 接口指针数组，因而它可以同时处理多个消费者，这完全有可能的。

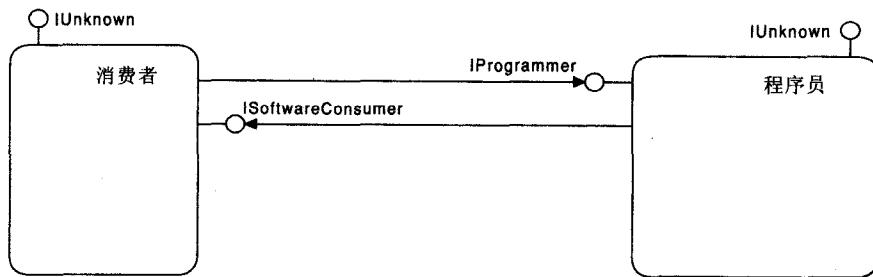


图 7.7 协作对象

有了上面的实现之后，现在程序员的 StartHacking 方法可以通过消费者来指示什么时候产品已经完成了：

```

STDMETHODIMP Programmer::StartHacking(void) {
    assert(m_pConsumer);
    // 预先通知还是推后
    HRESULT hr = m_Consumer->OnProductWillBeLate(3);
    if (FAILED(hr))
        return PROGRAMMER_E_UNREALISTICCONSUMER;
    // 产生一些代码
    extern char *g_rgpszTopFiftyStatements[];
    for (int n = 0; n < 100000; n++)
        printf(g_rgpszTopFiftyStatements[rand() % 50]);
    // 通知消费者已完成
    hr = m_pConsumer->OnProductIsDone();
    return S_OK;
}

```

ISoftwareConsumer 接口的实现代码是否与程序员对象属于同一个套间，这并不重要。实际上，对 StartHacking 方法的调用有可能来自消费者对象所在的套间，在这种情况下，调用者的套间将被重入，它的服务实际上就变成了同步回调。虽然上述方法实现对消费者对象发出了嵌套调用，但是程序员对象仍有可能在将来任意时刻调用消费者的方法。这种特权一直持续到调用 Unadvise 方法终止关联为止。

因为 IPrgammer 和 ISoftwareConsumer 接口有可能是先后被设计出来的，所以“IPrgammer 接口专门提供一个或者多个方法用来建立两者之间的关系”就变成了“如何使用程序员对象”的协议的一部分，而且这样做也是非常合理的。“程序员对象可以使用一个或者多个软件消费者对象”这一事实也可以被写到文档中，作为 IPrgammer 接口的协议的一部分，从而使 IPrgammer 接口的语义约定更加明确。然而，在有些情况下，协作接口或者回调接口被设计成超出了任何其他接口的范畴。下面的接口就是这样一个例子：

```
[ uuid(75DA645D-DD0F-11d0-8C58-0080C73925BA), object ]
interface IShutdownNotify : IUnknown {
    HRESULT OnObjectDestroyed([in] IUnknown *pUnk);
}
```

这个接口假定，IShutdownNotify 的实现者对于接收来自其他对象的结束事件非常感兴趣。但是这个接口定义并没有指定“这些感兴趣的参与方通过什么样的机制来通知对象，它们希望在对象被析构的时候可被通知到”。如图 7.8 所示，一种可能的策略是定义第二个配套接口，并且让对象实现这个接口：

```
[ uuid(75DA645E-DD0F-11d0-8C58-0080C73925BA), object ]
interface IShutdownSource : IUnknown {
    HRESULT Advise([in] IShutdownNotify *psn,
                  [out] DWORD *pdwCookie);
    HRESULT Unadvise([in] DWORD dwCookie);
}
```

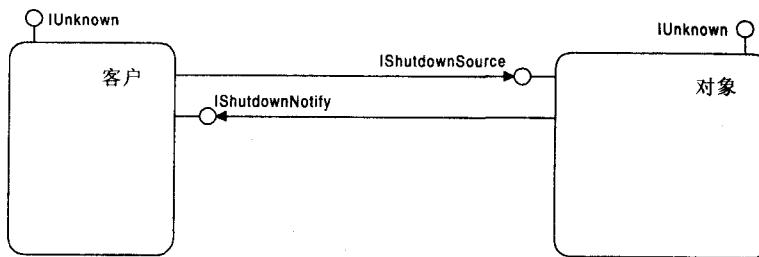


图 7.8 专用于对外的接口管理

然而，这个接口存在的目的，只是为了使客户可以让它们的 IShutdownNotify 接口能够连接到对象上。如果有大量的回调接口类型存在，那么就需要定义同等数量的连接管理接口。很显然，我们需要一种更为通用的机制，这就是连接点（connection point）机制。

连接点是 COM 的惯用语，表示向一个对象注册或者注销回调接口。在建立高度连接的对象网络的时候，连接点不是必需的。而且连接点也不能建立双向通信。相反，连接点机制把“对外的接口注册”的一般性概念表示成少量的标准基础接口。这些接口中最基础的就是 IConnectionPoint：

```
[ object, uuid(B196B286-BAB4-101A-B69C-00AA00341D07) ]
interface IConnectionPoint : IUnknown {
    // 可以连接何种类型接口
    HRESULT GetConnectionInterface([out] lid * pIID);
    // 获得一个“真实”object 的身份指针
    HRESULT GetConnectionPointContainer(
        [out] IConnectionPointContainer ** ppCPC);
```

```

// 拥有并使用 pUnkSink 直至已被注意到
HRESULT Advise([in] IUnknown * pUnkSink,
               [out] DWORD * pdwCookie);
// 停止拥有和使用与 dwCookie 相关联的指针
HRESULT Unadvise([in] DWORD dwCookie);
// 获取当前指针信息
HRESULT EnumConnections([out] IEnumConnections ** ppEnum);
}

```

如图 7.9 所示，对象针对每个“可被用作回调接口”的接口类型，提供单独的 IConnectionPoint 接口实现。由于 IConnectionPoint 接口并不是对象实体本身的一部分，所以它不必通过 QueryInterface 方法被暴露给外界。相反，COM 提供了另一个接口，由对象本身把这个接口暴露给客户，使客户能够查询到某个特定的回调接口类型所对应的 IConnectionPoint 实现：

```

[ object,uuid(B196B284-BAB4-101A-B69C-00AA00341D07) ]
interface IConnectionPointContainer : IUnknown {

    // 获取所有可能的 IconnectionPoint 实现
    HRESULT EnumConnectionPoints(
        [out] IEnumConnectionPoints ** ppEnum);

    // 获取 riid 的 IconnectionPoint 实现
    HRESULT FindConnectionPoint(
        [in] REFIID riid,
        [out] IConnectionPoint ** ppCP
    );
}

```

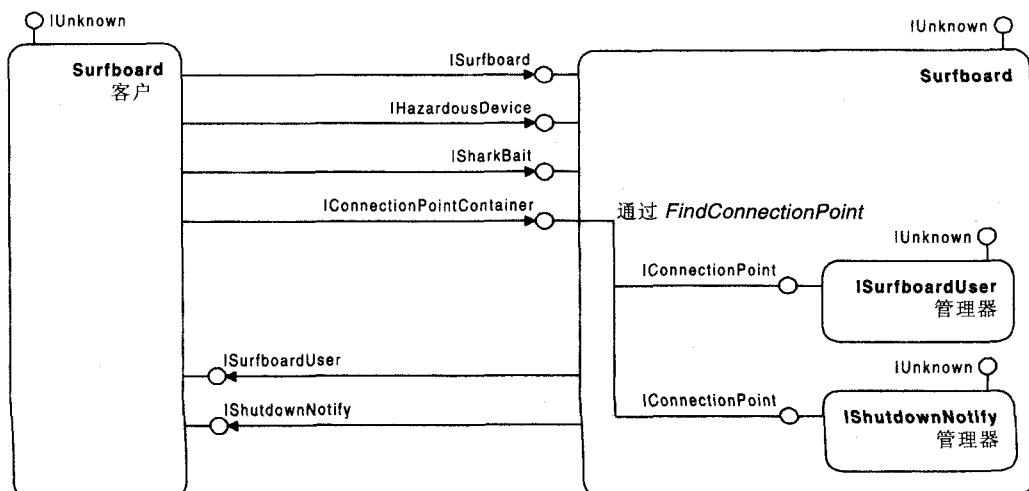


图 7.9 连接点结构

如图 7.9 所示，每个 IConnectionPoint 实现都是从一个单独的 COM 实体上暴露出来的。

有了上面这些接口定义之后，客户就可以把它的 IShutdownNotify 实现与某一个对象关联起来，如下所示：

```

HRESULT HookupShutdownCallback(IUnknown *pUnkObject,
                               ISHUTDOWNNOTIFY *pShutdownNotify,    DWORD &rdwCookie)
{
    IConnectionPointContainer *pcpc = 0;
    HRESULT hr = pUnkObject->QueryInterface(
        IID_IConnectionPointContainer, (void**)&pcpc);
    if (SUCCEEDED(hr)) {
        IConnectionPoint *pcp = 0;
        hr = pcpc->FindConnectionPoint(IID_ISHUTDOWNNOTIFY, &pcp);
        if (SUCCEEDED(hr)) {
            hr = pcp->Advise(pShutdownNotify, &rdwCookie);
            pcp->Release();
        }
        pcpc->Release();
    }
}

```

与此对应的、用来撤销关联的代码如下：

```

HRESULT TeardownShutdownCallback(IUnknown *pUnkObject,
                                 DWORD dwCookie) {
    IConnectionPointContainer *pcpc = 0;
    HRESULT hr = pUnkObject->QueryInterface(
        IID_IConnectionPointContainer, (void**)&pcpc);
    if (SUCCEEDED(hr)) {
        IConnectionPoint *pcp = 0;
        hr = pcpc->FindConnectionPoint(IID_ISHUTDOWNNOTIFY, &pcp);
        if (SUCCEEDED(hr)) {
            hr = pcp->Unadvise(dwCookie);
            pcp->Release();
        }
        pcpc->Release();
    }
}

```

注意，在这两段例子代码中，客户都利用 IConnectionPointContainer::FindConnectionPoint 方法，来查询对应于 ISHUTDOWNNOTIFY 的 IConnectionPoint 实现对象。如果 FindConnection 调用失败了，那么这表明对象并不理解 ISHUTDOWNNOTIFY 接口的语义。这可以防止用户将任意的回调接口附到一个“没有得到实现者完全同意”的对象上。

如同 IUnknown 一样，IConnectionPointContainer 和 IConnectionPoint 的实现绝大部分都是有样板可循的。C++ 对象针对每个它希望支持的对外接口类型，都要求单独的 COM

实体。实现 `IConnectionPoint` 的一种技术是利用嵌套类/复合类的变种技术，请考虑下面对对象实体间的区别：

```

class Surfboard : public ISurfboard,
                  public IHazardousDevice,
                  public ISharkBait,
                  public IConnectionPointContainer {
    LONG m_cRef; // COM 引用计数
    // Surfboards 不支持给定类型的多重对外接口,
    // 因此为每个
    // 回调接口的可能类型声明一个指针
    IShutdownNotify *m_pShutdownNotify;
    ISurfboardUser *m_pSurfer;

    // 处理 IConnectionPoint 的身分关系
    // 定义一个 IshutdownNotify 嵌套成员和类
    class XCPShutdownNotify : public IConnectionPoint {
        Surfboard *This(void); // 使用固定偏移
        // IUnknown 方法...
        // IConnectionPointMethods...
        } m_xcpShutdownNotify;

    // 定义一个 ISurfboardUser 嵌套成员和类
    class XCPsurfboardUser : public IConnectionPoint {
        Surfboard *This(void); // 使用固定偏移
        // IUnknown 方法...
        // IConnectionPointMethods...
        } m_xcpSurfboardUser;
    // IUnknown 方法...
    // ISurfboard 方法...
    // IHazardousDevice 方法...
    // ISharkBait 方法...
    // IConnectionPointContainer 方法...
};


```

注意，`Surfboard` 类的实例将有两个独立的 `IConnectionPoint` 实现，一个用于附着 `IShutdownNotify` 回调接口，另一个用于附着 `ISurfboardUser` 接口。这两个实现被分成单独的 C++ 类，从而使每个 `IConnectionPoint` 实现都有自己独立的 `IUnknown` 和 `IConnectionPoint` 实现。特别地，这里将有三个不同的 `QueryInterface` 实现，它们分别有自己的一组接口指针集合，从而建立起三个不同的 COM 对象实体。

前面的类定义意味着下面的针对 `Surfboard` 主类的 `QueryInterface` 实现：

```

STDMETHODIMP Surfboard::QueryInterface(REFIID riid,
                                         void**ppv) {
    if (riid == IID_IUnknown || riid == IID_ISurfboard)
        *ppv = static_cast<ISurfboard*>(this);
    else if (riid == IID_IHazardousDevice)

```

```

    *ppv = static_cast< IHazardousDevice *>(this);
else if (riid == IID_ISharkBait)
    *ppv = static_cast< ISharkBait *>(this);
else if (riid == IID_IConnectionPointContainer)
    *ppv = static_cast< IConnectionPointContainer *>(this);
else
    return (*ppv = 0), E_NOINTERFACE;
((IUnknown*)*ppv)->AddRef();
return S_OK;
}

