COM技术内幕

——微软组件对象模型

[美] Dale Rogerson 著
杨秀章 译
江 英 审

清华大学出版社

(京)新登字 158 号

COM 技术内幕

Inside COM

Dale Rogerson

Copyright © 1997 by Dale Rogerson.

Original English language Edition Copyright © 1997 by Dale Rogerson.
Published by arrangement with the original publisher, Microsoft Press,
a division of Microsoft Corporation, Redmond, Washington, U.S.A.

本书中文版由 Microsoft Press 授权清华大学出版社出版。

北京市版权局著作权合同登记号 图字 01-98-0270 号

版权所有,翻印必究。
本书封面贴有 Microsoft Press 激光防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

COM 技术内幕:微软组件对象模型/(美)罗杰森(Rogerson, D.)著;杨秀章译 .—北京:清华大学出版社,1998.12

ISBN 7-302-03320-X

Ⅰ. C… Ⅱ. ① 罗… ② 杨… Ⅲ. 软件开发 Ⅳ. TP311.52

中国版本图书馆 CIP 数据核字(1999)第 00830 号

出版者:清华大学出版社(北京清华大学校内,邮编 100084)
http://www.tup.tsinghua.edu.cn
印刷者:清华大学印刷厂
发行者:新华书店总店北京发行所
开 本:787×960 1/16 印张:19.5 字数:425 千字
版 次:1999 年 3 月 第 1 版 1999 年 3 月 第 1 次印刷
书 号:ISBN 7-302-03320-X/TP·1786
印 数:0001~5000
定 价:50.00 元(带光盘)

译者前言

童年时玩过的积木块、七巧板等玩具至今仍时常引起我美好的回忆，那时的我时常沉浸在一种忘我的想象中，用那些简单的形状构造出自己能够想象得到的各种千奇百怪的物体。上了大学，接触到 UNIX，我又深深地为它所提供的各种功能专一、小巧灵活的实用工具所折服：通过管道，我可以把那些精巧的小程序组合起来，构造出各种功能复杂的工具，实现单个程序达不到的功能。

初次接触到 COM(组件对象模型)这个概念，我首先想到的就是积木玩具和 UNIX 中那些小巧的实用程序。人类孜孜以求的目标就是将自然改造得更加符合自己的需要，让自然物按自己的行为习惯和方式那样工作。而 COM 正是微软公司为了使计算机工业的灵魂——软件——的生产更加符合人类行为方式而开发的一种新的软件开发技术。

在 COM 架构下，人们可以开发出各种各样的功能专一的“软件”积木块(组件)，然后将它们按照需要“搭”起来，构成复杂的应用系统。由此带来的好处是多方面的：可以将系统中的组件用新的组件替换掉，以随时进行系统的升级与定制；可以在多个应用系统中重复利用同一个组件；可以方便地将应用系统扩展到网络环境下；COM 与语言、平台无关的特性使得所有的程序员均可充分发挥自己的才智与专长编写出组件模块；等等。

Dale Rogerson 是微软公司的资深技术人员，对微软各种技术的背景有着极为深刻的了解。由他所著的《COM 技术内幕》是一部全面、系统介绍 COM 技术的著作。为了及时将 COM 这一最新的软件开发方法与思想介绍给国内的读者，我们决定将这本书翻译出来，以期能够为国内的软件业的发展作出一定的贡献！

由于时间仓促，同时限于对 COM 的认识与理解程度，书中的错误与不足之处在所难免，恳请读者不吝指正。

译　者

1997.12 于北京　清华园

目　录

引言 …… Ⅸ
第1章　组件 …… 1
1.1　使用组件的优点 …… 2
1.1.1　应用程序定制 …… 2
1.1.2　组件库 …… 3
1.1.3　分布式组件 …… 3
1.2　对组件的需求 …… 4
1.2.1　动态链接 …… 4
1.2.2　信息封装 …… 5
1.3　COM …… 6
1.3.1　COM 组件是…… …… 7
1.3.2　COM 不是…… …… 7
1.3.3　COM 库 …… 8
1.3.4　COM 方法 …… 8
1.3.5　COM 超越了用户的需要 …… 8
1.4　本章小结 …… 9
第2章　接口 …… 11
2.1　接口的作用 …… 11
2.1.1　可复用应用程序架构 …… 12
2.1.2　COM 接口的其他优点 …… 13
2.2　COM 接口的实现 …… 13
2.2.1　编码约定 …… 14
2.2.2　一个完整的例子 …… 15
2.2.3　非接口通信 …… 18
2.2.4　实现细节 …… 18
2.3　接口理论:第二部分 …… 20
2.3.1　接口的不变性 …… 20
2.3.2　多态 …… 20
2.4　接口的背后 …… 21

2.4.1 虚拟函数表 …… 21
2.4.2 vtbl 指针及实例数据 …… 23
2.4.3 多重实例 …… 24
2.4.4 不同的类,相同的 vtbl …… 24
2.5 本章小结 …… 26
第 3 章 QueryInterface 函数 …… 27
3.1 接口查询 …… 28
3.1.1 关于 IUnknown …… 28
3.1.2 IUnknown 指针的获取 …… 29
3.1.3 关于 QueryInterface …… 29
3.1.4 QueryInterface 的使用 …… 30
3.1.5 QueryInterface 的实现 …… 31
3.1.6 关于类型转换 …… 32
3.1.7 一个完整的例子 …… 35
3.2 关于 QueryInterface 的实现规则 …… 40
3.2.1 同一 IUnknown …… 40
3.2.2 客户可以获取曾经得到过的接口 …… 41
3.2.3 可以再次获取已经拥有的接口 …… 41
3.2.4 客户可以从任何接口返回到起始接口 …… 42
3.2.5 若能够从某接口获取某特定接口,则从任意接口都将能够获取此接口 …… 42
3.3 QueryInterface 定义了组件 …… 43
3.3.1 接口集 …… 44
3.4 新版本组件的处理 …… 44
3.4.1 何时需要建立一个新版本 …… 46
3.4.2 不同版本接口的命名 …… 46
3.4.3 隐含合约 …… 46
3.5 本章小结 …… 47
第 4 章 引用计数 …… 48
4.1 生命期控制 …… 48
4.2 引用计数简介 …… 49
4.2.1 引用计数接口 …… 52
4.2.2 AddRef 和 Release 的实现 …… 53
4.3 何时进行引用计数 …… 60
4.3.1 引用计数的优化 …… 60

4.3.2 引用计数规则 …… 63
4.4 本章小结 …… 65
第 5 章 动态链接 …… 66
5.1 组件的创建 …… 66
5.1.1 从 DLL 中输出函数 …… 67
5.1.2 DLL 的装载 …… 69
5.2 客户和组件的划分 …… 71
5.2.1 程序清单 …… 72
5.3 对象串 …… 77
5.4 本章小结 …… 78
第 6 章 关于 HRESULT、GUID、注册表及其他细节 …… 79
6.1 HRESULT …… 79
6.1.1 HRESULT 值的查找 …… 81
6.1.2 HRESULT 值的使用 …… 83
6.1.3 用户自己代码的定义 …… 84
6.2 GUID …… 86
6.2.1 为什么要使用 GUID …… 86
6.2.2 GUID 的声明和定义 …… 87
6.2.3 GUID 的比较 …… 89
6.2.4 将 GUID 作为组件标识符 …… 89
6.2.5 通过引用传递 GUID 值 …… 90
6.3 Windows 注册表 …… 90
6.3.1 注册表的组织 …… 90
6.3.2 注册表编辑器 …… 90
6.3.3 CLSID 关键字结构 …… 91
6.3.4 关于注册表的其他细节 …… 92
6.3.5 ProgID …… 93
6.3.6 自注册 …… 95
6.3.7 组件类别 …… 96
6.3.8 OleView …… 97
6.4 COM 库函数 …… 98
6.4.1 COM 库的初始化 …… 98
6.4.2 内存管理 …… 98
6.4.3 将字符串转化成 GUID …… 99
6.5 本章小结 …… 101

第7章 类厂 …… 102
7.1 CoCreateInstance …… 102
7.1.1 CoCreateInstance 的声明 …… 103
7.1.2 CoCreateInstance 的使用 …… 103
7.1.3 类上下文 …… 104
7.1.4 客户程序清单 …… 105
7.1.5 CoCreateInstance 的不灵活性 …… 107
7.2 类厂 …… 107
7.2.1 CoGetClassObject …… 107
7.2.2 IClassFactory …… 108
7.2.3 CoCreateInstance 与 CoGetClassObject 的比较 …… 109
7.2.4 类厂的若干特性 …… 110
7.3 类厂的实现 …… 110
7.3.1 DllGetClassObject 的使用 …… 111
7.3.2 组件的创建过程 …… 111
7.3.3 组件代码清单 …… 112
7.3.4 流程控制 …… 119
7.3.5 组件的注册 …… 120
7.4 同一 DLL 中的多个组件 …… 121
7.4.1 类厂实现的复用 …… 122
7.5 DLL 的卸载 …… 123
7.5.1 DllCanUnloadNow 的使用 …… 123
7.5.2 LockServer …… 123
7.6 本章小结 …… 124
第8章 组件复用:包容与聚合 …… 125
8.1 包容和聚合 …… 126
8.1.1 包容简介 …… 126
8.1.2 聚合简介 …… 127
8.1.3 包容与聚合的比较 …… 127
8.2 包容的实现 …… 128
8.2.1 接口扩展 …… 131
8.3 聚合的实现 …… 132
8.3.1 QueryInterface 的实现 …… 133
8.3.2 不正确的 IUnknown …… 135
8.3.3 聚合的未知接口 …… 136

8.3.4 内部组件的创建 …… 140
8.3.5 外部组件中指向内部组件接口的指针 …… 143
8.4 一个完整的例子 …… 145
8.4.1 盲目聚合 …… 161
8.5 现实世界中的聚合和包容 …… 163
8.5.1 组件的内部状态信息 …… 163
8.5.2 虚拟函数的模拟 …… 165
8.6 本章小结 …… 166
第 9 章 编程工作的简化 …… 167
9.1 客户端的简化 …… 167
9.1.1 智能接口指针 …… 168
9.1.2 C++ 包装类 …… 179
9.2 服务器端的简化 …… 180
9.2.1 未知接口基类 …… 181
9.2.2 类厂基类 …… 185
9.2.3 CUnknown 和 CFactory 的使用 …… 191
9.2.4 集成步骤 …… 196
9.3 本章小结 …… 197
第 10 章 EXE 中的服务器 …… 198
10.1 不同的进程 …… 198
10.1.1 本地过程调用 …… 199
10.1.2 调整 …… 200
10.1.3 代理/残根 DLL …… 200
10.2 IDL/MIDL 简介 …… 202
10.2.1 关于 IDL …… 202
10.2.2 IDL 接口描述举例 …… 203
10.2.3 MIDL 编译器 …… 208
10.3 本地服务器的实现 …… 212
10.3.1 示例程序的运行 …… 212
10.3.2 去掉入口点函数 …… 213
10.3.3 类厂的启动 …… 213
10.3.4 对 LockServer 的修改 …… 217
10.4 远程访问能力 …… 219
10.4.1 DCOMCNFG.EXE 所完成的工作 …… 220
10.4.2 工作机理 …… 221

10.4.3 其他 DCOM 信息 …… 222
10.5 本章小结 …… 224
第 11 章 调度接口与自动化 …… 225
11.1 一种新的通信方式 …… 226
11.1.1 旧的通信方式 …… 226
11.1.2 IDispatch 接口 …… 226
11.2 IDispatch 的使用 …… 230
11.2.1 Invoke 函数的参数 …… 232
11.2.2 示例 …… 237
11.2.3 VARIANT 类型 …… 239
11.2.4 BSTR 数据类型 …… 241
11.2.5 SAFEARRAY 类型 …… 242
11.3 类型库 …… 243
11.3.1 类型库的创建 …… 243
11.3.2 类型库的使用 …… 246
11.3.3 注册表中的类型库 …… 247
11.4 IDispatch 接口的实现 …… 248
11.4.1 异常的引发 …… 250
11.4.2 参数调整 …… 251
11.5 本章小结 …… 252
第 12 章 多线程 …… 253
12.1 COM 线程模型 …… 254
12.1.1 Win32 线程 …… 254
12.1.2 COM 线程 …… 254
12.1.3 套间 …… 255
12.1.4 套间线程 …… 257
12.1.5 自由线程 …… 258
12.1.6 调整与同步 …… 258
12.2 套间线程的实现 …… 260
12.2.1 自动调整 …… 261
12.2.2 手工调整 …… 261
12.2.3 编码 …… 262
12.2.4 对套间线程例子的说明 …… 263
12.3 自由线程的实现 …… 270
12.3.1 对自由线程例子的说明 …… 271

12.3.2 自由线程参数调整的优化 …… 276
12.4 关于线程模型的注册表关键字 …… 278
12.5 本章小结 …… 278
第 13 章 一个完整的例子 …… 280
13.1 Tangram 程序 …… 281
13.1.1 Tangram 的运行 …… 281
13.1.2 所用的组件 …… 282
13.1.3 客户程序 …… 282
13.1.4 TangramModel 组件 …… 283
13.1.5 TangramGdiVisual 和 TangramGLVisual 组件 …… 284
13.1.6 TangramGdiWorld 和 TangramGLWorld 组件 …… 285
13.2 展示 …… 286
13.3 IDL 文件 …… 286
13.3.1 DLLDATA.C 文件 …… 287
13.4 循环引用计数 …… 287
13.4.1 不调用 AddRef …… 288
13.4.2 使用显示终止 …… 288
13.4.3 使用一个单独的组件 …… 289
13.5 事件和连接点 …… 290
13.5.1 IEnumXXX …… 292
13.6 本章小结 …… 293
结束语 …… 294