```

注意，通过主对象的 `QueryInterface` 实现是无法访问 `IConnectionPoint` 接口的。每个嵌套类的 `QueryInterface` 应类似如下：

```

STDMETHODIMP
Surfboard::XCPShutdownNotify::QueryInterface(REFIID riid,
                                              void d***ppv) {
    if (riid == IID_IUnknown || riid == IID_IConnectionPoint)
        *ppv = static_cast< IConnectionPoint *>(this);
    else
        return (*ppv = 0), E_NOINTERFACE;
    ((IUnknown*) *ppv) ->AddRef();
    return S_OK;
}

```

同样的实现也适用于 `XCPSurfboardUser` 类。注意，在 `Surfboard` 对象和两个实现了 `IConnectionPoint` 接口的子对象之间没有任何实体身份关系。

为确保 `Surfboard` 对象不会提前销毁自己，连接管理器子对象只是简单地把它们的 `AddRef` 和 `Release` 方法传给外面的包容对象 `Surfboard`：

```

STDMETHODIMP_(ULONG)
Surfboard::XCPShutdownNotify::AddRef(void) {
    return This()->AddRef(); /* AddRef 包容对象 */
}

STDMETHODIMP_(ULONG)
Surfboard::XCPShutdownNotify::Release(void) {
    return This()->Release(); /* Release 包容对象 */
}

```

上面的方法实现代码假设 `This` 方法使用某种固定偏移算法，返回一个指向包容对象 `Surfboard` 的指针。

客户调用对象的 `FindConnectionPoint` 方法，就可以发现对象的 `IConnectionPoint` 接口。`Surfboard` 类的 `FindConnectionPoint` 方法实现应如下所示：

```

STDMETHODIMP Surfboard::FindConnectionPoint(REFIID riid,
                                             IConnectionPoint **ppcp) {
    if (riid == IID_IShutdownNotify)

```

```

    *ppcp = &m_xcpShutdownNotify;
else if (riid == IID_ISurfboardUser)
    *ppcp = &m_xcpSurfboardUser;
else
    return (*ppcp = 0), CONNECT_E_NOCONNECTION;
(*ppcp)->AddRef();
return SOK;
}

```

注意，只有当对象接收到“它知道该如何回调”的接口时，对象才提交 `IConnectionPoint` 接口。同时也请注意，它与大多数 `QueryInterface` 接口实现有惊人的相似性。主要的区别在于，`QueryInterface` 协商内部的 (*inbound*) 接口，而 `FindConnectionPoint` 协商对外的 (*outbound*) 接口。

因为 `IConnectionPoint::Advise` 方法只接受静态类型的 `IUnknown` 接口作为回调接口指针，<sup>3</sup> 所以对象的 `Advise` 实现必须调用 `QueryInterface` 方法，将回调指针强制转换为适当类型的接口指针：

```

STDMETHODIMP
Surfboard::XCPShutdownNotify::Advise(IUnknown *pUnk,
                                         DWORD *pdwCookie) {
    assert(pdwCookie && punk);
    *pdwCookie = 0;
    if (ThisO->m_pShutdownNotify) // 已经有一个了
        return CONNECT_E_ADVISELIMIT;
    // 正确回调类型的QueryInterface
    HRESULT hr = pUnk->QueryInterface(IID_IShutdownNotify,
                                         (void**)&(This()->m_pShutdownNotify));
    if (hr == E_NOINTERFACE)
        hr = CONNECT_E_NOCONNECTION;
    if (SUCCEEDED(hr)) // 生成一个有意义的 cookie
        *pdwCookie = DWORD(This()->m_pShutdownNotify);
    return hr;
}

```

前面曾经提到过，`QueryInterface` 隐式地调用了 `AddRef`，这意味着 `Surfboard` 对象拥有的回调引用，在离开了 `Advise` 方法之后仍然有效。也请注意，如果回调对象没有实现适当的接口，则上面方法中的结果 `HRESULT` 被映射为 `CONNECT_E_NOCONNECTION`。如果由于其他某些原因 `QueryInterface` 调用失败，则 `QueryInterface` 返回的 `HRESULT` 被传递给调用者。<sup>4</sup>

<sup>3</sup> 这是连接点机制设计中一个已知的缺陷。另一个广为人知的缺陷是，对于每个回调接口类型，客户都必须要显式地调用 `FindConnectionPoint`。由于这两个缺陷都意味着更多的通信来回，所以它们都影响了性能。使用连接点机制所造成的性能影响就好像是一种暗示，提醒接口应该设计成可以跨套间访问。

<sup>4</sup> 一个可能发生的常见错误是访问被拒绝。当对象企图与回调对象的进程进行通信的时候，由于调用者的访问控制不允许来自对象一方安全个体的调用，这种错误就会发生。

根据以上的 Advise 实现，对应的 Unadvise 方法如下所示：

```
STDMETHODIMP
Surfboard::XCPShutdownNotify::Unadvise(DWORD dwCookie) {
    // 确保对应于有效连接的 Cookie
    if (DWORD(This())->m_pShutdownNotify) != dwCookie)
        return CONNECT_E_NOCONNECTION;
    // 释放连接
    ThisO->m_pShutdownNotify->Release();
    ThisO->m_pShutdownNotify = 0;
    return S_OK;
}
```

IConnectionPoint 接口另外还有三个方法，其中两个方法实现起来非常简单：

```
STDMETHODIMP
Surfboard::XCPShutdownNotify::GetConnectionInterface(
    IID *piid) {
    assert(piid);
    // 返回该子对象管理的接口的 IID
    *piid = IID_IShutdownNotify;
    return S_OK;
}

STDMETHODIMP
Surfboard::XCPShutdownNotify::GetConnectionPointContainer(
    IConnectionPointContainer **ppcpc) {
    assert(ppcpc);
    (*ppcpc = This())->AddRef(); // 返回包含对象
    return S_OK;
}
```

剩下的方法 EnumConnections 允许调用者枚举所有已经建立连接的接口。这个方法是可选的，可以合法地返回 E\_NOTIMPL。

为了让一个实现类宣扬它支持哪个对外接口，IDL 提供了 [source] 属性：

```
[ uuid(315BC280-DEAT-11d0-8C5E-0080C73925BA) ]
coclass Surfboard {
    [default] interface ISurfboard;
    interface IHazardousDevice;
    interface ISharkBait;
    [source] interface IShutdownNotify;
    [source, default] interface ISurfboardUser;
}
```

而且，COM 提供了两个接口，允许运行时环境要求对象返回有关它的内部接口类

型和对外接口类型的信息：

```
[ object, uuid(B196B283-BAB4-101A-B69C-00AA00341D07) ]
interface IProvideClassInfo : IUnknown {
    // 返回对象共类(coclass)的描述信息
    HRESULT GetClassInfo([out] ITypeLib ** ppti);
}

[ object, uuid(A6BC3AC0-DBAA-11CE-9DE3-00AA004BB851) ]
interface IProvideClassInfo2 : IProvideClassInfo {
    typedef enum tagGUIDKIND {
        GUIDKIND_DEFAULT_SOURCE_DISP_IID = 1
    } GUIDKIND;
    // 返回缺省对外分发接口
    HRESULT GetGUID([in] DWORD dwGuidKind,
                    [out] GUID * pGUID) ;
}
```

实现这两个接口并不困难：

```
STDMETHODIMP Surfboard::GetClassInfo(ITypeLib ** ppti) {
    assert(ppti != 0);
    ITypeLib * pt1 = 0;
    HRESULT hr = LoadRegTypeLib(LIBID_BeachLib, 1, 0, 0, &pt1);
    if (SUCCEEDED(hr)) {
        hr = pt1->GetTypeLibOfGuid(CLSID_Surfboard, ppti) ~
        pt1->Release();
    }
    return hr;
}
STDMETHODIMP Surfboard::GetGUID(DWORD dwKind, GUID * pguid) {
    if (dwKind != GUIDKIND_DEFAULT_SOURCE_DISP_IID || !pguid)
        return E_INVALIDARG;
    // ISurfboardUser 定义为分发接口
    *pguid = IID_ISurfboardUser;
    return S_OK;
}
```

虽然对外的接口并不要求一定是分发接口（dispatch interface，即 IDispatch 接口），但是有些脚本环境确实有这样的要求，目的是为了能够把回调映射到脚本代码中。

假设 ISurfboardUser 接口被定义为如下所示的 dispinterface：

```
[ uuid(315BC28A-DEA7-11d0-8C5E-0080C73925BA) ]
dispinterface ISurfboardUser {
methods:
    [id(1)] void OnTiltingForward([in] long nAmount);
    [id(2)] void OnTiltingSideways([in] long nAmount);
}
```

Visual Basic 程序员可以声明如下“能够理解缺省回调类型”的变量：

```
Dim WithEvents sb as Surfboard
```

这个变量定义的出现使 Visual Basic 程序员可以编写事件控制函数。Visual Basic 事件控制函数只是一些简单的函数或者子过程，其名字为 VariableName\_EventName。例如，为了处理上述 sb 变量的 OnTiltingForward 回调，Visual Basic 程序员可以编写以下的代码：

```
Dim WithEvents sb as Surfboard
Sub sb_OnTiltingForward(ByVal nAmount as Long)
    MsgBox "The surfboard just tilted forward"
End Sub
```

Visual Basic 虚拟机实际上将会动态地构造出一个 ISurfboardUser 实现，把进来的方法调用映射到适当的用户自定义函数上。

## 7.7 IDL 中的别名技术

我们通常有必要把传统遗留下来的数据类型和程序代码集成到一个以 COM 为基础的系统中。理想情况下，最好有一种简单有效的办法可以把遗留下来的代码映射到 IDL 兼容的对应物上面。如果确实是这样的话，那么过渡到 COM 就可以直截了当地进行。然而，在许多情况下，一个应用的遗留数据类型和代码并不能简单合理地被映射到 IDL 中。为了解决这样的问题，IDL 提供了几种别名（aliasing）技术，允许接口设计者提供转换过程，由这些转换过程把遗留的数据类型和代码映射到合法的、可远程化的 IDL 表示。

有一个很好的例子可以说明这项技术的用途，这就是 IEnum 系列接口。COM 的枚举器习惯用法是在可感知 COM 的 IDL 编译器之前就已经被设计好的。这意味着 IEnum 接口的原始设计者并不能够检验接口设计是否符合现在已知的 IDL 映射规则。枚举器的 Next 方法并不能被清晰地映射到 IDL 上。<sup>5</sup> 考虑以下 Next 方法的 IDL 原型：

```
HRESULT Next([in] ULONG cElems,
             [out, size_is(cElems), length_is(*pcFetched)] double *prg,
             [out] ULONG *pcFetched);
```

<sup>5</sup> 可能有人会争辩说，原来的接口定义是合理的，IDL 不够灵活，所以无法描述这类常见的程序设计习惯用法。尽管这种说法对于 1992 年在 COM IDL 之前定义的接口，可能是有效的辩护，但是对于今天定义的接口，这种说法是不公正的。简而言之，所有的接口都应该遵从 COM IDL 的规则，除非有非常强烈的特殊理由。

不幸的是，原来在 IDL 之前的接口定义中 Next 方法声明：调用者可以传递一个 null 指针作为第三个参数，并假设第一个参数指示只有一个元素被请求。这使调用者可以很方便地一次获取一个元素：

```
double dblElem;
hr = p->Next(1, &dblElem, 0);
```

不幸的是，接口的这种用法虽然在以前是合法的，但是违反了前面的 IDL 定义，因为顶级[out]参数被置成 null 是非法的（接口代理没有空间用来保存结果）。为了解决这种不一致性，每个 Next 方法定义必须使用[call\_as]属性，以便把方法的可调用形式（callable form）变成远程形式（remotable form）。

[call\_as]属性允许接口设计者以两种形式表达一个方法。方法的可调用形式必须使用[local]属性，以便禁止产生列集代码。这个版本的方法可以保证“客户将以什么形式调用方法”以及“对象将以什么形式实现方法”完全一致。方法的远程形式必须使用[call\_as]属性，以便把产生的列集器与接口存根中适当的方法联系起来。这个版本描述了接口的远程形式，并且必须使用标准的 IDL 语法来描述“远程调用该方法所要求的请求和响应消息”。为了将[call\_as]技术应用到 Next 方法上，IDL 应如下所示：

```
interface IEnumDouble : IUnknown {
// 此方法可以为调用者和对象看见
    [local] HRESULT Next([in] ULONG cElems,
                        [out] double *prgElems,
                        [out] ULONG *pcFetched);

// 方法的远程形式
    [call_as(Next)] HRESULT RemoteNext([in] ULONG cElems,
                                       [out, size_is(cElems), length_is(*pcFetched)] double *prg,
                                       [out] ULONG *pcFetched);
    HRESULT Skip([in] ULONG cElems);
    HRESULT Reset(void);
    HRESULT Clone([out] IEnumDouble **ppe);
```

结果得到的 C/C++头文件将包含一个接口定义，该接口包含 Next 方法，但没有 RemoteNext 方法定义。在我们目前所关心的客户和对象中，并没有 RemoteNext 方法。它的存在只是为了使接口列集器能够正常地远程调用方法。尽管 Next 和 RemoteNext 方法有同样的参数列表，但是这并不意味着在使用这项技术时必须要这样做。事实上，有时候为了充分表达某个操作应如何被远程使用，我们必须要在方法的远程形式中增加额外的参数。