引　言

作为一个软件开发人员，你是否有过在发送了你的应用程序之后而又想对它进行修改或给它加上一些新的特性呢？你是否想逐步地开发你的程序而不是每两年就将其完全重写一次呢？你是否想使你的应用程序能够在更高的程度上被定制，更灵活、更具有动态性呢？你是否想加速你的应用程序开发进度呢？你是否需要开发分布式的应用程序呢？你是否希望能够按开发非分布式应用程序那样开发分布式的应用程序呢？

你对组件编程感兴趣吗？你是否想将你的应用程序划分成组件呢？你想学习什么是COM吗？你想学习(OLE)自动化吗？你在学习 OLE 的过程中是否被弄得焦头烂额呢？你是否认为 COM 和 OLE 都是极为困难的呢？你是否想了解 Microsoft 的一些技术，如 ActiveX、DirectX 以及 OLE 的基础呢？你是否想定制 Microsoft 的各种应用程序或操作系统呢？

若你对上述问题的答案都是“是”，那么本书对你而言实在是再合适不过了。上述所有的问题都由一种技术关联着，即 Microsoft 组件对象模型(Component Object Model)，也即大家都熟知的 COM。本书所介绍的就是如何用 C++ 开发定制的 COM 组件。

COM 是开发软件组件的一种方法。组件实际上是一些小的二进制可执行程序，它可以给应用程序、操作系统以及其他组件提供一些服务。开发定制的 COM 组件就如同开发动态的、面向对象的 API。多个 COM 对象可以连接起来以形成应用程序或组件系统。并且组件可以在运行时、在不重新链接或编译应用程序的情况下被卸下或替换掉。Microsoft 的许多技术，如 ActiveX、DirectX 以及 OLE 等都是基于 COM 而建立起来的。并且 Microsoft 的开发人员也大量地使用 COM 组件来定制他们的应用程序及操作系统。

预备知识要求

本书是针对具有一些 Win32 编程经验的中级 C++ 程序员而写的。但初级 C++ 程序员也不必因此望而生畏。因为使用 C++ 开发 COM 组件实际上并不是很难，也不需要什么高级的 C++ 技巧。举个例子来说，本书中所用到的最高级的 C++ 功能也不过是多重继承，但为进行 COM 编程，读者所需了解的多重继承的细节将不会超出本书所涉及的那些内容。对于初级 C++ 程序员而言，学习 COM 还有一个好处就是 COM 将帮助他们形成良好的程序设计风格。当然初学者阅读 C++ 方面的一些专门著作肯定也是有极大好处

的。

Microsoft Windows 的编程经验对于学习本书所介绍的内容将是有帮助的，但这些经验并不是必需的。在本书中作者尽量避免使用特定于 Windows 的代码。因此，UNIX 用户在理解书中的示例代码也应该是没有问题的。Windows 程序员比之非 Windows 程序员具有的一个优势是他们对开发 Windows 应用的各种工具比较熟悉。

另外要说明的是 Microsoft 基本类库(MFC)及其使用经验并不是必需的。实际上 MFC 并没有给开发 COM 组件提供任何好处。正因如此，本书前 12 章所给出的示例代码中并没有使用 MFC。

写给非 Windows 程序员

对于 UNIX、Macintosh、Linux、VMS 或其他操作系统的开发人员，也将能够从本书所介绍的内容获益。因为 COM 所蕴含的概念并不是只在 Microsoft Windows 操作系统下才有效。COM 并不是一个大的 API，它实际上像结构化编程及面向对象编程方法那样，也是一种编程方法。在任何一种操作系统中，开发人员均可以遵循“COM 方法”。当然，Windows 提供了一些按“COM 方法”进行编程时可以使编程任务得以简化的代码。但这些代码有许多是可以在开发人员所偏爱的操作系统上方便地重新建立的。如果不想自己重新建立这些代码，也并不是说 COM 方法不能用了。Microsoft 正在开发 COM 的 Macintosh 版本，并且 Software AG 正在努力将 COM 移植到几乎所有的现有操作系统上。因此在不久的将来，任何操作系统上的开发人员都将可以使用一个标准的、兼容的 COM 版本。

C++

虽然 COM 本身是与编程语言无关的，但开发人员总得选用一种编程语言来编写所需的组件。可供选择的语言可以是 C、Java、Python、甚至可以是 Microsoft Visual Basic。但是许多组件是用 C++ 编写的，而且这种状况还会继续下去。因此，本书所给出的示例代码均为 C++ 代码。使用同一种语言的好处是可以按照一种具体的方式来讲述 COM 规范，这样将使之更容易理解。即使读者最终决定使用 Java 或 Python 来实现所需的组件，但在用 C++ 从头开发 COM 组件的过程中所获得的知识对于读者也将是有莫大好处的。

只使用传统的 C++ 语言

由于并不是所有的 C++ 编译器都支持加入到 C++ 语言中的最新成分，故作者在本书中将尽量避免使用这些最新的功能，例如在本书中将不会用到 bool 关键字及类似 mutable 之类的关键字。除了在第九章中用到的一些方便的接口指针类之外，本书中的其他部

分都没有用到模板类,因为这些内容将使本书所介绍的概念复杂化而导致理解上的困难。

本书中使用到了一些新的类型转换操作符,如 static_case、const_cast 以及 reinterpret_cast 等,这主要是考虑到这些操作符在 Microsoft Visual C++ 中出现的时间已经比较长了。读者可以用这些新的类型转换操作符来代替那些老的类型转换方式。例如可以将

```
CFoo* pI = (Cfoo*)this;
```

写成

```
CFoo* pI = static_cast<CFoo*>this;
```

但在某些情况下,为了保证程序的易读性,作者仍将使用老的类型转换方式。

关于本书中的示例代码

本书中的每一章都包含一到两个示例程序。作者尽量使这些程序简短而完整。短的例子阅读起来比较容易。因为所有代码总共也不过那么几页。对于那些较长的例子,需要读者阅读本书所附 CD 上的那些文件。使用简单的例子的另一个好处是我们可以将讨论的重点放在 COM 组件的需求上,而无需读者费大力去搞清楚复杂的例子中的那些不必要的细节及复杂性。

下列特点是本书中所有示例程序所共有的:

- 所有示例代码在 CD 上都是预编译好的,均可以在 Microsoft Windows 95 或 Microsoft Windows NT 上直接运行。
- 所有在正常工作前需要进行系统注册的示例程序都有一个名为 REGISTER.BAT 的批处理文件。使用此文件可以完成所需的注册。
- 所有的示例代码均尽量少用 Win32 API。
- 示例代码均没有用到 Microsoft 基本类。
- 所有示例的完整代码均可在所附的 CD 上找到。为编译这些文件,只需要一个 C++ 编译器及 Win32 SDK 中的头文件及库文件。若读者安装有 Visual C++,则不需要 Win32 SDK,因 Visual C++ 包含了所需的所有成分。实际上几乎所有的与 Windows 兼容的编译器都包含这些成分。
- 各示例程序可以用 Visual C++ 4.x 或 5.0 进行编译。当然这并不是说一定要用 Visual C++ 来编译这些代码。作者在编写这些代码的时候正好在进行 Visual C++ 5.0的开发工作,故选择了 Visual C++ 5.0 来编译它们。
- 所有示例均可以方便地从命令行进行编译。例如使用 Visual C++,则输入 *cl <filename>* 即可完成大多数示例的编译工作。
- 对于复杂一些的例子,作者编写了相应的简单的可供 Visual C++ 的编译器使用的 makefile。为编译这些示例代码,可输入 *nmake -f makefile* 或 *nmake*。为使这些 makefile 简单易读,在各 makefile 中仅包含一个调试版本。

Tangram 示例应用程序

在本书所附 CD 中还包含一个使用 COM 组件构造的完整的应用程序：Tangram。但 Tangram 可以说是严重违反了上述的那些规则。首先它不仅大量地使用了 Win32 API，特别是 GDI，并且它还使用了 MFC 及 OpenGL。其次，它并不简单，其中包含分布在几个 DLL 及 EXE 中的多个组件。可以说 Tangram 向读者展示了“现实世界”中的 COM，而书中的其他示例则更像“象牙塔”中的 COM。Tangram 的源程序及可执行代码均可以在所附 CD 上找到。

编码风格

虽然在大多数例子中并没有使用 MFC，但各示例程序均是按 MFC 风格而写成的。例如在成员变量的前面均加有前缀 *m_*。这样对于类似于 *m_SleepyBear* 的变量，读者一看就可知道它是一个成员变量。另外所有类的名称前面均有一个大写的字母 *C*。例如 *CCozyBear* 表示的是类 Cozy Bear 的名称。表 I-1 列出了书中所用的其他一些前缀。

表 I-1 MFC 编码风格前缀示例

前缀	含义	示例
C	类	*CConnectionPoint*
I	接口	*IConnectionPoint*
*m_*	成员变量	*BOOL m_bSpIleyBear;*
*s_*	静态成员变量	*static int s_iBears;*
*g_*	全局变量	*int g_Bears[100];*

对于有 Windows 编程经验的读者，对于匈牙利记法一定不会陌生。匈牙利记法实际上是一套用变量类型来标记变量名称的约定。在本书中作者用了匈牙利记法的一个子集，如表 I-2 所列。可以看到本书所用的一些记法带有一些任意性，因为它部分是由其他 COM、OLE 及 ActiveX 的开发人员所建议的匈牙利记法而得到的。

表 I-2 本书所用匈牙利命名规范

前缀	含义	示例
p	指针	*int * pCount;*
pI	指向接口的指针	*Ibear * pIBear;*
b	布尔型	*BOOL bBear;*
I	整数	*int iNumberOfBears;*
dw	*DWORD*	*DWORD dwBear;*

续表

前缀	含义	示例
c	计数	*DWORD cRefs;*
sz	字符数组	*char szName[] = "Fuzzy";*
wsz	宽字符数组	*wchar_t wszName[] = L"Fuzzy";*

为什么要写这本书?

不知读者学过物理没有?如果读者修过基于微积分的物理课程,那么很明显,微积分将是一门所需的先修课。在微积分课上,读者将学会如何将这方面的知识应用到其他各种各样的领域中。只有在学会并理解了微积分之后,才能将所得到的这些在数学方面的技能应用到物理问题的解决中。但并不是在所有的情况下都需要遵循这种学习次序。实际上当初牛顿发明微积分的主要目的是要用它来解决经典物理中的机械及动力学问题。只是在后来微积分才作为一种有效的工具被用于物理之外的学科中。

COM 和 OLE 的关系非常类似于上面我们所讲到的微积分与物理的关系。所不同的只是微积分是用来解决物理问题的,而 COM 则是用来解决诸如如何将一个电子工作表嵌入到字处理程序之类的问题。此类问题的解决方案正是大家所熟知的 OLE。关于 OLE 的著作已经是相当丰富了,但讲 COM 的似乎还没有。关于 OLE 的第一本最好、最全面的著作当数 Kraig Brockschmidt 的《Inside OLE》了。

当 Kraig 在编写他那本书时,COM 只有唯一的一个应用,即 OLE。任何一个想学习 COM 的人也都需要学习 OLE,虽然这两部分内容是相互关联的。这一点同早期的微积分与物理的关系是类似的。那时候,如果不学习物理,是用不着学习微积分的。

到今天,可以说 COM 已经是无处不在了。并且可以预计 COM 将变得比 OLE 更为重要。Microsoft 现在已经开发出了许多同 OLE 没有任何关系的 COM 接口及组件,如 Microsoft的三维图形 API Direct3D。正因如此,在 Nigel Thompson 在编写《3D Graphics Programming for Windows 95》时,他要在书中加入一章讲述如何使用 COM,虽然这有点像老师在正式进入物理课程的讲授时要花一点时间来复习一下微积分一样。这样做的结果是学生并没有理解物理课的内容,他们只是在那儿生搬硬套地处理那些公式。

这本书的目的之一就是将 COM 同 OLE 分开并重点讲述 COM 的有关概念。在此作者将根本不会涉及到任何有关 OLE 的内容,而只是讲述有关 COM 的一般性机制。在读者学会了这些机制之后,就可以方便地将它们用到 OLE、DirectX 以及 ActiveX 组件的开发中,就如同可以用微积分来解决除物理问题之外的其他许多问题一样。

所以,若读者真的有兴趣搞清楚建立 COM 组件的方法,那么这本书将再适合不过了。读者可以把所学到的技术应用于 ActiveX、OLE 及自己的组件的开发中。未来将是 COM

的天下,而这本书将帮助读者把握未来。至少它可以帮助读者如何更有效地在 C++ 程序中使用多重继承。

技术支持

虽然作者竭尽全力保证书中内容及所附 CD 上内容的准确性,但错误可能还是在所难免。为此 Microsoft Press 在如下的 www 站点提供关于它所出版书籍的更正内容:

http://www.microsoft.com/mspress/support/

读者关于本书及所附 CD 的评论、疑问及其他意见可以使用下列方法反馈给 Microsoft Press:

邮件:

Microsoft Press
Attn: *Inside COM* Editor
One Microsoft Way
Redmond, WA 98052-6399

电子邮件:

MSPINPUT@MICROSOFT.COM

应说明的是,上述邮件地址并不提供产品支持。当读者需要这方面的支持时,可以使用如下的一些方法。

首先是 Microsoft Developer Network(MSDN) Web 站点,www 地址为:

http://www.microsoft.com/MSDN/

为充分利用 MSDN 所提供的各种服务,可以订购之。订购信息可以在此站点获得,可拨打(800)759-5474。

在下述 Web 站点,Microsoft 提供了大量的支持信息(包括已发现的问题及解决方法等):

http://www.microsoft.com/support/

有关 COM 的问题,可以打电话给 Microsoft 的 Win32 SDK AnswerPoint 人员。电话号码是(800) 936-5800。

有关 Microsoft Visual C++ 的问题,可以于工作日早六点至晚六点(西部时间)拨打标准的支持电话(206)635-7007。

第1章 组 件

一个应用程序通常是由单个的二进制文件组成的。当编译器生成此应用程序之后，在对下一版本重新编译并发行新生成的版本之前，应用程序一般不会发生任何变化。操作系统、硬件及客户需求的改变都必须等到整个应用程序被重新编译之后才能够得以认可。整个软件工业就这样随着已发行软件的日益“老化”而奔向未来。

目前这种状况已经发生了变化。人们认识到应用程序在发行之后不应保持那种静止的状态。开发人员应找出一种方法，以能够给已经发行的软件不断地注入新的活力。这种解决方案就是将单个的应用程序分隔成多个独立的部分，也即组件(如图 1-1 所示)。

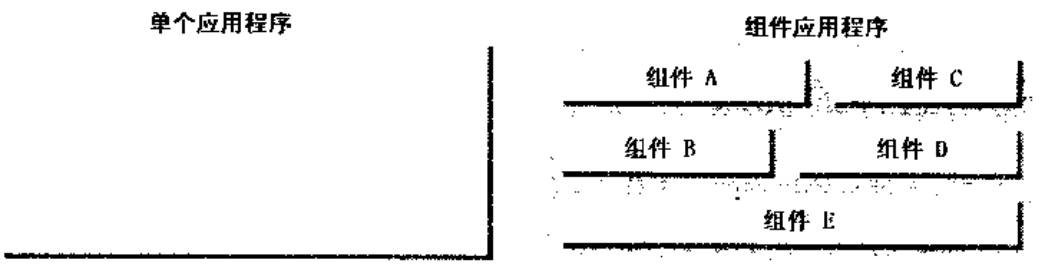


图 1-1 将单个应用程序(左)分割成多个组件(右)，可使之适应各种变化

这种作法的好处是可以随着技术的不断发展而用新的组件取代已有的组件(参见图 1-2)。此时的应用程序将不再像以前那样是一个在发行之前就已命中注定要过时的静态实体，而是可以随着新组件不断取代旧的组件而趋于完善。并且从已有的组件可以建立全新的应用。

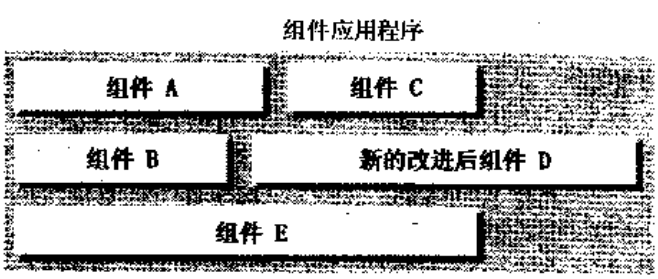


图 1-2 用改进后的新组件 D 取代原有的组件 D

传统的作法是将应用程序分割成文件、模块或类，然后将它们编译并链接成一个铁板一块状的应用程序。它与组件建立应用程序的过程(称为是组件架构)有很大的不同。一

个组件同一个微型应用程序类似，即都是已经编译、链接好并可以使用的了，应用程序就是由多个这样的组件打包而得到的。各定制的组件可以在运行时同其他组件连接起来以构成某个应用程序。在需要对应用程序进行修改或改进时，只需将构成此应用程序的组件中的某个用新的版本替换掉即可。

当然将铁板状的应用程序拆分成组件需要一个强大的"锤子"。我们所用的"锤子"就被称作是 COM。COM，即组件对象模型，是关于如何建立组件以及如何通过组件建构应用程序的一个规范。四年前，Microsoft 就已经开始开发 COM 了，其目的是使 Microsoft 的各种软件产品更加灵活、更具有动态性并且更易于定制。目前 Microsoft 的几乎所有应用程序都使用了 COM。Microsoft 的 AcitveX 技术就是基于 COM 而建立起来的。

本书讲述的是如何使用 C++ 来建立 COM 组件。通过本书所给出的例子，读者将学会如何建立可用于组装应用程序的 COM 组件，这些应用程序不但可随时间的流逝而"幸存"，而且可逐步发展、进化。

在详细讨论 COM 的有关细节之前，我们先来看一下使用组件架构有哪些好处，以及使用组件来建立应用程序有哪些需求。

1.1 使用组件的优点

前面已经提到过组件架构的一个优点：应用程序可随时间的流逝而发展进化。读者将会发现，除此之外，使用组件还有一些可以使对已有应用程序的升级更加方便和灵活的优点，如应用程序的定制。组件库以及分布式组件等。

1.1.1 应用程序定制

用户通常希望能够定制他们所用的应用程序（这一点如同房主总是想根据他们的爱好来定制房间一样），以使应用程序能够按他们自己的工作方式那样工作。在编程人员建立对实用应用程序的定制方案时，他们需要的是可被修改的应用程序。而组件架构从

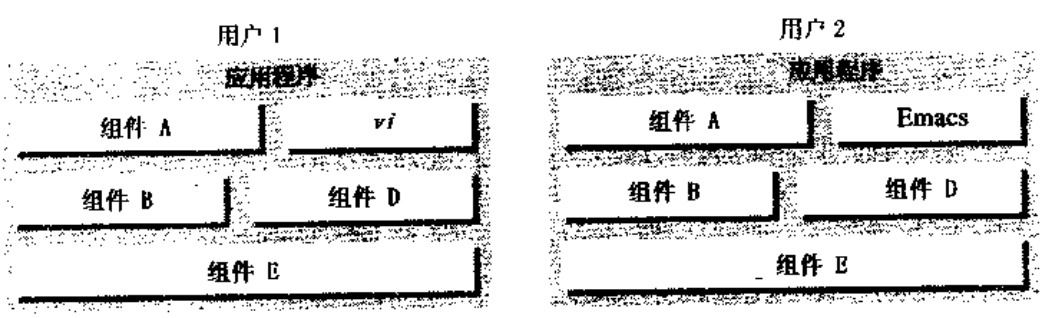


图 1-3 从组件建立应用程序可使用户进行更高程度的定制。在本例中，用户 1 使用的编辑器是 *vi*，而用户 2 使用的则是 Emacs

本质上讲就是可被定制的，因用户可以用更能满足他们需要的组件来将某个组件替换掉。

例如，假定某些组件是基于编辑器 *vi* 和 Emacs 的。在图 1-3 中，用户 1 将应用程序配置为使用 *vi*，而用户 2 更喜欢使用 Emacs。按照此种方式，可以加入新的组件或改变已有组件而方便地定制应用程序。

1.1.2 组件库

组件架构最引人注目的优点之一是快速应用程序开发。这一优点可以使开发人员从某个组件库中取出所需的组件并将其快速地组装到一块以构造所需的应用程序，如同搭积木块一样。

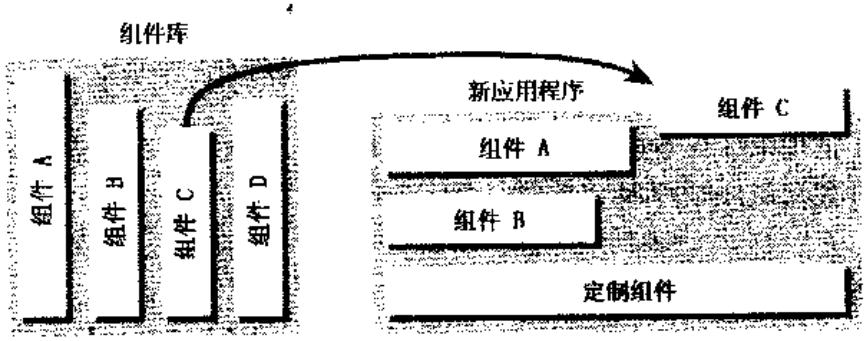


图 1-4 组件可以被存放到库中，并可被取出来以快速地开发应用程序

这种从标准的部件构造应用程序的作法，很长时间以来一直是软件工程师们的一个未曾实现的梦想。但是现在这一梦想随着 AcitveX 控件（以前被称作是 OLE 控件）的开发而正在被变为现实。Visual Basic、C、C++ 以及 Java 程序员都可以利用 AcitveX 控件以加速应用程序及 Web 页面的开发。当然所有的应用程序都不可避免地需要一些特殊的组件，虽然如此，一个应用程序的大部分仍然可以用标准的组件建立起来。

1.1.3 分布式组件

随着网络带宽及其重要性的提高，对由分布在网络上的多个部分组成的应用程序的需求毫无疑问将保持强劲的增长势头。组件架构可以使得开发这类分布式应用的过程得以简化。实际上客户机/服务器式的应用就是向组件架构跨出的第一步。在那里整个应用程序被分成两部分：客户机部分及服务器部分。

在将已有应用程序转化成分布式应用程序时，若已有应用程序是由组件组装成的，那么转化过程将会简单得多。这是因为，首先，应用程序已经被划分成可以位于远地的各外功能部分。其次，由于任一组件均是可以被替换的，由此可以将某个组件替换专门负责同某个远地组件通信的组件。例如在图 1-5 中，组件 C 和组件 D 被放到了网络上远地机器上。在本地，它们被替换成了两个新的组件：Remoting C 和 Remoting D。这两个新的组件

的作用是将其他组件发来的请求通过网络转发给 Component C 和 Component D。本地机器上的应用程序并不需要知道实际所用到的组件到底在何处。类似地，远地组件也不需要知道它们是否位于远地。这样通过加入合适的远地组件，应用程序完全不需要知道实际的组件到底在那里。

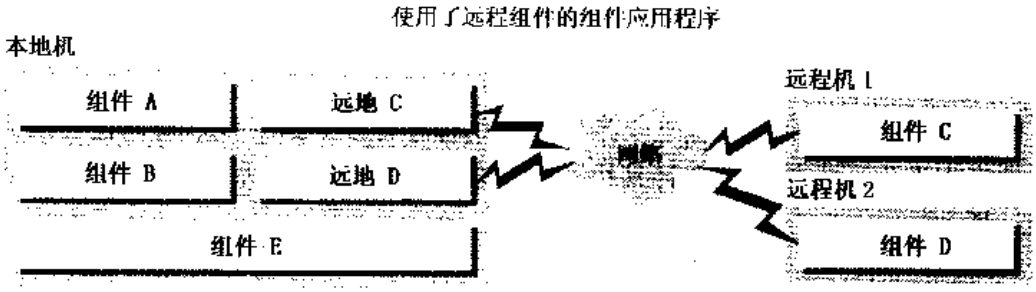


图 1-5 位于网络上某远地机上的组件

在了解了使用组件有那些优点之后，下面来看一下建立组件有那些需求。然后我们再来看一下在建立组件时 COM 将起到一个什么样的角色。

1.2 对组件的需求

使用组件的种种优点直接来源于可以动态地将它们插入或卸出应用程序。当然为了实现这种功能，所有的组件必须满足两个条件。第一，组件必须动态连接。第二，它们必须隐藏(或封装)其内部实现细节。很难说这两个需求哪个更为重要，正如我们说不清楚到底是先有鸡还是先有蛋一样。实际上这两个需求是互相依赖的。作者认为，动态链接对于组件而言是一个至关重要的需求，而信息隐藏则是动态链接的一个必要条件。下面我们具体看一下这两方面的需求。

1.2.1 动态链接

我们的最终目标是使用户在应用程序的运行过程中能够将组件替换掉。虽然并不是在所有的应用程序中都需要给用户提供这种控制，但我们却希望能够拥有某种手段来支持此种控制能力。实际上，为在运行时能够改变组件，要求我们能够将组件动态地链接到一起。

为理解动态链接在这里的重要性，读者可以设想一下对于一个由组件构成的，但不能在运行时进行链接的应用程序，当用户需要改变其中的某个组件时会发生什么。此时开发人员不得不将整个程序重新链接或编译一遍，然后重新发行新的版本。但是这种重新编译或链接对于最终用户而言几乎是不可能的事情。即使他们知道如何重新链接，但他们可能没有链接程序，或者没有合适的链接程序。这种需要在每次改变一个组件时就将

其重新链接一遍的程序同传统的铁板一块的程序实际上没有什么差别。

1.2.2 信息封装

下面我们来看一下为什么动态链接需要信息封装。为组成一个应用程序，需要将各组件连接起来。当需要将某个组件用新的组件替换掉时，需要将此组件同系统断开，然后将新的组件连上去。显然新的组件必须按同样的方式连接到系统中，否则将需要重新编写、重新编译或重新链接这些组件。不论组件或应用程序是否支持动态链接，如果改变了某个组件同其他组件的连接方式，那么整个系统的整体性实际上被破坏了，此时至少需要将整个系统重新编译一遍，甚至可能需要重新编码。

为说明上述情况为什么需要信息封装，下面先定义一些术语。对于一个应用程序或组件，如果它使用了其他组件，那么我们称之为一个客户，客户通过接口同其他组件进行连接。如果某个组件发生了变化但其接口没有任何变化，那么它的客户将不需要进行任何修改。类似地，若客户发生了变化但没有改变其接口，那么它所连接的组件也不需要任何改变。但是如果对客户或组件的修改导致了对接口的修改，那么接口的另一方也应发生相应的变化。

因此为充分发挥动态链接的功能，组件及客户都应尽其可能不要改变它们的接口。这意味着它们必须被封装起来。也就是说，组件及客户的内部实现细节不能反映到接口中。接口同内部实现细节的隔离程度愈高，组件或客户发生变化时对接口的影响将越小。在接口没有发生任何变化时，对组件的修改将几乎不会对应用程序的其他部分产生什么影响。