加入了[local]/[call\_is]方法对之后，所产生的接口列集器源代码不再能够被成功地链接了，原因是存在未确定的外部符号。这是因为接口设计者必须提供两个额外的函数。其中一个函数将被接口代理用于把[local]形式的方法变换成[call\_is]形式。对于上面的接

口定义，IDL 编译器期望接口设计者提供下面的函数：

```
HRESULT STDMETHODCALLTYPE IEnumDouble_Next_Proxy(
    IEnumDouble *This, ULONG cElems,
    double *prg, ULONG *pcFetched);
```

第二个函数被接口存根用来把[call\_is]形式的方法变换为[local]形式。对于前面同样的接口定义，IDL 编译器期望接口设计者提供下面的函数：

```
HRESULT STDMETHODCALLTYPE IEnumDouble_Next_Stub(
    IEnumDouble *This, ULONG cElems,
    double *prg, ULONG *pcFetched);
```

为方便起见，这两个函数的原型都出现在 IDL 编译器产生的 C/C++ 头文件中。

如图 7.10 所示，用户定义的[local]-to-[call-as]函数被用来填充接口代理的 vtable，并被客户调用。这个函数应该调用由 IDL 编译器产生的远程版本，把客户的调用翻译成一个远程调用。对于枚举器的 Next 函数，所有要做的工作就是确保第三个参数不是一个 null 指针，代码如下：

```
HRESULT STDMETHODCALLTYPE IEnumDouble_Next_Proxy(
    IEnumDouble *This, ULONG cElems,
    double *prg, ULONG *pcFetched) {
    // 应用在客户方
    if (pcFetched == 0 && cElems != 1)
        return E_INVALIDARG;
    // 为最后一个 [out] 参数提供一个位置
    ULONG cFetched;
    if (pcFetched == 0) pcFetched = &cFetched;
    // 调用最后一个参数为非空指针的远程方法
    return IEnumDouble_RemoteNext_Proxy(This, cElems,
                                         prg, pcFetched);
}
```

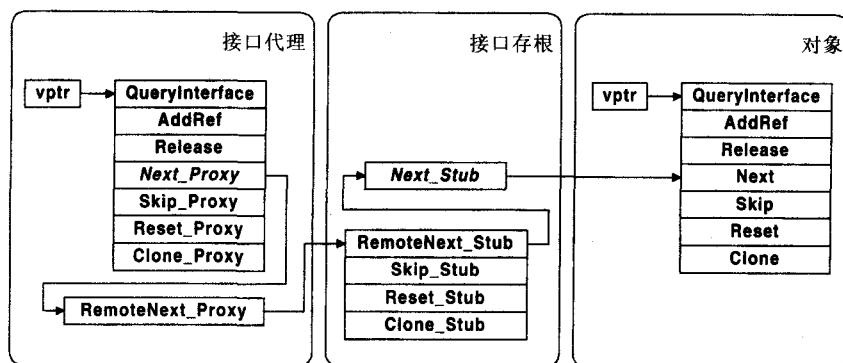


图 7.10 方法别名技术

注意，在所有的情况下，方法的远程版本都要接收一个非 null 指针作为最后一个参数。

用户定义的[call\_as]-to-[local]函数将被接口存根在散集了远程版本的方法之后调用。这个函数应该把远程形式的调用翻译成对实际对象的本地调用。由于对象的实现代码有时候不够完善，它在返回 S\_OK 的时候有所疏忽，没有指示出有多少个对象被返回了，所以对象一端的映射函数可以确保这个参数一定被正确设置：

```

HRESULT STDMETHODCALLTYPE IEnumDouble_Next_Stub(
    IEnumDouble *This, ULONG cElems,
    double *prg, ULONG *pcFetched) {
    // 对实际对象 A 调用方法
    HRESULT hr = This->Next(cElems, prg, pcFetched);
    // 应用在客户方
    if (hr == S_OK) // S_OK 暗示所有元素已发送
        *pcFetched = cElems; // [length_is] 必须是显式的
    return hr;
}

```

注意，接口存根调用这个函数的时候，其最后一个参数总是非 null 的。

[call\_as]技术对于在方法基础上（method-by-method basis）提供“可调用形式-to-远程形式”之间的转换是非常有用的。COM 还提供了针对单个数据类型的转换能力，通过[transmit\_as]和[wire\_marshal] typedef 属性指定用户自定义的转换。所有这三种技术都不应该被认为是主流的接口设计技术，它们的存在很大程度上是为了支持遗留代码和遗留数据类型。IDL 提供的另一个 hook（钩子）技术是 cpp\_quote。cpp\_quote 关键字允许任意的 C 或者 C++语句出现在 IDL 文件中，即使该语句在 IDL 中不是合法的。考虑下面的 cpp\_quote 简单用法，它把一个内联函数定义插入到 IDL 产生的头文件中：

```

// surfboard.idl
cpp_quote("static void Exit(void) { ExitProcess(1); }")

```

有了这个 IDL 之后，产生出来的 C/C++头文件将包含下面的语句：

```

// surfboard.h
static void Exit(void) { ExitProcess(1); }

```

有了 cpp\_quote 关键字之后，我们就可以在 IDL 编译器上实现各种技巧。一个例子是数据类型 REFIID。这个类型真正的 IDL 定义是：

```
typedef IID *REFIID;
```

然而，C++中的类型被定义为：

```
typedef const IID& REFIID;
```

然而，C++风格的引用在 IDL 中是不允许的。为了解决这个问题，系统 IDL 文件使

用了下面的技术：

```
// 摘自 wtypes.idl (近似)
cpp_quote("#if 0")
typedef IID *REFIID; // 纯 IDL 定义
cpp_quote("#endif")
cpp_quote("#ifdef __cplusplus")
cpp_quote("#define REFIID const IID&") // C++ 定义
cpp_quote("#else")
cpp_quote("#define REFIID const IID * const")// C 定义
cpp_quote("#endif")
```

结果得到的 C/C++ 头文件看起来如下所示：

```
// 摘自 wtypes.h (近似)
#if 0
    typedef IID *REFIID;
#endif
```

这种多少有点怪异的做法是很有必要的，因为许多核心 COM 接口在被设计的时候，根本就没有考虑到 IDL。

## 7.8 异步方法

缺省情况下，COM 方法调用是同步的。这意味着在客户线程接收到 ORPC 消息并把它散集出来之前，客户线程一直被阻塞。这种模型非常适合于同一个线程中的方法调用，并且在缺省情况下也是合理的。在 Windows NT 5.0（即 Windows 2000）之前，我们在发出一个方法请求之后，如果不显式地产生其他附加的线程，那么在方法执行期间就无法继续处理其他的事情。Windows NT 5.0（即 Windows 2000）发布的 COM 引入了对异步方法调用的支持。异步方式成为方法的一个属性，在 IDL 中我们可以使用 [async\_uuid] 属性来表达。

在本书写作时，这项技术的细节尚未最后确定。请参考最新的技术文档以获取更多的信息。

## 7.9 我们走到哪儿了？

本章讨论了许多有关设计和使用 COM 接口的话题。尽管本章并没有包含所有有用

的设计方法，但是本章确实企图解决一些在前面章节中没有讨论到的重要话题。在我写作本书的两年时间里，我个人对 COM 的理解也在不断进步，我越来越相信，开发人员对于 COM 中深奥的设施（比如连接点、名字对象、分发接口）不必付出太多的精力去钻研，而应该将重点放在 COM 的三个基本元素上：接口、类对象和套间。如果你对这三个概念有了彻底的理解，那么我坚信，通过使用 COM，没有翻越不过去的山头。

## 附录 A

### 对象技术的演变

本附录的一个缩写版本曾登载在 1998 年 1 月份的《Microsoft System Journal》杂志上。之所以要把它作为附录收入本书中，是因为它展示了 COM 的历史远景。

面向对象软件开发在 80 年代晚期得到了商业界的广泛接受。在 80 年代，面向对象的中心思想集中在“对类的使用”上，于是开发人员把状态和行为建模成为单一的抽象单元。状态和行为的这种绑定有助于“通过使用封装手段实现模块化”。在经典的面向对象中，对象属于类，客户通过“以类为基础的引用”来操纵对象。这是大多数 C++ 和 Smalltalk 环境以及各种函数库所隐含的编程模型。虽然使用基于过程的语言并利用严格的编程风格，我们也有可能获得“基于类的编程”所具备的许多好处，但是在开发工具和语言厂商明确地支持面向对象之前，面向对象的软件开发并没有得到广泛的接受。对于面向对象的成功起关键作用的开发环境包括 Apple 的 MacApp 框架（以 Object Pascal 为基础），ParcPlace 和 Digitalk 早期的 SmallTalk 开发环境，以及 Borland 公司的 Turbo C++。

使用这种显式支持面向对象开发环境的主要好处在于，如果一组类似的对象都与另一个类型兼容，那么我们可以对这组对象使用多态性。为了支持多态性，面向对象引入了继承（inheritance）和动态绑定（dynamic binding）的概念，允许相似的类能够被显式地组织到相关的抽象集合中。考虑下面简单的 C++ 类层次体系：

```
class Dog {  
public:
```

```

    virtual void Bark(void);
};

class Pug : public Dog {
public:
    virtual void Bark(void);
};

class Collie : public Dog {
public:
    virtual void Bark(void);
};

```

因为类 Collie 和 Pug 都与 Dog 类型兼容，所以客户可以写出如下所示的一般性代码：

```

void BarkLikeADog(Dog& rdog) {
    rdog. Bark();
}

```

由于 Bark 方法是虚拟的，并且是动态绑定的，所以 C++ 的方法分发机制可以确保正确的代码会被执行。这意味着 BarkLikeADog 函数并不依赖于被引用对象的精确类型，只要它与 Dog 类型兼容即可。这个例子很容易被转换到其他支持面向对象的编程语言中。

前面的类层次结构体现了第一波面向对象浪潮中的实用技术。在这个浪潮中占主导地位的一个特征是“实现继承（implementation inheritance）”的使用。在具有严格的编程风格的环境中，实现继承是一项强有力的编程技术。然而，如果误用了这项技术的话，则在结果类型层次中，基类和派生类之间可能会出现过度的耦合。关于这种耦合关系一个常见的情形是，针对某个方法，我们往往不清楚派生类的版本是否必须要调用基类的实现。例如，考虑 Pug 类的 Bark 实现：

```

void Pug::Bark(void) {
    this->BreathIn();
    this->ConstrictVocalChords();
    this->BreathOut();
}

```

假如像上面的代码所示，底层 Dog 类中 Bark 的实现没有被调用到，那么事情会怎么样呢？有可能基类的方法要记录下某个对象被调用的次数（即狗叫的次数）以备将来有用？如果是这样的话，那么 Pug 类违反了底层 Dog 实现类的部分语义。为了正常地使用实现继承，必须要有足够的内部知识来维持底层基类的一致性或者完整性。这些内部细节知识超过了作为简单客户所应该具备的知识。由于这个原因，实现继承通常被视为白盒重用。

在面向对象思想体系中，既要保持多态性的好处，又要降低过度的类型系统耦合关系，一种方法就是只继承类型特征，不继承实现代码。这正是“基于接口的软件开发”

背后的基本原则，可以被视为面向对象的第二波浪潮。基于接口的程序设计是传统面向对象实践的一种精练，它假设继承主要是表达类型关系的一种机制，而不是实现层次的机制。基于接口的软件开发建立在“接口（interface）与实现（implementation）分离”的基础上。在基于接口的软件开发中，接口与实现是两个不同的概念。接口构造出抽象请求的模型，而对象则是由这些请求组成的。实现则构造出“可实例化的实体类型”的模型，它可以支持一个或者多个接口。虽然使用传统的、第一波浪潮中的开发环境并利用严格的编程风格，我们也有获得许多“基于接口的软件开发”的好处，但是在开发工具和语言厂商明确地提供支持之前，“基于接口的软件开发”并没有得到广泛的接受。对于“基于接口的软件开发”的成功起关键作用的开发环境包括 Microsoft 的组件对象模型（COM）、Iona 的 Orbix ORB（Object Request Broker，对象请求代理）环境，以及 Java 对于“基于接口的软件开发”的显式支持。<sup>1</sup>

使用这种支持“基于接口的软件开发”的环境的关键好处在于，它建立起来的模型能够把“对象是什么”和“对象如何实现”看作两个不同的概念。考虑下面非常简单的 Java 类型层次体系：

```
interface IDog {
    public void Bark();
};

class Pug implements IDog {
    public void Bark() { ... }
};

class Collie implements IDog {
    public void Bark() { ... }
};
```

因为类 Collie 和 Pug 都与接口 Dog 类型兼容，所以客户可以写出如下所示的一般性代码：

```
void BarkLikeADog(IDog dog) {
    dog.Bark();
}
```

从客户的观点来看，这个类型层次基本上等同于前面基于 C++的例子。然而，由于 IDog 接口的 Bark 方法不可能具有实现，所以在 IDog 接口定义和 Pug 或者 Collie 类之间不存在耦合关系。虽然这确实意味着 Pug 和 Collie 都必须完全定义出它们自己对于 Bark 方法的看法，但是 Pug 和 Collie 的实现者不需要考虑派生类会有什么副作用强加在底层 IDog 基类上。