这种将客户同组件实现相应隔离开来的要求对于组件加上了一些限制，如下所列：

1 组件必须将其实现所用的编程语言封装起来。任一客户都应能使用任一组件，不论它们是用什么编程语言实现的。将实现用的编程语言暴露出来只会在组件及客户间引入新的依赖。

2 组件必须以二进制的形式发布。如果想将实现组件的编程语言隐藏起来，那么在发布时它们必须是已被编译、链接好并且马上就可以投入使用的。

3 组件必须可以在不妨碍已有用户的情况下被升级。一个组件的新版本必须既能够同老版本的客户一起使用，也可以同新版本的客户一起使用。

4 组件在网络上的位置必须可以被透明地重新分配。组件及使用它的程序应能够在同一进程中、不同的进程中或不同的机器上运行。客户对远程组件的处理方式应与对本地组件的处理方式是一样的。否则，当将某个本地组件移动到网络上的另外某个地方时，客户程序必须被重新编译。

下面我们详细讨论上述各点。

一、与语言的无关性

许多人实际上并不相信与语言的无关性对于组件的重要性。为方便讨论起见，假定某个应用程序只能使用由 Objective C 编写的组件进行定制。这样一来，将不会有太多的人来为此应用程序编写组件，因为大多数编程人员使用的是 C++。以后我们可能会用 C++ 来编写组件，从而使得应用程序可用的组件多起来。但如果后来另外有一种语言，假设是 EspressoBeans 变得流行起来，导致大家都放弃 C++，转而使用这一语言。为了使应用程序不被淘汰，我们需要用这种新语言来编写组件。如此就有了三种截然不同的编写组件的方法。此时我们将在市场上陷入困境。在我们的市场部分中更多的用户使用的是 Visual Basic，但竞争对手却允许其客户使用任何一种语言来编写组件，包括 Visual Basic。这样他们将在市场竞争中幸存下来。

在一个与语言无关的架构中，任何人均可以编写组件，并且这些组件不会因为编程语言的发展而过时。这种架构将极大的促进软件市场的繁荣。

二、版本

用户可能会拥有使用同一个组件的两个客户应用程序。假定其中的某个应用程序被指定使用某个组件的新版本而另一个应用程序被指定来使用其老版本。显然人们希望老应用程序仍能够使用新安装的新版本组件。在图 1-6 中，老应用程序使用的是新的 *vi* 组件，如同新应用程序一样。

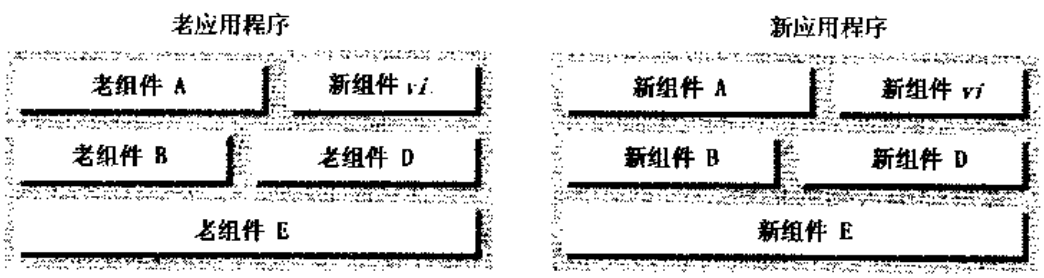


图 1-6 新的组件不应妨碍老组件，并且应能增强其他新组件的功能

但这种向后兼容的能力不应对组件的发展造成限制。强行改变某个组件的功能以使之适应新应用程序的需要、同时使其能支持老的应用程序仍应是可能的。

下面我们来看一下 COM 在建立组件方面如何满足这些需求。

1.3 COM

COM 是一个说明如何建立可动态互变组件的规范。它提供了为保证能够互操作，客户和组件应遵循的一些标准。标准对于组件架构的重要性同其他任何一个具有可改变部

分的系统是一样的。例如,如果没有 VHS(家用录像系统)录像带的标准,那么将可能会有人造出能够在 VCR 上使用的录像带。花园水管的尺寸及户外龙头的尺寸也是由标准所控制的。PCMCIA 卡及其插槽的制造也需要遵循一系列的标准。电视或收音机所接收的信号也是由标准规定的。当某个系统的各不同部分是由在不同国家、不同组织工作的不同的人制造出来时,标准的重要性就更为突出了。没有标准,任何东西都将不能一起工作。甚至在 Microsoft 我们也有一些要遵守的编程标准。(至少大数情况下我们是遵守这些标准的。)

COM 规范就是一套为组件架构设置标准的文档。本书中所开发的所有组件都将遵循这一标准。在本书所附 CD 上包含有这一规范的文本。但读者可能会问:那么 COM 到底是什么呢?

1.3.1 COM 组件是……

COM 组件是以 Win32 动态链接库(DLLs)或可执行文件(EXEs)的形式发布的可执行代码组成的。遵循 COM 规范编写的组件将能够满足对组件架构的所有需求。

COM 组件是动态链接的。COM 使用 DLL 将组件动态链接起来。但前面我们已经看到,动态链接本身并不能满足对于组件架构的需求。为满足这些需求,组件还必须是封装的。

对于 COM 组件的封装是很容易的,这是因为它们能够满足下面的一些限制条件:

- COM 组件是完全与语言无关的。任何过程性语言,从 Ada 到 C 到 Java 到 Modula-3 到 Oberon 到 Pascal 均可用来开发组件。并且任何一种语言均可以被修改得能够使用组件,如 Smalltalk 及 Visual Basic 等。实际上,我们是有办法编写出能够被宏语言使用的组件的。
- COM 组件可以以二进制的形式发布。
- COM 组件可以在不妨碍老客户的情况下被升级。在第 3 章中我们将会看到,COM 提供了一种实现同一组件不同版本的标准方法。
- COM 组件可以透明地在网络上被重新分配位置。对远程机器上的组件同本地机器上的组件的处理方式没有什么差别。

COM 组件按照一种标准的方式来宣布它们的存在。使用 COM 的发布方案,客户可以动态地找到它所需的组件。

COM 组件是一种给其他应用程序提供面向对象的 API 或服务的极好方法。对于可用于快速构造应用程序、与语言无关的组件库的建立,COM 组件也是不在话下。

关于 COM 组件是什么还可以说出好多,但 COM 却不是某些人会误认为是的那些概念。

1.3.2 COM 不是……

COM 并不是一种计算机语言。将 COM 同某种计算机语言相提并论是没有意义的。

例如比较 C++ 和 COM 谁优谁劣就没有什么意义,因 COM 和 C++ 是各自有着不同的用途。COM 所说明的是如何编写组件,但具体选用什么语言来编写则完全是自由的。在本书中,我们将使用 C++ 来开发 COM 组件。

将 COM 同 DLL 相比或相提并论也是不合适的。实际上 COM 是使用了 DLL 来给组件提供动态链接的能力。在作者看来,利用 DLL 动态链接能力最佳的方法是 COM。对于可用 DLL 解决的任一问题,均可以用 COM 组件更好地加以解决。对作者而言,除了用来支持 COM 组件外,在其他地方一般不会用到 DLL。这正是因为 COM 可以有效地利用 DLL。

COM 也并不是像 Win32 API 那样的一个函数集。它并没有提供类似于 *MoveWindows* 这样的服务。(但 COM 确实也提供了一些组件管理服务,下面我们将要讲到,)相反,COM 更主要的是一种编写能够按面向对象 API 形式提供服务的组件的方法。COM 也并不是类似于 Microsoft 基本类库(MFC)这样的 C++ 类库。COM 给开发人员提供的是一种开发与语言无关的组件库的方法,但 COM 本身并没有提供任何实现。

1.3.3 COM 库

COM 当然并不仅仅是一套 API——它也确实需要一些具体的实现。COM 具有一个被称作是 COM 库(COM Library)的 API,它提供的是对所有客户及组件都非常有用的组件管理服务。当在非 Windows 系统下开发 COM 风格的组件时,要实现此 API 中的大多数函数并不是一件太难的事情。COM 库可以保证对所有组件大多数重要的操作都可以按相同的方式完成。同时 COM 库也可以节省开发人员花在他们自己组件及客户的实现上的时间。COM 库中的大多数代码均可以支持分布式或网络化的组件。Windows 系统上分布式 COM(DCOM)的实现中提供了一些同网络上其他组件通信所需的代码。这不但可以节省开发人员花在网络编程上的时间,而且可以使得他们无需了解如何进行网络编程。

1.3.4 COM 方法

对作者而言,COM 最值得称道的地方也许就是我们可以将其作为一种编写程序的方法。例如,可以在任何操作系统上使用任何编程语言按 COM 风格进行编程。为编写类 COM 组件,开发人员并不需要任何在 Windows 系统上实现的 COM 代码。本书前八章中的示例代码可以很容易地修改成不使用任何 Windows 代码。COM 充分体现了组件编程的概念。同结构化编程及面向对象的编程方法一样,COM 也是一种组织软件的方法。好的软件设计概念在 COM 规范中得到了充分的体现。

1.3.5 COM 超越了用户的需要

COM 可以满足前面讨论过的所有组件架构的需要。它使用 DLL 来提供可在运行时被替换掉的组件。COM 借助于如下一些手段保证这些组件可以充分利用动态链接所带来的各种好处:

- 提供了一个所有组件都应遵守的标准。
- 允许使用组件的多个不同版本，而且这一点对于用户而言几乎是透明的。
- 使得可以按相同的方式来处理类似的组件。
- 定义了一个与语言无关的架构。
- 支持对远程组件的透明链接。

为实现上述目标，COM 强制开发人员必须将客户和组件严格地隔离开。COM 所具有的能力正是来源于这种强制性的隔离。对 COM 历史感兴趣的读者可阅读下面的内容。

COM 简史

开发 COM 的目的是为了使应用程序更易于定制、更为灵活。最初的目标是提供对对象链接及嵌入的支持，其基本的想法是以文档为中心的观点。在这种观点下，用户可以从一个字处理程序中编辑电子表格。对象链接及嵌入的 Microsoft 版本被称作是 OLE。OLE 的第一个版本使用动态数据交换(DDE)作为客户及组件之间的通信方式，并没有引入 COM。DDE 是在 Windows 消息传递架构的基础上建立起来的。对于 DDE，作者所能说的最客气的话也许是它确实能够工作。但 DDE 非常之慢，并且编写出能够正确工作的 DDE 代码是一件相当困难的工作。另外 DDE 也并不太健壮和灵活。总而言之，必须找出一种更好的方法。

解决问题的方法就是 COM。相对 DDE 而言，COM 更小、更快、也更加强壮和灵活。由于这个原因，OLE 的第二个版本使用了 COM 而不再使用 DDE。这样 COM 就成了 OLE 的新的基础。但 OLE 是开发出来的第一个 COM 系统。由于这个原因，OLE 并不能很好地说明 COM 的功能与作用。由于实现上的原因，OLE 显然比较庞大、缓慢，并且难于编码。但这并不是由于使用 COM 而造成的。

要记住的是，OLE 确实是想试图解决以前未曾解决过的一些问题。使用 OLE，用户可以将一个图片从一个厂家的绘图程序中引入到另一厂家的字处理软件中，并可在不离开字处理软件的情况下编辑此图片。OLE 试图使用户能够不受任何限制地完成这些操作。它并没有强制客户和组件只能建立一些受限的连接，相反它允许客户和组件之间建立丰富的连接。组件可以完成它想完成的任何事情，而客户则必须进行相应的准备。这使得 OLE 程序编写起来极为困难。

Microsoft 即将推出一系列基于 COM 的新产品。相信它们将给用户带来一些惊喜。其他的一些开发人员也正在编写 COM 组件。COM 的前途是极为光明的。

1.4 本章小结

软件工业技术的迅速发展使得传统的那种每隔几年就由开发人员将应用程序升级一次的作法越来越不能适应软件发展的需要。解决这一问题的方法就是将应用程序分割成

一些小的应用或组件。然后将这些组件在运行时组装起来以形成所需的应用程序。每一个组件都可以在不影响其他组件的情况下被升级。这使得应用程序可以随时向前发展进化。

COM 提供了编写组件的一个标准方法。遵循 COM 标准的组件可以被组合起来以形成应用程序。至于这些组件是谁编写的、如何实现的都是无关紧要的。每一个 COM 组件均可同其他组件一起使用。实现这种可动态改变组件的关键问题是信息的封装。对于封装，COM 是通过组件和客户之间的连接或接口来实现的。在下一章中读者将看到为什么接口对于组件是如此重要以及如何实现 COM 接口。

759776

第2章 接 口

最近美国的航天飞机 Atlantis 成功地实现了同俄罗斯的空间站 Mir 的对接。美国人在他们的航天飞机上安装了一个俄罗斯的对接部件，以便能够同 Mir 上相应的部件对接起来。读者可以想象一下对这个俄罗斯的对接部件做些什么。例如可以将其安装在自己的房子上，这样航天飞机也将能够对接上去。或者也可以将此部件安装在一个 VW Beetle、天线或车房上，这样航天飞机也将能够对接上去。只要对接部件不改变，并且航天飞机上安装有相应的部件，那么就可以实现对接。有了这两个部分，就可以将 VW 对接到房子上，而不必将其放到车库中。可以想象一下这是一件多么有趣的事情！

上面我们所提到过的对接部件就是接口的一个例子。接口提供了两个不同对象间的一种连接。上面的对接部件对于航天器的对接虽然非常有用，但对于将计算机程序中的若干部分连接起来，它们是无能为力的。实际上计算机程序是通过一组函数而连接起来的。这组函数实际上就定义了程序中不同部分的接口。