在第一波和第二波之间，一个很明显的相似之处是：一个简单的概念就可以刻划出

---

<sup>1</sup> 技术上讲，Java 不同于 COM 和几乎所有的 CORBA 产品（例如 Iona 的 Orbix、IBM 的 DSOM 以及 BEA 的 Object Broker），它还支持传统的实现继承。

它们的特征（分别为类和接口）。在两种情况下，概念本身并不是成功的催化剂，相反，一个或者多个关键的开发环境才真正激起了软件工业的极大兴趣。

第二波系统中一个很有趣的情况是，实现被看作是一个黑盒子；也就是说，对于对象的客户而言，所有的实现细节都被认为是不透明的。通常，当开发人员开始使用基于接口的技术（比如 COM）的时候，这种不透明性所提供的自由度被忽略了，从而使得新手可以非常简单地看待接口、实现和对象之间的关系。请考虑 Excel 电子表格的例子，它通过 COM 暴露自己的功能。Excel 的电子表格实现类大约暴露了 25 个不同的 COM 接口，这些接口允许电子表格对象参与到各种以 COM 为基础的技术中（链接[Linking]、嵌入[Embedding]、实地激活[Inplace Activation]、自动化[Automation]、Active 文档对象[Active Document Object, ADO]、超链接[Hyperlinking]，等等）。由于每个接口在每个对象上都要求 4 字节的虚函数指针（vptr），所以这意味着电子表格对象除了要维护内部用于记录用户数据的状态信息以外，还需要大约 100 字节的内存负担。由于给定的电子表格对象可能包含大量的表格单元，对于大型的电子表格，它可能要管理所有正在被使用中的每个单元，所以这 100 字节的负担被分摊到几百 KB 的电子表格数据中，也就不足一提了。

Excel 电子表格的实际实现是非常复杂的，因为电子表格中每个独立的单元都可以通过 COM 接口被客户访问到。从 COM 的观点来看，这些单元接口（cell interface）每个都属于不同的 COM 实体，通过电子表格对象的 QueryInterface 调用是不能发现这些接口的。相反，必须要使用电子表格对象暴露给客户的其他接口（比如 IOleItemContainer）才能够发现这些单元接口。“每个单元都被通过 COM 接口的形式暴露给客户”这一事实意味着，Excel 的实现者必须小心处理，以避免过多的与 COM 相关的负担。请考虑一个包含 1000 个单元的电子表格对象。假设粗略估计一下，每个单元平均要求 16 字节的内存用来保存 Excel 本身的单元状态。这意味着 1000 单元的电子表格对象需要消耗大约 16000 字节内存，这些内存与 COM 无关。对于这个电子表格，电子表格对象本身所需要的 100 字节虚函数指针内存对于内存消耗的影响很小。然而，由于每一个单元本身大约要暴露 8 个不同的接口，所以这意味着，为每个单元管理其虚函数指针就需要消耗 32 字节与 COM 相关的内存。假设大多数 COM 开发环境都使用如此原始的实现技术，那么这 1000 个单元的电子表格将需要大约 32100 字节的内存用于存放虚函数指针，这差不多是 Excel 本身数据所需内存的两倍。很显然，这样的代价太大了。

为了理解 Excel 开发组如何解决 vptr 负担的问题，我们最好重新考虑状态和行为之间的关系，看一看 COM 是如何实现这种关系的。图 A.1 显示了一个简单的 COM 对象在内存中的情况。注意，对象所占有的内存块既包含 vptr，也包含数据成员。考察这个图的一种方法是，认为数据成员代表了对象的状态，而虚函数指针（以及它们所对应的虚函数表）代表了对象的行为。在大多数的对象实现中，这两部分对象信息驻留在连续的内存块中。然而，COM 并没有这样的要求。COM 只处理 vptr，而把状态管理的任务留给实现者。如果实现者决定把对象的状态和 vptr 分配在不同的内存块中，如图 A.2 所

示，COM 不会有任何异议。毕竟，“如何管理对象的状态”只不过是另一个实现细节而已，应该被隐藏到对象的接口层后面，客户不需要知道这些细节知识。

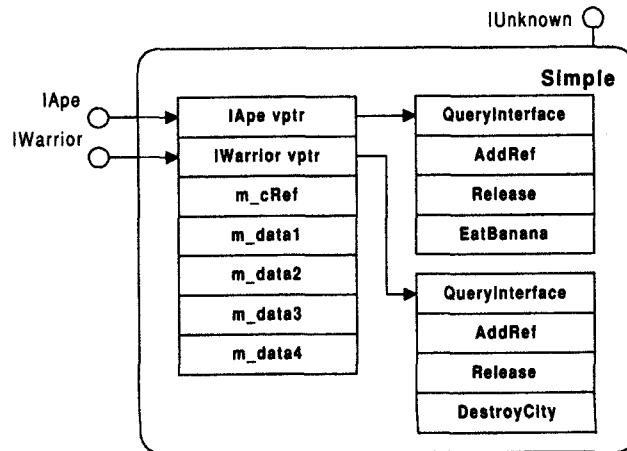


图 A.1 一个简单的 COM 对象

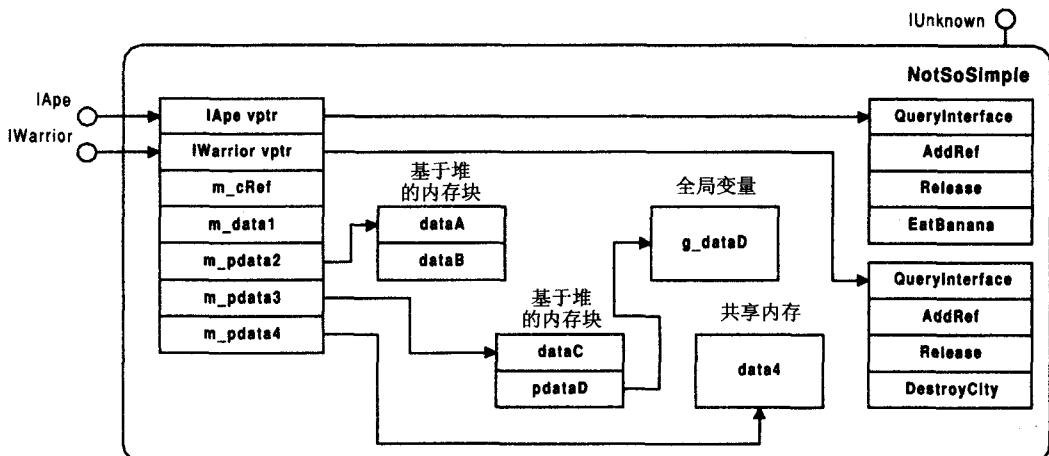


图 A.2 一个不那么简单的 COM 对象

由于 COM 并不要求对象的状态与 vptr 被分配在连续的内存中，所以 Excel 开发组能够相当可观地优化电子表格的内存消耗。考虑单个电子表格单元的情形。虽然用来保存单元内容的 16 字节内存是必须要被分配的，但是用于 vptr 的 32 字节内存并不需要与单元数据驻留在连续的内存块中。而且，除非要通过 COM 接口访问这个单元，否则这 32 字节的 vptr 内存根本就不需要。这意味着 Excel 可以动态地根据需要为单元分配 vptr 内存，分配内存的时候可以以每个单元为基础进行。由于大多数单元永远也不会被通过 COM 接口访问，所以在大多数情况下几乎没有 vptr 内存消耗。这种建立“次最轻量级（flyweight）对象”的方法可以根据需要提供行为，它也是 tearoff 技术的变种，tearoff

技术最初出现在 Crispin Goswell 所写的一本非常优秀的书《COM Programmer's Cookbook》(<http://www.microsoft.com/oledev>) 中。这两项技术都使用“lazy evaluation”的方法延迟分配 vptr 内存。

flyweight 和 tearoff 技术都是 COM 开发技术，但是 COM 并不强行或者显式支持这两种技术。相反，这些技术都来源于“有效管理状态”的需求。我们在把 COM 应用到分布式应用开发中时，其他一些与状态管理有关的问题也随之产生，包括分布式错误恢复、安全性、并发管理、负载平衡和数据一致性等。不幸的是，COM 对于“对象如何管理它的状态”一无所知，所以 COM 对于这些问题几乎无能为力。虽然开发人员有可能自己编制一套方案来处理状态管理，但如果有一个通用的基础设施，可使我们能够在“具有状态意识”的情况下开发对象，则必然会有显著的好处。Microsoft Transaction Server (MTS) 就是这样一个基础设施。

COM 编程模型扩展了传统的面向对象编程模型，强迫开发人员考虑接口与实现之间的关系。MTS 编程模型扩展了 COM 编程模型，它强迫开发人员也要考虑状态和行为之间的关系。MTS 的基本思想是，从逻辑上看，一个对象是状态和行为组合在一起的模型，但从物理实现上，要把它明确地分开。让 MTS 来管理对象的状态，应用开发人员就可以充分利用 MTS 对于“并发性和锁的管理、错误恢复、数据一致性和细粒度的访问控制”的支持。这意味着对象的大多数状态并没有被保存在与 vptr (代表了对象的行为) 连续的内存空间中。相反，MTS 提供了一些设施，可以把对象的状态保存在永久的或者短时的存储介质中。这些存储介质在 MTS 运行环境的控制之下，对象的方法可以安全地访问它们，而无须考虑锁的管理或者数据一致性问题。有的对象要求在机器失败或者程序非正常终止的情况下，其状态必须保持一致，那么这样的状态可以被保存在永久存储介质中，MTS 会保证通过网络发生的所有修改变化都是原子性操作。短时状态可以被保存在由 MTS 管理的内存中，MTS 会保证这些内存的访问按序进行，避免数据被破坏。

如同基于类和基于接口的开发一样，MTS “具有状态意识”的编程模型要求更多的训练和培养，并要求开发人员更多的关注。幸运的是，与基于类和基于接口的开发一样，MTS “具有状态意识”的编程模型也可以被逐步地接受。当然，逐步接受意味着 MTS 的好处也将会逐渐地体现出来。这允许开发人员可以以“符合本地开发文化的步调”来应用 MTS。

由于 Microsoft 内部 MTS 和 COM 开发组的合并，所以很显然 MTS 代表了 COM 演变过程的下一个步伐。我热诚鼓励所有的 COM 开发人员尽快进入到面向对象开发的第三波浪潮中。

## 附录 B

---

### 代 码 摘 录

本书配套的源代码包含一个完整的 COM 应用（COM Chat），以及作者自己使用的一个工具代码库。这些源代码都可以从 <http://www.develop.com/essentialcom> 页面上下载获得。为方便起见，本附录列出了 COM Chat 应用的代码。

#### COM Chat：一个基于 COM 的聊天程序

COM Chat 是一个完全基于 COM 的程序，它实现了一个多话题的分布式聊天应用。该应用由三个二进制组件组成：comchat.exe 是聊天服务器；comchatps.dll 是所有 COM Chat 接口的接口列集器；client.exe 是一个基于控制台的客户应用。该应用以单个 COM 类（`CLSID_ChatSession`）为基础。正如图 B.1 所示，类对象实现了 `IChatSessionManager` 接口，每个聊天会话对象实现了 `IChatSession` 接口。如果客户希望接收到聊天通知，那么它必须给聊天会话对象提供一个 `IChatSessionEvents` 接口。

##### comchat.idl

```
// COMChat.idl
```

```

interface IChatSessionEvents;

[
    uuid(5223A050-2441-11d1-AF4F-0060976AA886),
    object
]
interface IChatSession : IUnknown
{
    [propget] HRESULT SessionName([out, string] OLECHAR **ppwsz);
    HRESULT Say([in, string] const OLECHAR *pwszStatement);
    HRESULT GetStatements([out] IEnumString **ppes);
    HRESULT Advise([in] IChatSessionEvents *pEventSink,
                  [out] DWORD *pdwReg);
    HRESULT Unadvise([in] DWORD dwReg);
}

[
    uuid(5223A051-2441-11d1-AF4F-0060976AA886),
    object
]
interface IChatSessionEvents : IUnknown
{
    HRESULT OnNewUser([in, string] const OLECHAR *pwszUser);
    HRESULT OnUserLeft([in, string] const OLECHAR *pwszUser);
    HRESULT OnNewStatement([in, string] const OLECHAR *pwszUser,
                          [in, string] const OLECHAR *pwszStmnt);
}

[
    uuid(5223A052-2441-11d1-AF4F-0060976AA886),
    object
]
interface IChatSessionManager : IUnknown
{
    HRESULT GetSessionNames([out] IEnumString **ppes);
    HRESULT FindSession([in, string] const OLECHAR *pwszName,
                       [in] BOOL bDontCreate,
                       [in] BOOL bAllowAnonymousAccess,
                       [out] IChatSession **ppcs);
    HRESULT DeleteSession([in, string] const OLECHAR *pwszName);
}

cpp_quote("DEFINE_GUID(CLSID_ChatSession,0x5223a053,0x2441,")
cpp_quote("0x11d1,0xaf,0x4f,0x0,0x60,0x97,0x6a,0xa8,0x86);")