DLL 的接口就是它所输出的那些函数；C++ 类的接口则是该类的一个成员函数集。但本书中所提到的接口则指的是一个 COM 接口，而非 DLL 或 C++ 接口。COM 中的接口也涉及到一组由组件实现并提供给客户使用的函数。对于 COM 来说，接口是一个包含一个函数指针数组的内存结构。每一个数组元素包含的是一个由组件所实现的函数的地址。对于 COM 而言，接口就是此内存结构，其他东西均是一个 COM 并不关心的实现细节。我们将在讨论如何用 C++ 实现一个接口后，在本章的末尾，详细地讨论此内存结构。

在 C++ 中，可以使用抽象基类来实现 COM 接口。由于一个 COM 组件可以支持任意数目的接口，因此对于此类组件，我们将用抽象基类的多重继承来实现之。在实现本章的第一个例子时，我们将讨论此方法。

下面先更详细地讨论接口在 COM 中的作用，然后再实现一个接口。在此实现之后，我们将讨论一下若干关于接口的令人感兴趣的问题。最后我们将介绍一下 COM 所需的接口的内存结构。

2.1 接口的作用

在 COM 中接口就是一切。对于客户来说，一个组件就是一个接口集。客户只能通过接口才能同 COM 组件打交道。下面我们将会看到，从整体上讲，客户对于一个组件可以说是知之甚少的。在某些情况下，客户甚至不必知道一个组件所提供的所有接口。

第 1 章中的插图表示的是由组件构成的应用程序。这些图同图 2-1 左侧的内容相似。但第 1 章中的那些图强调的是组件的重要性，这一点会形成误导。实际上从一个程序员的角度来讲，接口对于一个应用程序而言是最重要的。组件本身只不过是接口的实现细节。图 2-1 右侧更为精确地显示出了接口对于 COM 程序的重要性。

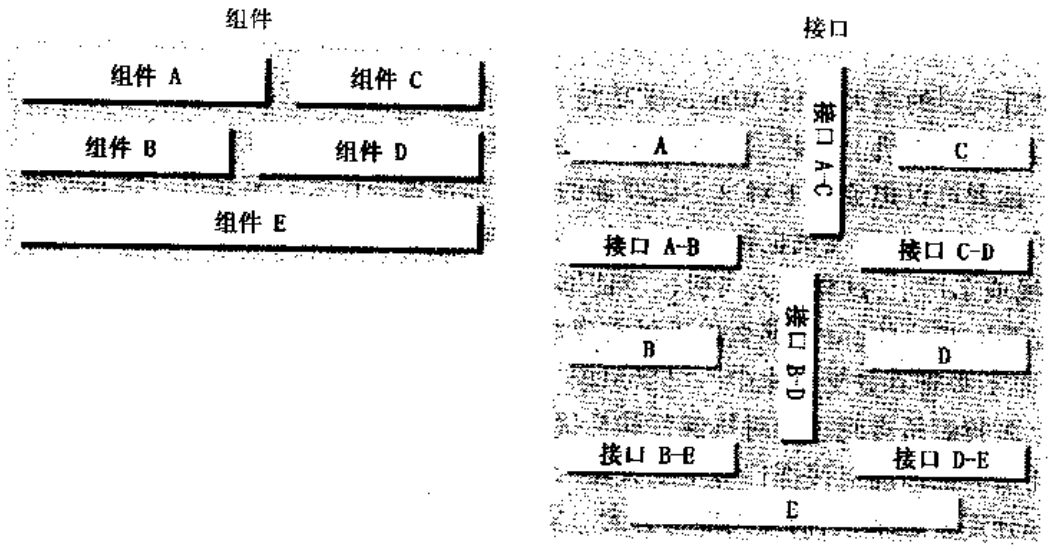


图 2-1 在 COM 中接口比实现接口的组件更为重要

2.1.1 可复用应用程序架构

说组件仅仅只是接口的实现细节当然有点言过其实。不管怎么说，一个未被实现的接口实际上什么也不能完成。但是组件可从应用程序中删除并可用另外一个组件来取代之。只要新的组件支持同组件相同的接口，那么整个应用程序将仍然能够工作。单个的组件并不能对整个应用程序产生决定性的作用。相反，用以连接组件的接口将对整个应用程序产生决定性的作用。只要接口保持不变，那么组件可以任意地更换。

接口同木板房中的大梁非常类似。这些大梁决定了整个房屋的结构。没有它们，房顶和四壁将无法抵抗住风吹雨打的侵袭。只要大梁不被破坏，那么整个房屋的结构将保持不变。就如将原来是由砖砌成的墙壁改为木制，虽然会导致房屋外观的改变，但并不影响其整体结构一样。同样，可以将应用程序所用的组件替换掉，这样应用程序的行为将会发生变化，但从结构上讲，整个应用程序并没有发生任何变化。使用组件来构造应用程序的最大的优点在于可以复用应用程序的结构。如果接口设计得好的话，将可以得到可复用极高的结构。例如，可以通过允许改变若干关键性的组件，同样的结构可以支持几种不同类型的应用。

但设计可复用的结构绝不是一件容易的事情。例如，它要求设计者应具有预测未来的能力。这一点对于大多数人来说可能是太难了。当然使用接口除了可以设计出可复用

的结构外，还有其他若干优点。

2.1.2 COM 接口的其他优点

本章开头我们讲到使用接口的两个原因。其一，它们可以将航天飞机对接到 VW Beetle 上。当然这只是一句玩笑而已。这样讲只是为了说明接口可以保护系统免受外界变化的影响。例如，只要保持对接部件不变，那么航天飞机和空间站可以任意变化。其二，接口使得客户可以用同样的方式来处理不同的组件。这种能力就被称作是多态。例如，航天飞机可以将任何一个安装有对接部分的物体当成是空间站。后面我们将详细讨论多态的概念。

这些优点是通过接口将某个特定的行为封装起来而获得的。例如对接部件封装的是对接这一行为。它隐藏了航天飞机所对接的物体的细节。由于航天飞机只是同另外一个对接部件对接，因此完全可以将空间站换成为一个非常大的铁模而不会对航天飞机产生任何影响，至少是不会对对接产生什么影响。

到此为止，相信读者对于关于接口的理论多多少少是有一些了解了。接下来应该将学到的这些理论应用到接口的实现中去。在本章的最后我们将再介绍其他的一些理论。

2.2 COM 接口的实现

下面将给出实现一些简单接口的代码。其中组件 CA 将使用 IX 和 IY 来实现两个接口。

```
class IX                    // First interface
{
public:
   virtual void Fx1() = 0 ;
   virtual void Fx2() = 0 ;
} ;
class IY                    // Second interface
{
public:
   virtual void Fy1() = 0 ;
   virtual void Fy2() = 0 ;
} ;
class CA :    public IX,    // Component
              public IY
{
public:
   // Implementation of abstract base class IX
   virtual void Fx1() {cout << "Fx1" << endl ;}
```

```
    virtual void Fx2() {cout << "Fx2" << endl ;}
    // Implementation of abstract base class IY
    virtual void Fy1() {cout << "Fy1" << endl ;}
    virtual void Fy2() {cout << "Fy2" << endl ;}
} ;
```

在上述实现中，*IX* 和 *IY* 是用于实现接口的*纯抽象基类*。纯抽象基类指的是仅包含纯抽象函数的基类。纯虚函数指用 =0 标记的虚拟函数。在定义纯抽象函数的类中，是不实现这些纯虚函数的。例如在上面的代码中，函数 *IX*::*Fx*1、*IX*::*Fx*2、*IY*::*Fy*1、*IY*::*Fy*2 并没有函数体。纯虚函数将在派生类中实现。在上面的代码段中，组件 CA 继承了两个纯抽象基类 IX 和 IY，并实现了它们所定义的纯虚函数。

为实现 *IX* 和 *IY* 的成员函数，*CA* 使用了多重继承。多重继承指的是一个类直接继承了一个以上的基类。在 C++ 中，大多数情况下人们使用的是单继承，也就是说一个类只有一个基类。在本章的后面，我们将详细讨论多接口及多重继承。

可以将一个抽象类看作是一个表单，派生类所做的就是填充此表单中的空白。抽象基类指定了其派生类应提供哪些函数，而派生类则具体实现这些函数。对纯虚函数的继承被称作是*接口继承*，这主要是因为派生类所继承的只是基类对函数的描述。抽象类并没有提供任何可供继承的实现细节。

在本书中我们将使用纯虚函数来实现所有的接口。由于 COM 是与语言无关的，对于什么是接口，它有一个二进制的标准。也就是说，表示一个接口的内存块必须具有一定的结构。在本章后面的"接口内幕"一节中将详细介绍这一结构。幸运的是，当使用纯抽象基类时，许多 C++ 编译器将可以生成具有这种结构的内存块。

实际上，上面的 *IX* 和 *IY* 并不是真正意义上的 COM 接口。为使之成为 COM 接口，它们必须继承一个名为 IUnknown 的接口。下一章将详细讨论 IUnknown 接口，故在此不对之作更详细的介绍。在本章的后面，我们将还是把 IX 和 IY 看作是 COM 接口。

2.2.1 编码约定

为将接口同其他类的定义区分开来，作者使用了几种不同的编码约定。例如，在所有接口名称的前面，都加有一个字母"I"，这样 IX 表示的是"接口 X"。而在类名称的前面所加的前缀则为"C"，如 CA 表示的是"类 A"。

另外一个约定是在本书中我们并不将一个接口定义成一个类，而是使用了 Microsoft Win32 软件开发工具(SDK)中 OBJBASE.H 头文件中的定义：

```
#define interface struct
```

在上述定义中使用 struct 的原因在于 struct 的成员将自动具有公有的属性，因此不需要另外在定义中加上 *public* 关键字。去掉 public 关键字可以减少一些混乱。根据这个约定，上例中的接口可以重新定义如下：

```
#include <objbase.h>  //Use for #define interface struct
interface  IX
{
  virtual  void_stdcall  Fx1()=0;
  virtual  void_stdcall  Fx2()=0;
};
interface  IY
{
  virtual  void_stdcall  Fy1()=0;
  virtual  void_stdcall  Fy2()=0;
};
```

在本书中,对接口的图形化表示是一个带有插座的浮动框。如图 2-2 所示即为一例。

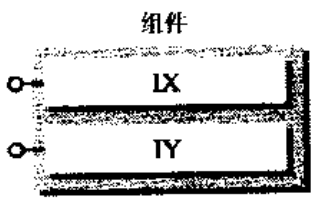


图 2-2 具有两个接口的组件的图形化表示

上述几个约定将作为本书用 C++ 来实现和表示 COM 接口的基础。

2.2.2 一个完整的例子

下面我们来看一下接口 IX 和 IY 的一个简单然而却是完整的实现的例子。在此将使用简单的 C++ 程序来实现组件而没有用到动态链接,这主要是为了简单起见。在第 5 章中我们将讨论如何使用动态链接。在程序清单 2-1 中,类 CA 实现了一个支持接口 IX 和 IY 的组件,而其客户则是函数 main。

在使用 Microsoft Visual C++ 来编译此文件时,可输入如下命令:

```
cl Iface.cpp
IFACE.CPP
```

程序清单 2-1 接口实现的完整例子。

```
//
// Iface.cpp
// To compile, use: cl Iface.cpp
//
#include <iostream.h>
#include <objbase.h>     // Define interface.
void trace(const char * pMsg) {cout << pMsg << endl ;}
// Abstract interfaces
interface IX
```

```
{
    virtual void __stdcall Fx1() = 0;
    virtual void __stdcall Fx2() = 0;
};
interface IY
{
    virtual void __stdcall Fy1() = 0;
    virtual void __stdcall Fy2() = 0;
};
// Interface implementation
class CA : public IX,
           public IY
{
public:
    // Implement interface IX.
    virtual void __stdcall Fx1() {cout << "CA::Fx1" << endl;}
    virtual void __stdcall Fx2() {cout << "CA::Fx2" << endl;}
    // Implement interface IY.
    virtual void __stdcall Fy1() {cout << "CA::Fy1" << endl;}
    virtual void __stdcall Fy2() {cout << "CA::Fy2" << endl;}
};
// Client
int main()
{
    trace("Client: Create an instance of the component.");
    CA* pA = new CA;
    // Get an IX pointer.
    IX* pIX = pA;
    trace("Client: Use the IX interface.");
    pIX->Fx1();
    pIX->Fx2();
    // Get an IY pointer.
    IY* pIY = pA;
    trace("Client: Use the IY interface.");
    pIY->Fy1();
    pIY->Fy2();
    trace("Client: Delete the component.");
    delete pA;
    return 0;
}
```

上述程序的输出为：

```
Client: Create an instance of the component.
Client: Use the IX interface.
CA::Fx1
CA::Fx2
Client: Use the IY interface.
CA::Fy1
CA::Fy2
Client: Delete the component.
```

在上例中读者可以看到客户和组件是通过两个接口来通信的。在接口的实现中使用了两个纯抽象基类 IX 和 IY。例中的组件是由类 CA 实现的，它继承了 IX 和 IY，并实现了这两个接口中的所有成员。

客户 main 函数创建了组件的一个实例，并获取了组件所能提供的、指向接口的指针。然后它像使用 C++ 指针那样使用了这两个指针，这是由于接口是用纯抽象基类实现的。总结一下可知上例的关键之处在于：

- COM 接口在 C++ 中是用纯抽象基类实现的。
- 一个 COM 组件可以提供多个接口。
- 一个 C++ 类可以使用多继承来实现一个可以提供多个接口的组件。

在上例中还有一些关于停息封装的问题，关于此我们将在下面的几章中详细讨论。下面仅就在程序清单 2-1 中所涉及的问题作一简要讨论。

_stdcall 或 Pascal 调用约定

读者可能会注意到上面的_stdcall 这个词，它实际上是 Microsoft 对编译器的一个扩展。任何一个支持开发 Win32 应用程序的编译器都会有此或与此等价的选项。例如 Borland、Symantec 以及 Watcom[①] 的编译器均有些选项。用_stdcall 标记的函数将使用标准的调用约定，即这些函数将在返回到调用者之间将参数从栈中删除。Pascal 函数对于栈的处理使用的也是同一种方式。在常规的 C/C++ 调用约定中，栈的清理工作则是由调用者完成的。大多数其他的编程语言，如 Visual Basic 缺省情况下使用的也是标准的调用约定。标准调用约定名称的由来在于所有的 Win32 API 函数，除了那些带有变参的外，使用的都是这种调用方式。带有变参的函数所用的仍然是 C 调用约定，即_cdecl。Windows 采用标准的调用约定的原因在于这种约定可以减少代码的大小，另外还有一个原因是早期的 Windows 是运行于 640KB 的系统上的。

Microsoft 平台上 COM 接口所提供的所有函数使用的均是标准的调用约定。参数数目可变的函数使用的则是 C 调用约定。一般人们希望接口的实现使用这些约定。但要说明的是这并不是 COM 的绝对需要。开发人员可以选择使用另外某些调用约定，但一定

① 当在非 Microsoft 的操作系统环境下实现 COM 组件时，若不想让它能够同 Microsoft COM 组件之间进行互操作，那么无需关心调用约定的问题。

要在文档中将这一点写清楚,并且还要注意用其他语言所编写的组件可能并不能使用此接口。

若读者希望用另外某个更易于记忆的词来代替_stdcall,可以使用 pascal。在 WINDEF.H 中,pascal 的定义如下:

```
#define pascal _stdcall
```

如果读者认为将 pascal 这个词放在代码中会让人莫名其妙,那么可以使用 ORIBASE.H 中所定义的如下宏:

```
#define STDMETHODCALLTYPE _stdcall
```

2.2.3 非接口通信

不知读者是否还记得前面我们提到过:客户和组件之间只是通过接口进行通信。但程序清单 2-1 中的客户却没有遵循这一规则。在那里,客户与组件间的通信是通过一个指向类 CA 的指针而不是通过指向接口的指针完成的。使用指向 CA 的指针要求客户知道类 CA 的定义(通常是一个头文件)。在类 CA 的声明中有许多有关实现的细节。对这些实现细节的修改将使得客户必须被重新编译。但前面已经讲过,增加或减少组件的接口不应打断已有的客户。这就是我们为什么一再坚持客户和组件之间只应通过接口进行通信。记住,接口是由没有实现细节的虚纯基类实现的。

当然当客户和组件在同一源文件中时,并没有必要将它们分开。但是当客户和组件是在动态链接的情况下,此种隔离则是必需的。在没有源代码的情况下更是如此。在第 3 章中我们将对前面的例子进行修改,使之不再使用指向 CA 的指针。这样一来,客户也将不再需要 CA 的类声明。

使用指向 CA 的指针并不是上面例子违背只用接口进行通信的唯一表现。在客户程序中使用了 new 和 delete,这些操作可以控制组件的生命期。new 和 delete 不但不是任何接口的一部分,而且只是 C++ 才包含这些操作。第 4 章将介绍一种使用接口而不是与某种语言特定的操作符来删除组件的方法。在第 6 章及第 7 章中我们将给出一种更有效的建立组件的方法。

下面我们对程序清单 2-1 给出的例子中客户及组件的实现细节作更为详细的讨论。

2.2.4 实现细节

程序清单 2-1 给出的是一个标准的 C++ 程序,它并没有什么特别异常之处,只不过碰巧是我们建立 COM 组件和客户的第一步。很多人容易将 COM 对组件的需求以及组件是如何实现的这两个问题弄混淆。在这一节中,我们将澄清这一问题。

一、类并非组件

在程序清单 2-1 中,类 CA 实现了单个的组件。但 COM 并没有要求一个 C++ 类只能同一个 COM 组件相应。实际上一个组件可以是由多个 C++ 类实现的,并且不用任何 C++ 类也可以实现一个组件,例如,当用 C 来实现组件时就不会用到类。所以在用C++ 开发组件时也不一定非用类不可。只不过用类来实现组件将比其他方法更为容易。

二、接口并非总是继承的

CA 继承了它所支持的各接口。但 COM 并没有要求实现某个接口的类必须从那个接口继承,这是因为客户并不需要了解 COM 组件的继承关系。对接口的继承只不过是一种实现细节而已。除了可以使用一个类来实现几个不同的接口外,还可以用单个的类来实现每一个接口,再使用指向这些类的指针。在 Kraig Brockschmidt 所著的《OLE 技术内幕》一书中使用的就是这种方法。一般使用一个类来实现所有的接口,这样比较简单,易于理解,并且可使得用 C++ 进行 COM 编程更为自然。

三、多重接口及多重继承

组件可以支持任意数目的接口。为支持多重接口,可以使用多重继承。在程序清单 2-1 中,CA 使用多重继承继承了它所支持的两个接口 IX 和 IY。这种支持多重接口的组件可以被看作是接口的集合。

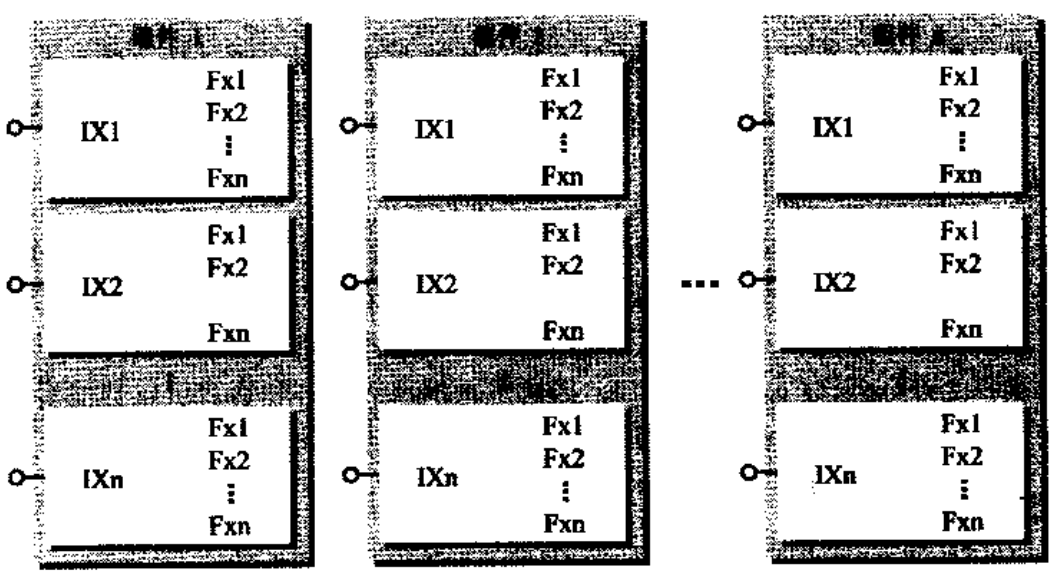


图 2-3 一个系统是一系列组件的集合;每个组件提供了一个接口集;而每一个接口则包含一系列函数

这种特点使得我们可以按一种递归的方式对组件架构进行层次的划分(参见图2-3)。在这种层次结构中，一个接口是一个函数集合，一个组件则是一个接口集，而一个系统则是一系列组件的集合。有些人将接口同特性等同起来。当向组件中增加一个接口时，此时他们会说组件现在可以支持此种特性了。但作者认为将接口看作是行为可能更为合适。一个组件所支持的接口同此组件的各种行为是相对应的。

四、命名冲突

对于一个支持多个接口的组件，接口函数的名字出现冲突是经常会遇到的。此种情况下，改变某个发生冲突的函数名称即可，COM对此并不关心。这是因为COM接口是一个二进制标准；客户同接口的连接并不是通过其名称或其成员函数的名称完成的，而是通过它在表示它的内存块中的位置完成的。在本章的末尾我们将讨论这一内存结构。在第8章中将使用这种改变接口及成员函数名称的技术。

解决命名冲突的另外一种方法是不使用多重继承。实现组件的类并不需要继承每一个接口，而可以合用指向实现某些接口的类的指针。

接口名称之间出现冲突的情况也是可能的。但如果所有的程序员都使用某种简单的约定则可以减少此种可能性。例如，可以在接口和函数名称的前面加上公司名称或产品名称。如对产品Xyz中的接口IFly，可以使用IXyzFly作为其名称。

2.3 接口理论：第二部分

在开始实现接口之前，作者曾提到过要进一步拓宽读者对于接口的了解。在这一节中我们将完成这一任务。所讨论的问题包括：COM接口的不变性、多态以及接口继承。

2.3.1 接口的不变性

从程序清单2-1中可以明显地看出，接口不会发生任何变化。实际上这是COM接口最令人震撼之处。一旦公布了一个接口，那么它将永远保持不变。当对组件进行升级时，一般不会修改已有的接口，而是加入一些新的接口。这些多重接口使得组件除了可以支持原有接口之外，还可以支持新的接口。因此多重继承为组件和客户可以智能地同对方的新版本进行交互提供了坚实的基础。在下一章中我们将讨论版本的问题。

2.3.2 多态

多态指的是可以按同一种方式来处理不同的对象。多重接口的支持能力给多态提供了更多的机会。若两个不同的组件支持同一接口，那么客户将可以使用相同的代码来处理其中的任何一个组件。也就是说，客户可以按照相同的方式来处理不同的组件。

多重接口使得多态的重要性更为突出。一个组件所支持的接口越多，这些组件就应

该越小。较小的接口表示较为简单的行为，而大的接口则表示更多的行为。一个接口所表示的行为越多，它的特定性将越强，因此它被其他组件复用的可能性将越小。对于不能复用的接口，使用此接口的客户代码也将不能复用。

例如可以判断一下，是用一个接口来表示一架直升机的所有行为，如飞行、浮动、盘旋、上升、滚动、颤动、摇摆、降落等的可复用性更高呢，还是用多个接口分别来表示这些行为的可复用性更高。显然一个表示飞行的接口将具有比表示直升机的接口具有更高的可复用性，因为大多数事物的行为同直升机的行为都是不同的，但却有许多事物能够飞行。

使用多态的一种令人吃惊的结果就是整个应用程序都将是可复用的。假定有一个名为 Viewer 的显示位图的应用程序。此位图是用支持一个名为 IDisplay 的接口的 COM 组件实现的。Viewer 仅通过其 IDisplay 接口同组件进行交互。当需要用 Viewer 来显示 VRML 文件时，此时所需要做的只是实现一个支持 IDisplay 接口的 COM 组件，而无需重新构造 Viewer 的一个新版本。新组件将显示 VRML 文件而不是位图。虽然显示 VRML 文件也需要进行大量的工作，但却无需将整个应用程序重写一遍。

这种复用整个应用架构的能力并不是随便就能出现的。在此需要精心设计接口，以使之能够支持各种不同的实现。这不但要求接口具有较高的通用性，而且客户也应按一种比较通用的方式来使用此接口，以避免对接口的实现造成不必要的限制。对于那些并没有考虑到将来可能会出现的组件的接口及应用程序，它们将不能充分利用多态所具有的各种优点，也将无法实现对整个应用架构的复用。例如在上例中，如果事先没有进行很好的规划，接口 IDisplay 可能并不会具有 VRML 文件及位图所要求的通用性及灵活性。并发组件软件的一个最大的问题是如何设计出具有高复用性、适应性、灵活性的接口，并考虑到将来可能会出现的新情况。

至此我们已经知道了如何实现一个组件以及组件具有哪些功能。下面我们来看一下 COM 接口到底是什么以及为什么可以用纯抽象基类来实现组件。

2.4 接口的背后

本章我们讨论的主要问题是：使用 C++ 的纯抽象基类来实现 COM 接口。本节将讨论为什么用抽象基类来实现 COM 接口是可行的。我们将看到这主要是由于纯抽象基类所定义的内存结构可以满足 COM 对接口的需求。

2.4.1 虚拟函数表

当定义一个纯抽象基类时，所定义的实际上是一个内存块的结构。纯抽象基类所有实现都是一些具有相同的基本结构的内存块。图 2-4 表示的是下列代码所定义的纯抽象基类的内存结构：

```
interface IX
```

```
{
    virtual void _stdcall Fx1() =0;
    virtual void _stdcall Fx2() =0;
    virtual void _stdcall Fx3() =0;
    virtual void _stdcall Fx4() =0;
};
```

定义一个纯抽象基类也就是定义了相应的内存结构。但此内存只是在派生类中实现此抽象基类时才会被分配。当派生类继承一个抽象基类时,它将继承此内存结构。

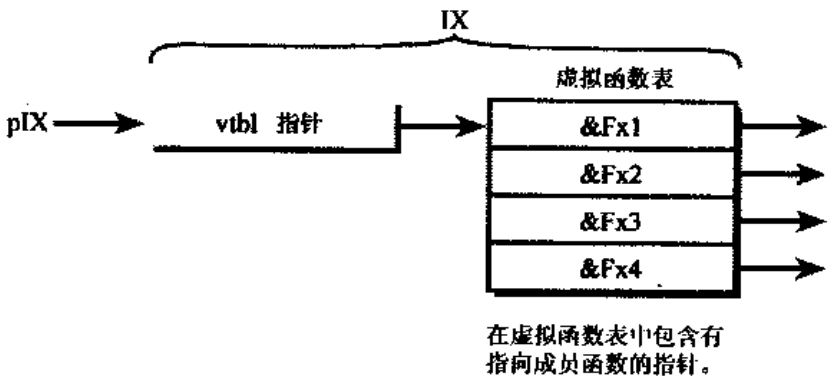


图 2-4 抽象基类所定义的内存块结构示例

从图 2-4 可以看出一个纯抽象基类所定义的内存结构可以分成两部分。图 2-4 的右边是虚拟函数表(vtbl),其中包含一组指向虚拟函数实现的指针。上例中 vtbl 中的第一项为派生类中所实现的 Fx1 函数的地址,第二项则是函数 Fx2 的地址,如此等等。图中左侧为一个指向 vtbl 的指针(称作是 vtbl 指针)。而指向抽象基类的指针则指向此 vtbl 指针。