```

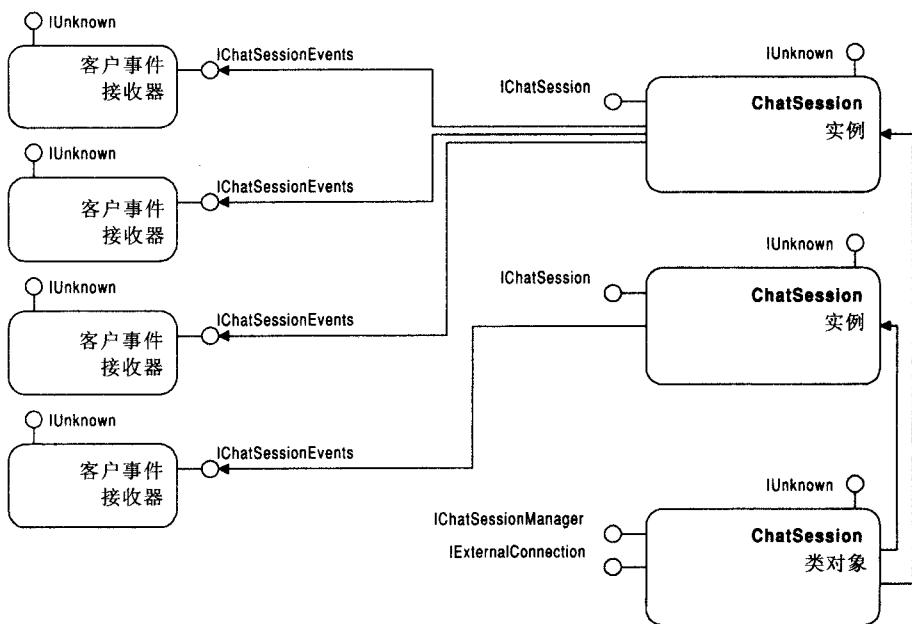


图 B.1 COM Chat

## client.cpp

```

// client.cpp

#define _WIN32_WINNT 0x403
#include <windows.h>
#include <stdio.h>
#include <initguid.h>
#include <wchar.h>
// include IDL-generated header files
#include "../include/COMChat.h"
#include "../include/COMChat_i.c"

void Error(HRESULT hr, const char *psz)
{
    printf("%s failed and returned 0x%x\n", psz, hr);
}

// utility function to print command line syntax
int Usage(void)
{
    const char *psz =
        "usage: client.exe <action> <user> <host>\n"
}
  
```

```

"   where:\n"
"       action = /sessions|/chat:session|/delete:session\n"
"       user = /user:domain\\user /password:pw |"
"           "/anonymous | <nothing>\n"
"       host = /host:hostname | <nothing>\n";
printf(psz);
return -1;
}

// utility function for printing a list of strings
void PrintAllStrings(IEnumString *pes)
{
    enum { CHUNKSIZE = 64 };
    OLECHAR *rgpwsz[CHUNKSIZE];
    ULONG cFetched;
    HRESULT hr;
    do
    {
        hr = pes->Next(CHUNKSIZE, rgpwsz, &cFetched);
        if (SUCCEEDED(hr))
        {
            for (ULONG i = 0; i < cFetched; i++)
                if (rgpwsz[i])
                {
                    wprintf(L"%s\n", rgpwsz[i]);
                    CoTaskMemFree(rgpwsz[i]);
                }
        }
    } while (hr == S_OK);
}

// utility function to print initial state of a chat session
void PrintToDate(IChatSession *pcs)
{
    IEnumString *pes = 0;
    HRESULT hr = pcs->GetStatements(&pes);
    if (SUCCEEDED(hr))
    {
        PrintAllStrings(pes);
        pes->Release();
    }
}

// this class implements the callback interface
// that receives chat notifications. It simply
// prints the event to the console
class EventSink : public IChatSessionEvents
{
public:

```

```

STDMETHODIMP QueryInterface(REFIID riid, void**ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IChatSessionEvents*>(this);
    else if (riid == IID_IChatSessionEvents)
        *ppv = static_cast<IChatSessionEvents*>(this);
    else
        return (*ppv = 0), E_NOINTERFACE;
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}
STDMETHODIMP_(ULONG) AddRef(void)
{
    return 2;
}
STDMETHODIMP_(ULONG) Release(void)
{
    return 1;
}
STDMETHODIMP OnNewStatement(const OLECHAR *pwszUser,
                           const OLECHAR *pwszStmt)
{
    wprintf(L"%-14s: %s\n", pwszUser, pwszStmt);
    return S_OK;
}
STDMETHODIMP OnNewUser(const OLECHAR *pwszUser)
{
    wprintf(L"\n\n>>> Say Hello to %s\n\n", pwszUser);
    return S_OK;
}
STDMETHODIMP OnUserLeft(const OLECHAR *pwszUser)
{
    wprintf(L"\n\n>>> Say Bye to %s\n\n", pwszUser);
    return S_OK;
}
};

// type of operations this client can perform
enum ACTION
{
    ACTION_NONE,
    ACTION_CHAT,
    ACTION_DELETE_SESSION,
    ACTION_LIST_SESSION_NAMES,
};

// run chat command
void Chat(const OLECHAR *pwszSession,
          IChatSessionManager *pcsm, // manager

```

```

COAUTHIDENTITY *pcai,      // user
    bool bAnonymous)           // anonymous
{
// create or get the named session
IChatSession *pcs = 0;
HRESULT hr = pcsm->FindSession(pwszSession, FALSE,
                                TRUE, &pcs);
if (SUCCEEDED(hr))
{
// adjust security blanket for session interface
if (!bAnonymous)
    hr = CoSetProxyBlanket(pcs, RPC_C_AUTHN_WINNT,
                           RPC_C_AUTHZ_NONE, 0,
                           RPC_C_AUTHN_LEVEL_PKT,
                           RPC_C_IMP_LEVEL_IDENTIFY,
                           pcai, EOAC_NONE);

// catch up on past messages
PrintToDate(pcs);
// hook up event sink to receive new messages
EventSink es;
DWORD dwReg;
hr = pcs->Advise(&es, &dwReg);
if (SUCCEEDED(hr))
{
// run UI loop to get statements from console and send them
OLECHAR wszStmt[4096];
while (_getws(wszStmt))
{
    hr = pcs->Say(wszStmt);
    if (FAILED(hr))
        Error(hr, "Say");
}
// tear down connection for event sink
pcs->Unadvise(dwReg);
}
else
    Error(hr, "Advise");
// release chat session
pcs->Release();
}
else
    Error(hr, "FindSession");
}

// run delete command
void Delete(const OLECHAR *pwszSession,
            IChatSessionManager *pcsm)
{
HRESULT hr = pcsm->DeleteSession(pwszSession);

```

```

    if (FAILED(hr))
        Error(hr, "DeleteSession");
}

// run list command
void List(IChatSessionManager *pcsm)
{
    IEnumString *pes = 0;
    HRESULT hr = pcsm->GetSessionNames(&pes);
    if (SUCCEEDED(hr))
    {
        printf("Active Sessions:\n");
        PrintAllStrings(pes);
        pes->Release();
    }
}

int main(int argc, char **argv)
{
// declare client control state
    bool bAnonymous = false;
    static OLECHAR wszSessionName[1024];
    static OLECHAR wszDomainName[1024];
    static OLECHAR wszUserName[1024];
    static OLECHAR wszPassword[1024];
    static OLECHAR wszHostName[1024];
    COSERVERINFO csi = { 0, wszHostName, 0, 0 };
    COSERVERINFO *pcsi = 0;
    COAUTHIDENTITY cai =
    {
        wszUserName,
        0,
        wszDomainName,
        0,
        wszPassword,
        0,
        SEC_WINNT_AUTH_IDENTITY_UNICODE
    };
    static COAUTHIDENTITY *pcai = 0;
    static ACTION action = ACTION_NONE;

// parse command line
    for (int i = 1; i < argc; i++)
    {
        if (strcmp(argv[i], "/anonymous") == 0)
            bAnonymous = true;
        else if (strstr(argv[i], "/delete:") == argv[i])
        {
            if (action != ACTION_NONE)
                return Usage();
        }
    }
}

```

```

        action = ACTION_DELETE_SESSION;
        mbstowcs(wszSessionName, argv[i] + 8, 1024);
    }
    else if (strstr(argv[i], "/chat:") == argv[i])
    {
        if (action != ACTION_NONE)
            return Usage();
        action = ACTION_CHAT;
        mbstowcs(wszSessionName, argv[i] + 6, 1024);
    }
    else if (strcmp(argv[i], "/sessions") == 0)
    {
        if (action != ACTION_NONE)
            return Usage();
        action = ACTION_LIST_SESSION_NAMES;
    }
    else if (strstr(argv[i], "/host:") == argv[i])
    {
        if (pcsi != 0)
            return Usage();
        mbstowcs(wszHostName, argv[i] + 6, 1024);
        pcsi = &csi;
    }
    else if (strstr(argv[i], "/password:") == argv[i])
    {
        mbstowcs(wszPassword, argv[i] + 10, 1024);
        cai.PasswordLength = wcslen(wszPassword);
    }
    else if (strstr(argv[i], "/user:") == argv[i])
    {
        if (pcai != 0 || bAnonymous)
            return Usage();
        char *pszDelim = strchr(argv[i] + 7, '\\');
        if (pszDelim == 0)
            return Usage();
        *pszDelim = 0;
        pszDelim++;
        mbstowcs(wszDomainName, argv[i] + 6, 1024);
        cai.DomainLength = wcslen(wszDomainName);
        mbstowcs(wszUserName, pszDelim, 1024);
        cai.UserLength = wcslen(wszUserName);
        pcai = &cai;
    }
}

if (action == ACTION_NONE)
    return Usage();
HRESULT hr = CoInitializeEx(0, COINIT_MULTITHREADED);
if (FAILED(hr))

```

```

        return hr;

    // allow anonymous callbacks from chat server
    hr = CoInitializeSecurity(0, -1, 0, 0,
                             RPC_C_AUTHN_LEVEL_NONE,
                             RPC_C_IMP_LEVEL_ANONYMOUS,
                             0, EOAC_NONE, 0);

    if (SUCCEEDED(hr))
    {
        // grab the requested session manager
        IChatSessionManager *pcsm = 0;
        hr = CoGetClassObject(CLSID_ChatSession, CLSCTX_ALL,
                             pcsi, IID_IChatSessionManager,
                             (void**)&pcsm);
        if (SUCCEEDED(hr))
        {
            // apply security blanket if desired
            if (!bAnonymous)
                hr = CoSetProxyBlanket(pcsm, RPC_C_AUTHN_WINNT,
                                      RPC_C_AUTHZ_NONE, 0,
                                      RPC_C_AUTHN_LEVEL_PKT,
                                      RPC_C_IMP_LEVEL_IDENTIFY,
                                      pcai, EOAC_NONE);

            // dispatch request
            switch (action)
            {
                case ACTION_CHAT:
                    Chat(wszSessionName, pcsm, pcai, bAnonymous);
                    break;
                case ACTION_DELETE_SESSION:
                    Delete(wszSessionName, pcsm);
                    break;
                case ACTION_LIST_SESSION_NAMES:
                    List(pcsm);
                    break;
                default:
                    Usage();
            }
        }
        // release session manager
        pcsm->Release();
    }
    CoUninitialize();
    return hr;
}

```

## chatsession.h

```
// ChatSession.h

#ifndef _CHATSESSION_H
#define _CHATSESSION_H

// this pragma shuts up the compiler warnings due to
// the pre MSC11SP1 debugger choking on long template names.
#pragma warning(disable:4786)

#define _WIN32_WINNT 0x403
#include <windows.h>
#include <map>
#include <vector>
#include <string>
using namespace std;

// bring in IDL-generated interface definitions
#include "..\include\COMChat.h"

// this class models a particular chat session
class ChatSession : public IChatSession
{
    friend class StatementEnumerator;
    LONG             m_cRef;
    CRITICAL_SECTION   m_csStatementLock;
    CRITICAL_SECTION   m_csAdviseLock;
    OLECHAR          m_wszSessionName[1024];
    bool              m_bIsDeleted;
    bool              m_bAllowAnonymousAccess;
    vector<wstring>   m_statements;
    struct LISTENER
    {
        LISTENER           *pPrev;
        LISTENER           *pNext;
        OLECHAR            *pwszUser;
        IChatSessionEvents *pItf;
    };
    LISTENER         *m_pHeadListeners;
    void SLock(void);
    void SUNlock(void);
    void ALock(void);
    void AUnlock(void);
    bool CheckAccess(const OLECHAR *pwszUser);
protected:
    virtual ~ChatSession(void);
}
```

```

        void Fire_OnNewStatement(const OLECHAR *pwszUser,
                                const OLECHAR *pwszStatement);
        void Fire_OnNewUser(const OLECHAR *pwszUser);
        void Fire_OnUserLeft(const OLECHAR *pwszUser);
public:
    ChatSession(const OLECHAR *pwszSessionName,
                bool bAllowAnonymousAccess);

    void Disconnect(void);
// IUnknown methods
    STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

// IChatSession methods
    STDMETHODIMP get_SessionName(OLECHAR **ppwsz);
    STDMETHODIMP Say(const OLECHAR *pwszStatement);
    STDMETHODIMP GetStatements(IEnumString **ppes);
    STDMETHODIMP Advise(IChatSessionEvents *pEventSink,
                       DWORD *pdwReg);
    STDMETHODIMP Unadvise(DWORD dwReg);
};

// this class enumerates the statements of a session
class StatementEnumerator : public IEnumString
{
    LONG             m_cRef;
    ChatSession      *m_pThis;
    vector<wstring>::iterator   m_cursor;
    CRITICAL_SECTION m_csLock;
protected:
    void Lock(void);
    void Unlock(void);
    virtual ~StatementEnumerator(void);
public:
    StatementEnumerator(ChatSession *pThis);

// IUnknown methods
    STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

// IEnumString methods
    STDMETHODIMP Next(ULONG cElems, OLECHAR **rgElems,
                     ULONG *pcFetched);
    STDMETHODIMP Skip(ULONG cElems);
    STDMETHODIMP Reset(void);
    STDMETHODIMP Clone(IEnumString **ppes);
};

```

```

// this class models the management of chat sessions
// and acts as the class object for CLSID_ChatSession
class ChatSessionClass : public IChatSessionManager,
                           public IExternalConnection
{
    friend class SessionNamesEnumerator;
    typedef map<wstring, ChatSession *> SESSIONMAP;
    LONG             m_cStrongLocks;
    SESSIONMAP       m_sessions;
    CRITICAL_SECTION m_csSessionLock;
    void Lock(void);
    void Unlock(void);
    bool CheckAccess(const OLECHAR *pwszUser);

public:
    virtual ~ChatSessionClass(void);
    ChatSessionClass(void);

    // IUnknown methods
    STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

    // IExternalConnection methods
    STDMETHODIMP_(DWORD) AddConnection(DWORD extconn, DWORD);
    STDMETHODIMP_(DWORD) ReleaseConnection(DWORD extconn, DWORD,
                                           BOOL bLastReleaseKillsStub);

    // IChatSessionManager methods
    STDMETHODIMP GetSessionNames(IEnumString **ppes);
    STDMETHODIMP FindSession(const OLECHAR *pwszSessionName,
                           BOOL bDontCreate,
                           BOOL bAllowAnonymousAccess,
                           IChatSession **ppcs);
    STDMETHODIMP DeleteSession(const OLECHAR *pwszSessionName);
};

// this class enumerates the session names of a server
class SessionNamesEnumerator : public IEnumString
{
    LONG             m_cRef;
    vector<wstring> *m_pStrings;
    SessionNamesEnumerator *m_pCloneSource;
    vector<wstring>::iterator m_cursor;
    CRITICAL_SECTION m_csLock;

protected:
    vector<wstring>& Strings(void);
    void Lock(void);
    void Unlock(void);
    virtual ~SessionNamesEnumerator(void);
};

```

```

public:
    SessionNamesEnumerator(ChatSessionClass *pSessionClass);
    SessionNamesEnumerator(SessionNamesEnumerator *pCloneSource);

    // IUnknown methods
    STDMETHODIMP QueryInterface(REFIID riid, void **ppv);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);

    // IEnumString methods
    STDMETHODIMP Next(ULONG cElems, OLECHAR **rgElems,
                      ULONG *pcFetched);
    STDMETHODIMP Skip(ULONG cElems);
    STDMETHODIMP Reset(void);
    STDMETHODIMP Clone(IEnumString **ppes);
};

#endif

```

## chatsession.cpp

```

// ChatSession.cpp

#include "ChatSession.h"
#include <iaccess.h>

// these routines are defined in svc.cpp to
// control server lifetime
extern void ModuleLock(void);
extern void ModuleUnlock(void);

// these access control objects are created
// in svc.cpp to control various privileged
// operations. Most operations in this class
// are non-privileged, so anyone can get in.
extern IAccessControl *g_pacUsers;
extern IAccessControl *g_pacAdmins;

// utility functions //////////////////////

// duplicate an OLECHAR * using CoTaskMemAlloc
OLECHAR *OLESTRDUP(const OLECHAR *pwsz)
{
    DWORD cb = sizeof(OLECHAR) * (wcslen(pwsz) + 1);
    OLECHAR *pwszResult = (OLECHAR*)CoTaskMemAlloc(cb);
    if (pwszResult)
        wcscpy(pwszResult, pwsz);
    return pwszResult;
}

```

```

}

// get the caller's username (or "anonymous" if
// no authentication was specified by the caller).
OLECHAR *GetCaller(void)
{
    OLECHAR *pwsz = 0;
    HRESULT hr = CoQueryClientBlanket(0,0,0,0,0,(void**)&pwsz,0);
    if (SUCCEEDED(hr))
        return OLESTRDUP(pwsz);
    else
        return OLESTRDUP(OLESTR("anonymous"));
}

// class ChatSession /////////////////
ChatSession::ChatSession(const OLECHAR *pwszSessionName,
                        bool bAllowAnonymousAccess)
: m_cRef(0),
  m_bAllowAnonymousAccess(bAllowAnonymousAccess),
  m_pHeadListeners(0)
{
    wcscpy(m_wszSessionName, pwszSessionName);
    InitializeCriticalSection(&m_csStatementLock);
    InitializeCriticalSection(&m_csAdviseLock);
}

ChatSession::~ChatSession(void)
{
    DeleteCriticalSection(&m_csStatementLock);
    DeleteCriticalSection(&m_csAdviseLock);
// tear down connected listeners
    while (m_pHeadListeners)
    {
        LISTENER *pThisNode = m_pHeadListeners;
        if (pThisNode->pItf)
            pThisNode->pItf->Release();
        if (pThisNode->pwszUser)
            CoTaskMemFree(pThisNode->pwszUser);
        m_pHeadListeners = pThisNode->pNext;
        delete pThisNode;
    }
}

// helper methods ///////////////
void ChatSession::Disconnect(void)
{
    CoDisconnectObject(this, 0);
}

```

```

// tear down connected listeners
ALock();
while (m_pHeadListeners) .
{
    LISTENER *pThisNode = m_pHeadListeners;
    if (pThisNode->pItf)
        pThisNode->pItf->Release();
    if (pThisNode->pwszUser)
        CoTaskMemFree(pThisNode->pwszUser);
    m_pHeadListeners = pThisNode->pNext;
    delete pThisNode;
}
AUnlock();
}

// send the OnNewStatement event to all listeners
void
ChatSession::Fire_OnNewStatement(const OLECHAR *pwszUser,
                                 const OLECHAR *pwszStatement)
{
    ALock();
    for (LISTENER *pNode = m_pHeadListeners;
         pNode != 0; pNode = pNode->pNext)
    {
        if (pNode->pItf)
            pNode->pItf->OnNewStatement(pwszUser, pwszStatement);
    }
    AUnlock();
}

// send the OnNewUser event to all listeners
void
ChatSession::Fire_OnNewUser(const OLECHAR *pwszUser)
{
    ALock();
    for (LISTENER *pNode = m_pHeadListeners;
         pNode != 0; pNode = pNode->pNext)
    {
        if (pNode->pItf)
            pNode->pItf->OnNewUser(pwszUser);
    }
    AUnlock();
}

// send the OnUserLeft event to all listeners
void
ChatSession::Fire_OnUserLeft(const OLECHAR *pwszUser)
{
    ALock();

```

```

for (LISTENER *pNode = m_pHeadListeners;
     pNode != 0; pNode = pNode->pNext)
{
    if (pNode->pItf)
        pNode->pItf->OnUserLeft(pwszUser);
}
AUnlock();
}

// lock wrappers
void ChatSession::SLock(void)
{
    EnterCriticalSection(&m_csStatementLock);
}

void ChatSession::SUnlock(void)
{
    LeaveCriticalSection(&m_csStatementLock);
}

void ChatSession::ALock(void)
{
    EnterCriticalSection(&m_csAdviseLock);
}

void ChatSession::AUnlock(void)
{
    LeaveCriticalSection(&m_csAdviseLock);
}

// helper method to check access to Say method
bool
ChatSession::CheckAccess(const OLECHAR *pwszUser)
{
    if (wcscmp(pwszUser, L"anonymous") == 0)
        return m_bAllowAnonymousAccess;
    // form trustee from caller and use Access Control
    // object hardwired to COMChat Users group
    TRUSTEEW trustee = {
        0, NO_MULTIPLE_TRUSTEE, TRUSTEE_IS_NAME,
        TRUSTEE_IS_USER,
        const_cast<OLECHAR*>(pwszUser)
    };
    BOOL bIsAllowed;
    HRESULT hr = g_pacUsers->IsAccessAllowed(&trustee, 0,
                                              COM_RIGHTS_EXECUTE,
                                              &bIsAllowed);
    return SUCCEEDED(hr) && bIsAllowed != FALSE;
}

```

```

// IUnknown methods
STDMETHODIMP
ChatSession::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IChatSession*>(this);
    else if (riid == IID_IChatSession)
        *ppv = static_cast<IChatSession*>(this);
    else
        return (*ppv = 0), E_NOINTERFACE;
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}

STDMETHODIMP_(ULONG)
ChatSession::AddRef(void)
{
    ModuleLock();
    return InterlockedIncrement(&m_cRef);
}

STDMETHODIMP_(ULONG)
ChatSession::Release(void)
{
    LONG res = InterlockedDecrement(&m_cRef);
    if (res == 0)
        delete this;
    ModuleUnlock();
    return res;
}

// IChatSession methods
STDMETHODIMP
ChatSession::get_SessionName(OLECHAR **ppwsz)
{
    if (!ppwsz)
        return E_INVALIDARG;
    else if ((*ppwsz = OLESTRDUP(m_wszSessionName)) == 0)
        return E_OUTOFMEMORY;
    return S_OK;
}

STDMETHODIMP
ChatSession::Say(const OLECHAR *pwszStatement)
{
    HRESULT hr = S_OK;
    // protect access to method
}

```

```

OLECHAR *pwszUser = GetCaller();
if (pwszUser && CheckAccess(pwszUser))
{
    SLock();
    try
    {
        wstring s = pwszUser;
        s += L":";
        s += pwszStatement;
        m_statements.push_back(s);
    }
    catch(...)
    {
        hr = E_OUTOFMEMORY;
    }
    SUnlock();
    if (SUCCEEDED(hr))
        Fire_OnNewStatement(pwszUser, pwszStatement);
}
else
    hr = E_ACCESSDENIED;
CoTaskMemFree(pwszUser);
return hr;
}

STDMETHODIMP
ChatSession::GetStatements(IEnumString **ppes)
{
    if (ppes == 0)
        return E_INVALIDARG;
    *ppes = new StatementEnumerator(this);
    if (*ppes == 0)
        return E_OUTOFMEMORY;
    (*ppes)->AddRef();
    return S_OK;
}

STDMETHODIMP
ChatSession::Advise(IChatSessionEvents *pEventSink,
                    DWORD *pdwReg)
{
    HRESULT hr = S_OK;
    if (pEventSink == 0 || pdwReg == 0)
        return E_INVALIDARG;
    LISTENER *pNew = new LISTENER;
    if (pNew == 0)
        return E_OUTOFMEMORY;
    OLECHAR *pwszUser = GetCaller();
    if (pwszUser)

```

```

{
    Fire_OnNewUser(pwszUser);
    ALock();
    pNew->pwszUser = pwszUser;
    if (pNew->pItf = pEventSink)
        pEventSink->AddRef();
    pNew->pNext = m_pHeadListeners;
    if (m_pHeadListeners)
        m_pHeadListeners->pPrev = pNew;
    pNew->pPrev = 0;
    m_pHeadListeners = pNew;
    AUnlock();
}
else
{
    delete pNew;
    return E_OUTOFMEMORY;
}
*pdwReg = reinterpret_cast<DWORD>(pNew);
return hr;
}

STDMETHODIMP
ChatSession::Unadvise(DWORD dwReg)
{
    if (dwReg == 0)
        return E_INVALIDARG;
    HRESULT hr = S_OK;
    LISTENER *pThisNode = reinterpret_cast<LISTENER *>(dwReg);
    ALock();
    if (pThisNode->pPrev)
        pThisNode->pPrev->pNext = pThisNode->pNext;
    else
        m_pHeadListeners = pThisNode->pNext;
    if (pThisNode->pNext)
        pThisNode->pNext->pPrev = pThisNode->pPrev;
    if (pThisNode->pItf)
        pThisNode->pItf->Release();
    OLECHAR *pwszUser = pThisNode->pwszUser;
    delete pThisNode;
    AUnlock();
    Fire_OnUserLeft(pwszUser);
    CoTaskMemFree(pwszUser);
    return hr;
}

// class StatementEnumerator /////////////
StatementEnumerator::StatementEnumerator(ChatSession *pThis)

```

```

: m_cRef(0),
m_pThis(pThis),
m_cursor(pThis->m_statements.begin())
{
    m_pThis->AddRef();
    InitializeCriticalSection(&m_csLock);
}

StatementEnumerator::~StatementEnumerator(void)
{
    m_pThis->Release();
    DeleteCriticalSection(&m_csLock);
}

// lock helpers (note that ChatSession is locked
// simultaneously)
void
StatementEnumerator::Lock(void)
{
    EnterCriticalSection(&m_csLock);
    m_pThis->SLock();
}

void
StatementEnumerator::Unlock(void)
{
    LeaveCriticalSection(&m_csLock);
    m_pThis->SUnlock();
}

// IUnknown methods
STDMETHODIMP
StatementEnumerator::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IEnumString*>(this);
    else if (riid == IID_IEnumString)
        *ppv = static_cast<IEnumString*>(this);
    else
        return (*ppv = 0), E_NOINTERFACE;
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}

STDMETHODIMP_(ULONG)
StatementEnumerator::AddRef(void)
{
    return InterlockedIncrement(&m_cRef);
}