似乎是一个偶然的巧合,COM 接口的内存结构同 C++ 编译器为抽象基类所生成的内存结构是相同的。因此,可以合用抽象基类来定义 COM 接口。从这个意义上讲,IX 既是一个接口,也是一个抽象基类。说它是一个 COM 接口是由于其内存结构符合 COM 规范的要求,而说它是一个抽象基类则是由于我们就是用此种方式来实现它的。

当然没有任何标准要求 C++ 编译器生成图 2-4 所 示的内存结构。这是由于用 C++ 编写的程序通常都是用同一个编译器。因此编译器只要同自己兼容就够了。但是大多数编译器所生成的内存结构都同图 2-4 所示相同。所有与 Windows 兼容的 C++ 编译器都能生成 COM 可以使用的正确的 vtbl[①]。

对于一个 COM 接口还有其他一些需求。例如,所有的 COM 接口都必须继承一个名为 IUnknown 的接口。关于此接口我们将在下一章讲述。这意味着所有 COM 接口的前三

① 接口的内存结构对于不同的操作系统可能是不同的。例如在 Mac 上,vtbl 指针将指向一个伪指针,而第一个函数指针则位于第二个表项中。

个项都是相同的,其中保存的是 IUnknown 中三个成员函数的实现的地址。关于此我们将在第 3 章中详细讨论。

2.4.2 vtbl 指针及实例数据

那么 vtbl 指针的用途何在呢?

vtbl 指针在由抽象基类函数指针到函数的过程中增加了一个额外的级别。正是这一额外的级别给接口的实现带来极大的灵活性。

C++ 编译器生成代码时,实现抽象基类的类可能会将特定于实例的信息同 vtbl 一块保存。例如,下面代码中的类 CA 实现了前面代码中的抽象基类 IX。

```
class CA: public IX
{
public:
    //Implement interface IX.
    virtual void _stdcall Fx1() {cout << "CA::Fx1" << endl ;}
    virtual void _stdcall Fx2() {cout << m_Fx2 << endl ;}
    virtual void _stdcall Fx3() {cout << m_Fx3 << endl ;}
    virtual void _stdcall Fx4() {cout << m_Fx4 << endl ;}
    //Constructor
    CA(double d)
    : m_Fx2(d*d), m_Fx3(d*d*d), m_Fx4(d*d*d*d)
    {}
    // Instance Data
    double m_Fx2 ;
    double m_Fx3 ;
    double m_Fx4 ;
} ;
```

对于上面我们所讲的那种编译器,vtbl 和 CA 的类数据将如下面的图 2-5 所示。注意,实例数据理应是可以通过指向类的指针 pA 来访问的,但由于客户通常并不知道实例数据是按此种方式保存的,因此客户也就无法访问它了。

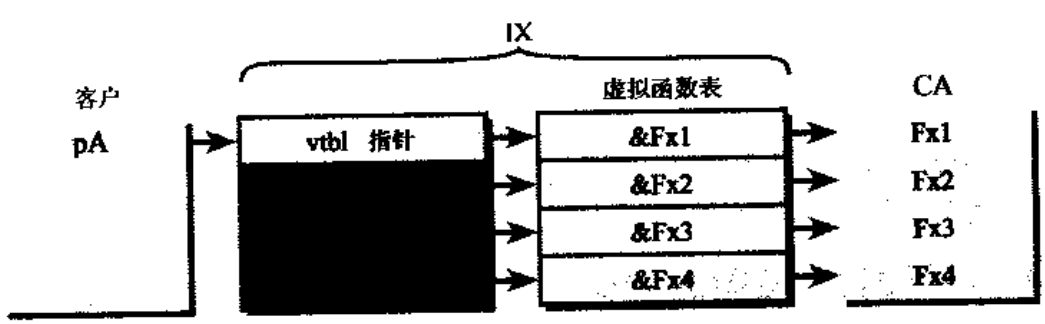


图 2-5 实例数据将同 vtbl 一块保存

虽然C++可以直接操作和使用实例数据，但COM组件绝不会访问任何实例数据。在COM中，对一个组件的访问只能通过函数完成，而绝不能直接通过变量。这一点同我们对COM组件的定义是相符的。纯抽象基类只有虚拟函数，而没有任何实例数据。

2.4.3 多重实例

vtbl的作用绝不仅是给实例数据的保存提供一个方便的位置。实际上它还可以同一个类的不同实例共享同一vtbl。若我们建立CA的两个不同实例，那么将会有两组不同的实例数据。但不同的实例可以共享同一vtbl及相同的实现。例如，假定我们建立两个CA对象：

```
int main()
{
    //Create first instance of CA.
    CA* pA1 = new CA(1.5);
    // Create second instance of CA.
    CA* pA2 = new CA(2.75);
    ⋮
}
```

上述对象将共享同一vtbl。其中的元素将指向虚拟成员函数的同一实现。但是各对象将各自有不同的实例数据(参见图2-6)。

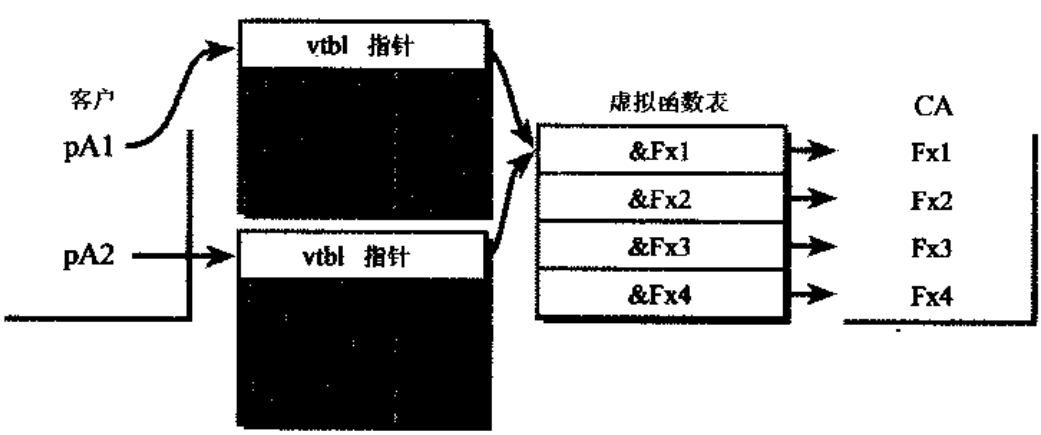


图2-6 同一类的多个实例将共享vtbl

虽然COM组件可以使用vtbl指针来共享vtbl，但这一点并不是必需的。COM组件的每一个实例中已有一个不同的vtbl。

2.4.4 不同的类，相同的vtbl

接口的真正的威力在于继承此接口的所有类均可以被客户按同一方式进行处理。假

定我们实现如下的一个也是继承了 IX 的类 CB:

```
class CB: public IX
{
public:
    //Implement inerface IX.
    virtual void _stdcall Fx1() {cout << "CB::Fx1" << endl ;}
    virtual void _stdcall Fx2() {cout << "CB::Fx2" << endl ;}
    virtual void _stdcall Fx3() {cout << "CB::Fx3" << endl ;}
    virtual void _stdcall Fx4() {cout << "CB::Fx4" << endl ;}
};
```

这样客户可以通过同一个 IX 指针来访问 CA 和 CB。

```
void foo(IX* pIX)
{
    pIX->Fx1() ;
    pIX->Fx2() ;
}
int main()
{
    // Create instance of CA.
    CA* pA = new CA(1.789) ;

    // Create instance of CB.
    CP* pB = new CB ;

    // Get IX pointer to CA.
    IX* pIX = pA ;
    foo(pIX) ;

    // Get IX pointer to CB.
    pIX = pB ;
    foo(pIX) ;
    ⋮
}
```

在上例中,我们将 CA 和 CB 都当成是 IX 接口来使用。这是多态的一个例子。图 2-7 显示了上例的内存结构。图中实例数据是用空框表示的,这是由于 COM 程序员一般都不太关心实例数据究竟是什么。

图 2-7 显示的两个类 CA 和 CB 分别具有不同的实例数据、vtbl 以及实现。但各 vtbl 因其具有相同的格式而可以按相同的方式来访问。函数 Fx1 的地址在两个 vtbl 中都占据了第一个表项,而 Fx2 均占据第二个表项,如此等等。这种格式是编译器根据抽象基类的定义而生成的。在某个类实现抽象基类时,此时它将被强制使用此种格式。对于组件也将是这样。当组件返回一个 IX 接口指针时,它必须保证此指针指向正确的结构。

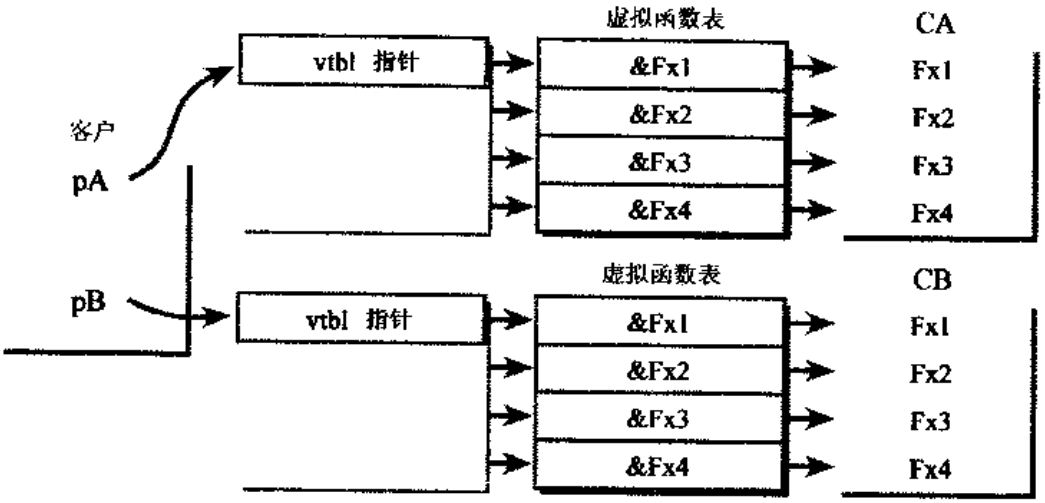


图 2-7 通过一个共同的抽象基类来一致地使用两个不同的类

2.5 本章小结

在本章中我们从接口的一般性概念讲起，一直讲到其内存结构。我们看到接口可以通过封装其内部实现细节而使一个由组件构成的系统免受变化的影响。只要接口不发生变化，那么客户或组件可以在不影响整个系统正常运行的情况下自由地变化。这使得我们可以用新的组件来代替老的组件。客户也可以一致地对待实现同一接口的所有组件。

另外我们介绍了如何在 C++ 中使用纯抽象基类来实现接口。C++ 编译器为纯抽象基类所生成的内存结构同 COM 接口所要求的内存结构是相同的。

这一章介绍了什么是接口、如何实现接口以及如何使用接口。但本章中的接口并不是真正的 COM 接口。COM 需要所有的接口都支持三个函数。这三个函数必须是接口的 vtbl 中的前三个函数。在下一章中我们将讨论这三个函数中的第一个函数 QueryInterface。

第 3 章　QueryInterface 函数

你是一个哺乳类动物吗?

> 是

你长两条腿吗?

> 否

你长着大耳朵吗?

> 是

你个头很大吗?

> 否

你是一只兔子吗?