```

```

}

STDMETHODIMP_(ULONG)
StatementEnumerator::Release(void)
{
    LONG res = InterlockedDecrement(&m_cRef);
    if (res == 0)
        delete this;
    return res;
}

// IEnumString methods
STDMETHODIMP
StatementEnumerator::Next(ULONG cElems, OLECHAR **rgElems,
                        ULONG *pcFetched)
{
    if (pcFetched == 0 && cElems > 1)
        return E_INVALIDARG;
    ZeroMemory(rgElems, sizeof(OLECHAR*) * cElems);
    Lock();
    ULONG cActual = 0;
    while (cActual < cElems
        && m_cursor != m_pThis->m_statements.end())
    {
        if (rgElems[cActual] = OLESTRDUP((*m_cursor).c_str()))
        {
            m_cursor++;
            cActual++;
        }
        else // allocation error, unwind
        {
            while (cActual > 0)
            {
                cActual--;
                CoTaskMemFree(rgElems[cActual]);
                rgElems[cActual] = 0;
            }
            break;
        }
    }
    Unlock();
    if (pcFetched)
        *pcFetched = cActual;
    return cElems == cActual ? S_OK : S_FALSE;
}

STDMETHODIMP
StatementEnumerator::Skip(ULONG cElems)
{

```

```

Lock();
ULONG cActual = 0;
while (cActual < cElems
    && m_cursor != m_pThis->m_statements.end())
{
    m_cursor++;
    cActual++;
}
Unlock();
return cElems == cActual ? S_OK : S_FALSE;
}

STDMETHODIMP
StatementEnumerator::Reset(void)
{
    Lock();
    m_cursor = m_pThis->m_statements.begin();
    Unlock();
    return S_OK;
}

STDMETHODIMP
StatementEnumerator::Clone(IEnumString **ppes)
{
    if (ppes == 0)
        return E_INVALIDARG;
    if (*ppes = new StatementEnumerator(m_pThis))
        return S_OK;
    return E_OUTOFMEMORY;
}

// class ChatSessionClass //////////////////////

ChatSessionClass::ChatSessionClass(void)
: m_cStrongLocks(0)
{
    InitializeCriticalSection(&m_csSessionLock);
}

ChatSessionClass::~ChatSessionClass(void)
{
    DeleteCriticalSection(&m_csSessionLock);
}

void
ChatSessionClass::Lock(void)
{
    EnterCriticalSection(&m_csSessionLock);
}

```

```

void
ChatSessionClass::Unlock(void)
{
    LeaveCriticalSection(&m_csSessionLock);
}
// helper method to protect access to DeleteSession
// to only allow COMChat Admins to delete groups
bool
ChatSessionClass::CheckAccess(const OLECHAR *pwszUser)
{
    if (wcscmp(pwszUser, L"anonymous") == 0)
        return false;

    TRUSTEEW trustee = {
        0, NO_MULTIPLE_TRUSTEE, TRUSTEE_IS_NAME,
        TRUSTEE_IS_USER, const_cast<OLECHAR*>(pwszUser)
    };
    BOOL bIsAllowed;
    HRESULT hr = g_pacAdmins->IsAccessAllowed(&trustee, 0,
                                                COM_RIGHTS_EXECUTE,
                                                &bIsAllowed);
    if (FAILED(hr))
        bIsAllowed = false;
    return SUCCEEDED(hr) && bIsAllowed != FALSE;
}

// IUnknown methods
STDMETHODIMP
ChatSessionClass::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IChatSessionManager*>(this);
    else if (riid == IID_IChatSessionManager)
        *ppv = static_cast<IChatSessionManager*>(this);
    else if (riid == IID_IExternalConnection)
        *ppv = static_cast<IExternalConnection*>(this);
    else
        return (*ppv = 0), E_NOINTERFACE;
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}

STDMETHODIMP_(ULONG)
ChatSessionClass::AddRef(void)
{
    return 2;
}

```

```

STDMETHODIMP_(ULONG)
ChatSessionClass::Release(void)
{
    return 1;
}

// IExternalConnection methods
STDMETHODIMP_(DWORD)
ChatSessionClass::AddConnection(DWORD extconn, DWORD)
{
    if (extconn & EXTCONN_STRONG)
    {
        ModuleLock();
        return InterlockedIncrement(&m_cStrongLocks);
    }
    return 0;
}

STDMETHODIMP_(DWORD)
ChatSessionClass::ReleaseConnection(DWORD extconn, DWORD,
                                    BOOL bLastReleaseKillsStub)
{
    if (extconn & EXTCONN_STRONG)
    {
        LONG res = InterlockedDecrement(&m_cStrongLocks);
        if (res == 0 && bLastReleaseKillsStub)
            CoDisconnectObject(
                static_cast<IExternalConnection*>(this), 0);
        ModuleUnlock();
        return res;
    }
    return 0;
}

// IChatSessionManager methods
STDMETHODIMP
ChatSessionClass::GetSessionNames(IEnumString **ppes)
{
    if (ppes == 0)
        return E_INVALIDARG;
    if (*ppes = new SessionNamesEnumerator(this))
    {
        (*ppes)->AddRef();
        return S_OK;
    }
    else
        return E_OUTOFMEMORY;
}

```

```

STDMETHODIMP
ChatSessionClass::FindSession(const OLECHAR *pwszSessionName,
                               BOOL bDontCreate,
                               BOOL bAllowAnonymousAccess,
                               IChatSession **ppcs)
{
    if (ppcs == 0)
        return E_INVALIDARG;
    HRESULT hr = E_FAIL;
    *ppcs = 0;
    OLECHAR *pwszUser = GetCaller();
    Lock();
    SESSIONMAP::iterator it = m_sessions.find(pwszSessionName);
    if (it == m_sessions.end())
    {
        if (bDontCreate)
            hr = E_FAIL;
        else if (!bAllowAnonymousAccess
                  && wcscmp(pwszUser, L"anonymous") == 0)
            hr = E_ACCESSDENIED;
        else
        {
            ChatSession *pNew =
                new ChatSession(pwszSessionName,
                                bAllowAnonymousAccess != FALSE);
            if (pNew)
            {
                pNew->AddRef();
                m_sessions.insert(
                    pair<wstring,
                        ChatSession*>(pwszSessionName,
                                      pNew));
                (*ppcs = pNew)->AddRef();
                hr = S_OK;
            }
            else
                hr = E_OUTOFMEMORY;
        }
    }
    else
    {
        (*ppcs = (*it).second)->AddRef();
        hr = S_OK;
    }
    Unlock();
    CoTaskMemFree(pwszUser);
    return hr;
}

```

```

STDMETHODIMP
ChatSessionClass::DeleteSession(const OLECHAR *pwszSessionName)
{
    if (pwszSessionName == 0)
        return E_INVALIDARG;
    HRESULT hr = E_FAIL;
    OLECHAR *pwszUser = GetCaller();
    if (CheckAccess(pwszUser))
    {
        Lock();
        SESSIONMAP::iterator it
            = m_sessions.find(pwszSessionName);
        if (it == m_sessions.end())
        {
            hr = E_FAIL;
        }
        else
        {
            (*it).second->Disconnect();
            (*it).second->Release();
            m_sessions.erase(it);
            hr = S_OK;
        }
        Unlock();
    }
    else
        hr = E_ACCESSDENIED;
    CoTaskMemFree(pwszUser);
    return hr;
}

// class SessionNamesEnumerator

vector<wstring>&
SessionNamesEnumerator::Strings(void)
{
    if (m_pStrings)
        return *m_pStrings;
    else
        return *(m_pCloneSource->m_pStrings);
}

void
SessionNamesEnumerator::Lock(void)
{
    EnterCriticalSection(&m_csLock);
}

```

```

void
SessionNamesEnumerator::Unlock(void)
{
    LeaveCriticalSection(&m_csLock);
}

SessionNamesEnumerator::SessionNamesEnumerator(
    ChatSessionClass *pSessionClass)
: m_cRef(0),
  m_pStrings(0),
  m_pCloneSource(0)
{
    typedef ChatSessionClass::SESSIONMAP::iterator iterator;
    ChatSessionClass::SESSIONMAP &sessions
        = pSessionClass->m_sessions;
    m_pStrings = new vector<wstring>;
    pSessionClass->Lock();
    for (iterator it = sessions.begin();
         it != sessions.end();
         it++)
    {
        m_pStrings->push_back((*it).first);
    }
    pSessionClass->Unlock();
    m_cursor = Strings().begin();
    InitializeCriticalSection(&m_csLock);
}

SessionNamesEnumerator::SessionNamesEnumerator(
    SessionNamesEnumerator *pCloneSource)
: m_cRef(0),
  m_pStrings(0),
  m_pCloneSource(pCloneSource)
{
    m_pCloneSource->AddRef();
    m_cursor = Strings().begin();
    InitializeCriticalSection(&m_csLock);
}

SessionNamesEnumerator::~SessionNamesEnumerator(void)
{
    if (m_pCloneSource)
        m_pCloneSource->Release();
    else if (m_pStrings)
        delete m_pStrings;
    DeleteCriticalSection(&m_csLock);
}

// IUnknown methods

```

```

STDMETHODIMP
SessionNamesEnumerator::QueryInterface(REFIID riid, void **ppv)
{
    if (riid == IID_IUnknown)
        *ppv = static_cast<IEnumString*>(this);
    else if (riid == IID_IEnumString)
        *ppv = static_cast<IEnumString*>(this);
    else
        return (*ppv = 0), E_NOINTERFACE;
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}

STDMETHODIMP_(ULONG)
SessionNamesEnumerator::AddRef(void)
{
    ModuleLock();
    return InterlockedIncrement(&m_cRef);
}

STDMETHODIMP_(ULONG)
SessionNamesEnumerator::Release(void)
{
    LONG res = InterlockedDecrement(&m_cRef);
    if (res == 0)
        delete this;
    ModuleUnlock();
    return res;
}

// IEnumString methods
STDMETHODIMP
SessionNamesEnumerator::Next(ULONG cElems, OLECHAR **rgElems,
                            ULONG *pcFetched)
{
    if (cElems > 1 && pcFetched == 0)
        return E_INVALIDARG;
    ULONG cActual = 0;
    vector<wstring> &rstrings = Strings();
    Lock();
    while (cActual < cElems
           && m_cursor != rstrings.end())
    {
        if (rgElems[cActual] = OLESTRDUP((*m_cursor).c_str()))
        {
            m_cursor++;
            cActual++;
        }
    }
}

```

```

        else // allocation error, unwind
    {
        while (cActual > 0)
        {
            cActual--;
            CoTaskMemFree(rgElems[cActual]);
            rgElems[cActual] = 0;
        }
        break;
    }
}
Unlock();
if (cActual)
    *pcFetched = cActual;
return cActual == cElems ? S_OK : S_FALSE;
}

STDMETHODIMP
SessionNamesEnumerator::Skip(ULONG cElems)
{
    ULONG cActual = 0;
    vector<wstring> &rstrings = Strings();
    Lock();
    while (cActual < cElems
        && m_cursor != rstrings.end())
    {
        m_cursor++;
        cActual++;
    }
    Unlock();
    return cActual == cElems ? S_OK : S_FALSE;
}

STDMETHODIMP
SessionNamesEnumerator::Reset(void)
{
    Lock();
    m_cursor = Strings().begin();
    Unlock();
    return S_OK;
}

STDMETHODIMP
SessionNamesEnumerator::Clone(IEnumString **ppes)
{
    if (ppes == 0)
        return E_INVALIDARG;
    SessionNamesEnumerator *pCloneSource = m_pCloneSource;
    if (pCloneSource == 0) // we are the source

```

```

    m_pCloneSource = this;
    *ppes = new SessionNamesEnumerator(pCloneSource);
    if (*ppes)
    {
        (*ppes)->AddRef();
        return S_OK;
    }
    return E_OUTOFMEMORY;
}

```

## SVC.CPP

```

// svc.cpp

#define _WIN32_WINNT 0x403
#include <windows.h>
#include <olectrl.h>
#include <initguid.h>
#include <iaccess.h>

#include "ChatSession.h"
#include "../include/COMChat_i.c"

#if !defined(HAVE_IID_IAccessControl)
// there is a common bug in the SDK headers and libs
// that causes IID_IAccessControl to be undefined.
// We define it here to give the GUID linkage.
DEFINE_GUID(IID_IAccessControl,0xEEDD23E0, 0x8410, 0x11CE,
            0xA1, 0xC3, 0x08, 0x00, 0x2B, 0x2B, 0x8D, 0x8F);
#endif

// standard MTA lifetime management helpers
HANDLE g_hEventDone = CreateEvent(0, TRUE, FALSE, 0);

void ModuleLock(void)
{
    CoAddRefServerProcess();
}

void ModuleUnlock(void)
{
    if (CoReleaseServerProcess() == 0)
        SetEvent(g_hEventDone);
}

// standard self-registration table
const char *g_RegTable[][3] = {
{ "CLSID\\{5223A053-2441-11d1-AF4F-0060976AA886}",
  0, "ChatSession" },

```

```

{ "CLSID\\{5223A053-2441-11d1-AF4F-0060976AA886}" ,
  "AppId",  "{5223A054-2441-11d1-AF4F-0060976AA886}"
},
{ "CLSID\\{5223A053-2441-11d1-AF4F-0060976AA886}\\LocalServer32",
  0, (const char*)-1 // rogue value indicating file name
},
{ "AppID\\{5223A054-2441-11d1-AF4F-0060976AA886}" ,
  0, "ChatSession Server" },
{ "AppID\\{5223A054-2441-11d1-AF4F-0060976AA886}" ,
  "RunAs", "Domain\\ReplaceMe"
},
{ "AppID\\{5223A054-2441-11d1-AF4F-0060976AA886}" ,
  "Chat Admins Group", "Domain\\ReplaceMe"
},
{ "AppID\\{5223A054-2441-11d1-AF4F-0060976AA886}" ,
  "Chat Users Group", "Domain\\ReplaceMe"
},
{ "AppID\\COMChat.exe",
  "AppId",  "{5223A054-2441-11d1-AF4F-0060976AA886}"
},
};

// self-unregistration routine
STDAPI UnregisterServer(void) {
    HRESULT hr = S_OK;
    int nEntries = sizeof(g_RegTable)/sizeof(*g_RegTable);
    for (int i = nEntries - 1; i >= 0; i--) {
        const char *pszKeyName = g_RegTable[i][0];

        long err = RegDeleteKeyA(HKEY_CLASSES_ROOT, pszKeyName);
        if (err != ERROR_SUCCESS)
            hr = S_FALSE;
    }
    return hr;
}

// self-registration routine
STDAPI RegisterServer(HINSTANCE hInstance = 0) {
    HRESULT hr = S_OK;
    // look up server's file name
    char szFileName[MAX_PATH];
    GetModuleFileNameA(hInstance, szFileName, MAX_PATH);
    // register entries from table
    int nEntries = sizeof(g_RegTable)/sizeof(*g_RegTable);
    for (int i = 0; SUCCEEDED(hr) && i < nEntries; i++) {
        const char *pszKeyName    = g_RegTable[i][0];
        const char *pszValueName = g_RegTable[i][1];
        const char *pszValue     = g_RegTable[i][2];
        // map rogue value to module file name

```

```

if (pszValue == (const char*)-1)
    pszValue = szFileName;
HKEY hkey;
// create the key
long err = RegCreateKeyA(HKEY_CLASSES_ROOT,
                        pszKeyName, &hkey);
if (err == ERROR_SUCCESS) {
// set the value
    err = RegSetValueExA(hkey, pszValueName, 0,
                        REG_SZ, (const BYTE*)pszValue,
                        (strlen(pszValue) + 1));
    RegCloseKey(hkey);
}
if (err != ERROR_SUCCESS) {
// if cannot add key or value, back out and fail
    UnregisterServer();
    hr = SELFREG_E_CLASS;
}
}
return hr;
}

// these point to standard access control objects
// used to protect particular methods
IAccessControl *g_pacUsers = 0;
IAccessControl *g_pacAdmins = 0;

// this routine is called at process init time
// to build access control objects and to allow
// anonymous access to server by default
HRESULT InitializeApplicationSecurity(void)
{
// load groupnames from registry
static OLECHAR wszAdminsGroup[1024];
static OLECHAR wszUsersGroup[1024];
HKEY hkey;
long err = RegOpenKeyEx(HKEY_CLASSES_ROOT,
                        _TEXT("AppID\\{5223A054-2441-11d1-AF4F-0060976AA886}"),
                        0, KEY_QUERY_VALUE,
                        &hkey);
if (err == ERROR_SUCCESS)
{
    DWORD cb = sizeof(wszAdminsGroup);
    err = RegQueryValueExW(hkey, L"Chat Admins Group",
                          0, 0, (BYTE*)wszAdminsGroup,
                          &cb);
    cb = sizeof(wszAdminsGroup);
    if (err == ERROR_SUCCESS)
        err = RegQueryValueExW(hkey,

```

```

        L"Chat Users Group",
        0, 0, (BYTE*)wszUsersGroup,
        &cb);

    RegCloseKey(hkey);
}

if (err != ERROR_SUCCESS)
    return MAKE_HRESULT(SEVERITY_ERROR, FACILITY_WIN32,
        GetLastError());

// declare vectors of user/groups for 2 access
// control objects

ACtrl_ACCESS_ENTRYW rgaaeUsers[] = {
{ {0, NO_MULTIPLE_TRUSTEE, TRUSTEE_IS_NAME,
    TRUSTEE_IS_GROUP, wszUsersGroup },
    ACTRL_ACCESS_ALLOWED, COM_RIGHTS_EXECUTE, 0,
    NO_INHERITANCE, 0 },
};

ACtrl_ACCESS_ENTRY_LISTW aaelUsers =
    sizeof(rgaaeUsers)/sizeof(*rgaaeUsers),
    rgaaeUsers
};

ACtrl_PROPERTY_ENTRYW apeUsers = { 0, &aaelUsers, 0 };
ACtrl_ACCESSW aaUsers = { 1, &apeUsers };

ACtrl_ACCESS_ENTRYW rgaaeAdmins[] = {
{ {0, NO_MULTIPLE_TRUSTEE, TRUSTEE_IS_NAME,
    TRUSTEE_IS_GROUP, wszAdminsGroup },
    ACTRL_ACCESS_ALLOWED, COM_RIGHTS_EXECUTE, 0,
    NO_INHERITANCE, 0 },
};

ACtrl_ACCESS_ENTRY_LISTW aaelAdmins =
    sizeof(rgaaeAdmins)/sizeof(*rgaaeAdmins),
    rgaaeAdmins
};

ACtrl_PROPERTY_ENTRYW apeAdmins = { 0, &aaelAdmins, 0 };
ACtrl_ACCESSW aaAdmins = { 1, &apeAdmins };

HRESULT hr = CoInitializeSecurity(0, -1, 0, 0,
    RPC_C_AUTHN_LEVEL_NONE,
    RPC_C_IMP_LEVEL_ANONYMOUS,
    0,
    EOAC_NONE,
    0);

if (SUCCEEDED(hr))
{
    hr = CoCreateInstance(CLSID_DCOMAccessControl,
        0, CLSCTX_ALL, IID_IAccessControl,
        (void**)&g_pacUsers);
}

```

```

if (SUCCEEDED(hr))
    hr = g_pacUsers->SetAccessRights(&aaUsers);
if (SUCCEEDED(hr))
{
    hr = CoCreateInstance(CLSID_DCOMAccessControl,
                          0, CLSCTX_ALL,
                          IID_IAccessControl,
                          (void**)&g_pacAdmins);
    if (SUCCEEDED(hr))
        hr = g_pacAdmins->SetAccessRights(&aaAdmins);
}
if (FAILED(hr))
{
    if (g_pacAdmins)
    {
        g_pacAdmins->Release();
        g_pacAdmins = 0;
    }
    if (g_pacUsers)
    {
        g_pacUsers->Release();
        g_pacUsers = 0;
    }
}
return hr;
}

// the main thread routine that simply registers the class
// object and waits to die
int WINAPI WinMain(HINSTANCE, HINSTANCE,
                    LPSTR szCmdParam, int)
{
    const TCHAR *pszPrompt =
        _TEXT("Ensure that you have properly ")
        _TEXT("configured the application to ")
        _TEXT("run as a particular user and that ")
        _TEXT("you have manually changed the ")
        _TEXT("Users and Admins Group registry ")
        _TEXT("settings under this server's AppID.");
}

HRESULT hr = CoInitializeEx(0, COINIT_MULTITHREADED);
if (FAILED(hr))
    return hr;

// look for self-registration flags
if (strstr(szCmdParam, "/UnregServer") != 0
    || strstr(szCmdParam, "-UnregServer") != 0)
{

```

```
    hr = UnregisterServer();
    CoUninitialize();
    return hr;
}
else if (strstr(szCmdParam, "/RegServer") != 0
         || strstr(szCmdParam, "-RegServer") != 0)
{
    hr = RegisterServer();
    MessageBox(0, pszPrompt, __TEXT("COMChat"),
              MB_SETFOREGROUND);
    CoUninitialize();
    return hr;
}

// set up process security
hr = InitializeApplicationSecurity();
if (SUCCEEDED(hr))
{
// register class object and wait to die
    DWORD dwReg;
    static ChatSessionClass cmc;
    hr = CoRegisterClassObject(CLSID_ChatSession,
                               static_cast<IExternalConnection*>(&cmc),
                               CLSCTX_LOCAL_SERVER
                               REGCLS_SUSPENDED|REGCLS_MULTIPLEUSE,
                               &dwReg);
    if (SUCCEEDED(hr))
    {
        hr = CoResumeClassObjects();
        if (SUCCEEDED(hr))
            WaitForSingleObject(g_hEventDone, INFINITE);
        CoRevokeClassObject(dwReg);
    }
    g_pacUsers->Release();
    g_pacAdmins->Release();
}
if (FAILED(hr))
    MessageBox(0, pszPrompt, __TEXT("Error"),
              MB_SETFOREGROUND);

CoUninitialize();
return 0;
}
```

# 中英词汇对照表

access control	访问控制
accessor	访问器
activation	激活
aggregation	聚合
apartment	套间
application	应用（一个服务器进程）
assertion	断言
ATL (Active Template Library)	活动模板库
attribute	属性
authentication	认证
authorization service	授权服务
base class	基类
broker	中介代理
build	编译链接
CATID (Category Identifier)	类别标识符
channel	通道
class	类
class emulation	类模仿
class object	类对象（即 class factory 类）
class store	类存储
class table	类表
CLSID (Class Identifier)	类标识符
COM (Component Object Model)	组件对象模型
COM-aware	可感知 COM 的
component category	组件类别
composition	复合
concurrency	并发
constructor	构造函数
containment	包容
CORBA (Common Object Request Broker Architecture)	公用对象请求代理体系结构
critical section	临界区

cursor	游标
DACL (Discretionary Access Control List)	任意访问控制表
data-driven	数据驱动的
DCOM (Distributed Component Object Model)	分布式组件对象模型
delegate	委托
derived class	派生类
destructor	析构函数
directive	(编译器) 指示符
discriminator	鉴别器
display name	显示名
DWP (draft working paper)	(C++) 草案工作文档
encapsulation	封装(性)
enumerator	枚举器
exception	异常
execution context	执行环境(即套间)
export	引出
export list	引出表
file moniker	文件名字对象
flag	标志
GIT (Global Interface Table)	全局接口表
GUID (Global Unique Identifier)	全局唯一标识符
handler	处理器
handler key	控制器键
heap	堆
identity	实体身份
IDL (Interface Definition Language)	接口定义语言
IID (Interface Identifier)	接口 ID
impersonation	模仿
implementation inheritance	实现继承
implicit upcast	隐式向上转换
import	引入
import library	引入库
inheritance	继承
inline	内联
in-process	进程内
interface inheritance	接口继承
interface marshaler	接口列集器
interface proxy	接口代理

interface stub	接口存根
interpretive marshaling	解释性列集
IPID (Interface Pointer Identifier)	接口指针标识符
item moniker	单项名字对象
late bind	迟绑定
LIB ID (Library Identifier)	库标识符
locator object	定位器对象
LSA (Local Security Authority)	局部安全控制中心
marshal	列集
master implementation	主实现
message filter	消息过滤器
method coloring	方法染色 (技术)
method remoting	方法远程传递
MFC (Microsoft Foundation Class)	微软基础类
moniker	名字对象 (即定位器对象)
MTA (Multithreaded Apartment)	多线程套间
MTS (Microsoft Transaction Server)	微软事务服务器
mutex	互斥体
name mangling	名字改编
NDR (Network Data Representation)	网络数据表示
normal marshal	常规列集
NTLM (NT LAN Manager)	Windows NT 的认证协议
OID (Object Identifier)	对象标识符
OLE (Object Linking and Embedding)	对象链接和嵌入
OOP (Oriented-Object Programming)	面向对象程序设计
OR (OXID Resolver)	OXID 解析器
ORPC (Object Remote Procedure Call)	对象远程过程调用
OSF (Open Software Foundation)	开放软件基金会
out-of-process	进程外
OXID (Object Exporter Identifier)	对象引出标识符
permission	许可
persistent image	永久映像
polymorphism	多态 (性)
pragma	编译指示
property	属性
proxy	代理
proxy manager	代理管理器
race condition	竞争条件

raw interface pointer	纯接口指针
reentrancy	重入性
registry	注册表
ROT (Running Object Table)	运行对象表
RTA (Rentalthreaded Apartment)	租用方式的线程套间
RTTI (Runtime Type Identification)	运行时类型识别
runtime	运行时
SCM (Sevice Control Manager)	服务控制管理器
security credential	安全凭证
self-registration	自注册
semaphore	信号量
shim class	垫片类
site interface	站点接口
smart pointer	智能指针
SSPI (Security Support Provider Interface)	安全性支持提供器接口
STA (Singlethreaded Apartment)	单线程套间
stack frame	堆栈结构
stringized	字符串化的
struct	结构
stub manager	存根管理器
subheader	子首部
surrogate	代理
table driven	表格驱动
table marshal	表格列集
tag	标记
tag namespace	标记名字空间
TLS (Thread Local Storage)	线程局部存储
token management	令牌管理
tokenized	符号化的
UDT (user-defined type)	用户自定义类型
unmarshal	散集
variant	变体
vptr (virtual pointer)	虚函数指针
vtbl (virtual function table)	虚函数表
well-implemented	实现良好的
worker thread	辅助线程
working set	工作集