> 是

对计算机略有所知的人都会想起早期的一个计算机游戏 Animal。这个程序虽小,但却展示了计算机能够学习这一事实:它实际上是一种专家系统的雏形。在玩这个游戏时,用户可以先假想某种动物,然后计算机会提出一些问题,并试图根据用户的回答猜出用户想象的动物到底是什么。

为实现上述目标,Ainmal 在内部维护有一个二叉树。在此树中,终结点表示的是动物的名称,而其他结点则表示各种问题。在程序开始运行时,它将沿着某一条分支,向用户询问结点中的问题,并进而根据用户的回答决定下面沿哪个分支前进。在到达终结点时,它将给出它所猜到的动物名称。若此名称不正确,它将提出一个能够将它所猜出的动物同用户所假想的动物区分开的问题。然后程序将把此问题连同用户所假想的动物加入到树中。

在本章中,我们将会看到 COM 客户在许多方面同 Animal 程序是类似的。例如 Animal 程序并不知道用户所假想的动物是什么,而 COM 客户也并不知道一个组件所支持的接口是什么。为知道某个组件是否支持某个特定的接口,客户可以在运行时询问组件。这一点同 Animal 向用户询问他假想的动物的特征是类似的。又如,Animal 并不理解它所能够猜出的动物的任何特性,而客户对其所用的组件的全部功能也一样是不甚了知。

一年夏天,作者曾给一些一年级的学生教授 Logo 编程。在课堂上,我们用 Logo 语言编写了一个 Animal 程序。在玩了一天之后,我们将它所构造的二叉树打印了出来,从中发现了一些有趣的现象。例如对于 Animal 而言,一只兔子是一种长有四条腿以及大耳朵的哺乳动物,而一头大象则是一种大个的四条腿、大耳朵哺乳动物。不用说,Animal 对于

什么是大象、什么是兔子,所知是极为有限的。

客户对于组件的了解可以说也是非常有限的。客户可以询问组件是否支持某个特定的接口。在进行多次这种询问之后,客户对于组件的认识将越来越清晰。但不管怎样,客户决不可能对组件有一个完整的了解。就像作者班上的孩子们对于兔子长得什么样是了若指掌,而 Animal 却绝不会到达这种程度。

Animal 对于动物所知有限的原因在于其实现方式的负面影响,但客户对于组件的不甚了解却不是一种负面效果。因为我们本来就希望客户对于组件要尽可能地少了解一些。事实上客户对于组件知道得越少,那么在不影响客户正常运行的情况下组件就可能最大限度地发生变化。若客户只是在它需要使用某种接口时才询问组件,那么只是在改变这些接口时才会影响到客户。在不改变与客户接口的情况下,可以用新的组件来代替已有的组件而不会对客户造成任何影响。对于客户不使用的接口,新组件用不着一定要支持。

后面将会看到,即使组件不支持客户所需要的某个接口,客户也可以在请求失败时很好地处理这种情形。在本章中我们将讨论客户如何向组件询问关于它所支持的接口、组件如何回答、以及这种请求应答方式的结果。另外我们还将看到这种使客户请求接口的方式如何提供了一个可以无缝地处理组件版本变化的强壮系统。

3.1 接口查询

由于 COM 中的所有内容最终都起于接口、又最终归于接口,因此首先我们讨论一下可以通过一种什么样的接口来查询其他接口。

客户同组件的交互都是通过一个接口完成的。在客户查询组件的其他接口时,也是通过接口完成的。这个接口就是 IUnknown。IUnknown 接口的定义包含在 Win32 SDK 中的 UNKNWN.H 头文件中。为清楚起见,在此引用如下:

```
interface IUnknown
{
    virtual HRESULT _stdcall QueryInterface(const IID& iid,void** ppv) = 0;
    virtual ULONG _stdcall AddRef() = 0;
    virtual ULONG _stdcall Release() = 0;
};
```

在 IUnknown 中定义了一个名为 QueryInterface 的函数。客户可以调用 QueryInterface 来决定组件是否支持某个特定的接口。本章将主要讨论 QueryInterface。下一章将讨论 AddRef 和 Release,这两个函数可以用来控制接口的生命期。

3.1.1 关于 IUnknown

作者总认为 IUnknown 是一个非常有趣的名字。这个接口实际上是所有的组件及客

户都知道的，但它的名字却表示出它是一个“未知的”接口。之所以使用这个名字主要也是为简单起见。所有的 COM 接口都需要继承 IUnknown。因此若某个客户拥有一个 IUnknown 接口的指针，它并不需要知道它所拥有的接口指针到底是什么类型的，而只需知道此接口可以用来查询其他接口就行了。

由于所有的 COM 接口都继承了 IUnknown，每个接口的 vtbl 中的前三个函数都是 QueryInterface、AddRef 和 Release(如图 3-1 所示)。这使得所有的 COM 接口都可以被当成 IUnknown 接口来处理。若某个接口的 vtbl 中的前三个函数不是这三个，那么它将不是一个 COM 接口。由于所有的接口都是从 IUnknown 继承的，因此所有的接口都支持 QueryInterface。因此组件的任何一个接口都可以被客户用来获取它所支持的其他接口。由于所有的接口指针同时也将是 IUnknown 指针，客户并不需要单独维护一个代表组件的指针，它所关心的将仅仅是接口的指针。

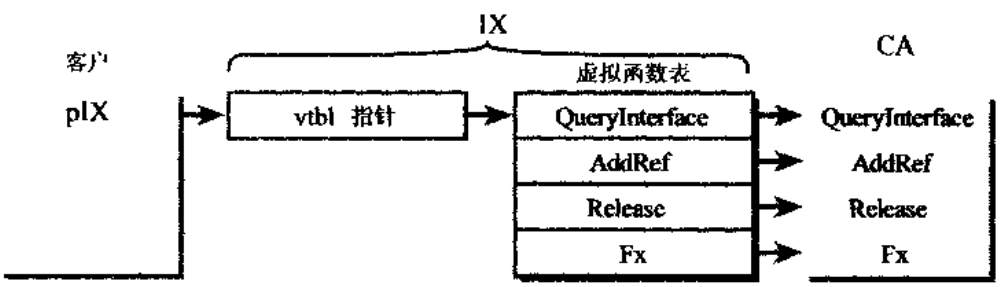


图 3-1 所有的 COM 接口都是从 IUnknown 继承的，因此在其 vtbl 中，前三个项都是 QueryInterface、AddRef 及 Release

3.1.2 IUnknown 指针的获取

那么客户如何获取一个指向 IUnknown 接口的指针呢？这里我们将用到一个名为 CreateInstance 的函数，它可以建立一个组件并返回一个 IUnkown 指针。

```
IUnkown* CreateInstance();
```

在创建组件时，客户可以使用 CreateInstance 而不必再使用 new 操作符。在本章中我们将实现此函数的一个简单版本，然后在后续的各章中将根据需要对之进行修改，第 6 章和第 7 章将介绍建立 COM 组件的正式方法。

在了解了客户如何获取一个 IUnknown 指针之后，下面我们来看一下客户如何使用 QueryInterface 来获取其他接口，然后再实际实现 QueryInterface。

3.1.3 关于 QueryInterface

IUnknown 中包含一个名为 QueryInterface 的成员函数，客户可以通过此函数来查询某个组件是否支持某个特定的接口。若支持，QueryInterface 将返回一个指向此接口的指针；

否则返回值将是一个错误代码。然后客户可以接着查询其他接口或将组件卸载。

QueryInterface带有两个参数：

```
HRESULT_stdcall QueryInterface(const IID& iid, void** ppv);
```

其中的第一个参数也标识客户所需的接口。此参数是一个"接口标识符"(IID)结构。在第6章"关于HRESULT、GUID、登记簿及其他细节"中我们将详细讨论IID。在本章中读者只需将其当成是一个标识所需接口的常量。另外一个指针是QueryInterface存放所请求接口指针的地址。

QueryInterface返回的是一个HRESULT值。此值实际上并不像其名称所表示的那样是标识某个结果的句柄。相反它是一个具有特定结构的32位值。QueryInterface可以返回S_OK或E_NOINTERFACE。客户不应将QueryInterface的返回值直接同这两个值进行比较，而应使用SUCCEEDED宏或FAILED宏。在第6章中我们将详细讨论HRESULT。

下面我们来看一下QueryInterface是如何使用的，然后再来介绍它是如何实现的。

3.1.4 QueryInterface的使用

假定客户有了一个指向IUnknown的指针pI，为知道相应的组件是否支持某个特定的接口，可以调用QueryInterface，并传给它一个接口标识符。若QueryInterface成功返回，那么就可以使用它返回的指针了。这个过程可以用如下的代码表示：

```
void foo(IUnknown * pI)
{
    // Define a pointer for the interface.
    IX* pIX = NULL;

    // Ask for interface IX.
    HRESULT hr = pI->QueryInterface(IID_IX, (void**) &pIX);

    // Check return value.
    if (SUCCEEDED(hr))
    {
        // Use interface.
        pIX->Fx();
    }
}
```

在上面的代码段中，我们查询了pI它是否支持由IID_IX所标识的接口。IID_IX的定义在与此组件相应的头文件中，或它也可能是在某种类型的库中定义的。在第13章中我们将看到这一点。

注意，在调用QueryInterface之前，我们将pIX初始化为NULL，这是一种比较好的编程方法。在后面的实现中我们将会看到QueryInterface在失败时将把返回的接口指针置为

NULL。但由于 QueryInterface 是由程序员而不是由系统实现的，因此某些组件可能并不会在查询失败时将此指针置为 NULL。因此为安全起见，在程序中还是我们自己将其置为 NULL 比较好。

关于如何使用 QueryInterface 的一些基础知识就介绍这么多了。后面我们将介绍一些 QueryInterface 使用的更为高级的技巧。但首先我们来看一下如何在我们的组件中实现 QueryInterface。

3.1.5 QueryInterface 的实现

QueryInterface 的实现实际上是比较简单的。它所需要完成的只不过是根据某个给定的 IID 返回指向相应接口的指针。若组件支持客户指定的接口，那么应返回 S_OK 以及相应的指针。若不支持，返回值应是 E_NOINTERFACE 并将相应的指针返回值置成 NULL。下面我们将介绍如何实现类 CA 所实现的组件中的 QueryInterface 函数。

```
interface IX : IUnknown {/*...*/};
interface IY : IUnknown {/*...*/};
class CA : public IX, public IY {/*...*/};
```

类 CA 及其接口的继承关系如图 3-2 所示。

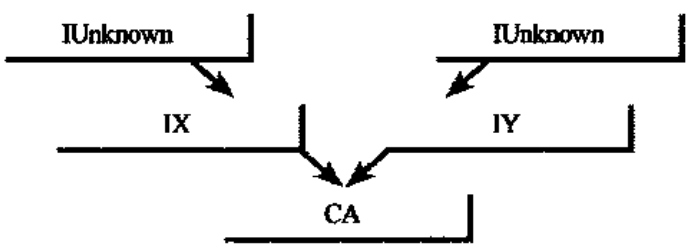


图 3-2 类 CA 及其接口的继承关系

非虚拟继承

注意 IUnknown 并不是虚拟基类。IX 和 IY 并不能按虚拟方式继承 IUnknown，这是由于会导致与 COM 不兼容的 vtbl。若 IX 和 IY 按虚拟方式继承 IUnknown，那么 IX 和 IY 的 vtbl 中的头三个函数指向的将不是 IUnknown 的三个成员函数。

下面的代码所实现的就是前面的类 CA 中的 QueryInterface。此处的 QueryInterface 的实现可以返回三种不同接口的指针：IUnknown、IX 及 IY。不过要注意的是返回的 IUnknown 指针都是同一个，即使 CA 继承了两个 IUnknown 接口（其中的一个来自于 IX，另一个来自于 IY）。

```
HRESULT _stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
```

```
    if (iid == IID_IUnknown)
    {
        // The client wants the IUnknown interface.
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IX)
    {
        // The client wants the IX interface.
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IY)
    {
        // The client wants the IY interface.
        *ppv = static_cast<IY*>(this);
    }
    else
    {
        // We don't support the interface the client
        // wants. Be sure to set the resulting pointer
        // to NULL.
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    static_cast<IUnknown*>(*ppv)->AddRef(); // See Chapter 4.
    return S_OK;
}
```

在上例中，QuertyInterface 是用一个简单的 if-then-else 语句实现的。当然用其他任何一种可以将一种类型映射成另外一种类型的结构也是可以实现的。例如可以使用数组、哈希表或者是树来实现 QueryInterface。在组件所实现的接口很多的情况下，使用这些结构可能会更好一些。但 case 语句是无法用的，因为接口标识符是一个结构而不是一个数。

在客户所查询的接口不被支持时，QueryInterface 将 * ppv 置为 NULL。这一点不但是 COM 规范所要求的，而且从其本身而言也是一种好的作法。将接口指针置为 NULL 将会使不检查返回值的客户崩溃。这一点比起返回一个随机的代码位所造成的损害可能会小得多。另外在 QueryInterface 的末尾调用 AddRef 实际上没有任何作用，在第 4 章中我们将实现 AddRef。

3.1.6 关于类型转换

可以注意到在上面的代码中，在将 this 指针保存在 ppv 中时，我们对之进行了类型的转换。保存在 ppv 中的值将会根据所用的转换不同而不同。例如将 this 指针转换成一个

IX 指针所得到的地址与将其转换成一个 IY 指针所得的地址是不同的。例如：

```
static_cast<IX*>(this) != static_cast<IY*>(this)
ctatic_cast<void*)(this) != static_cast<IY*>(this)
```

换用某些程序员更常用的旧方式的类型转换，即是：

```
(IX*)this != (IY*) this
(void*)this != (IY*) this
```

将 this 指针进行类型转换将会导致其值的改变，这一点主要是由于 C++ 的实现方式。为搞清楚究竟是为什么，可参阅附栏“多生继承及类型转换”。

一般在将 this 指针值赋给某个 void 指针时，应先将其转换成合适的类型。一个有趣的例子是返回 IUnknown 指针的情形。某些程序员可能会用：

```
*ppv = static_cast<IUnknown*>(this); // Ambiguous
```

但是将 this 指针转换成 IUnknown* 是不明确的。这是由于 IX 和 IY 都是从 IUnknown 继承得到的。因此在这种情况下，返回值应该是 static_cast<IUnknown*>(static_cast<IX*>(this))或 static_cast<IUnknown*>(static_cast<IY*>(this))。只不过在上例中选择哪一个是无关紧要的，因为它们使用的都是同一实现。但是在代码中要保持一致，因这两个指针实际上是不一样的，并且 COM 要求对 IUnknown 接口返回相同的指针。本章的后面我们将讨论这种需求。

多重继承及类型转换

通常将一种类型的指针转换成另外一种类型并不会改变它的值。但为了支持多重继承，在某些情况下，C++ 必须改变类指针的值。许多 C++ 程序员并不清楚多重继承的此种负面效果。例如假定有一个类 CA 定义如下：

```
class CA : public IX, public IY {...}
```

由于 CA 同时继承了 IX 和 IY，因此在可以使用 IX 或 IY 指针的地方均可以使用指向 CA 的指针。例如可以将指向 CA 的指针传给接收 IX 或 IY 指针的函数，这样此函数仍将能够正常工作。例如：

```
void foo(IX* pIX);
void bar(IY* pIY);

int main()
{
    CA* pA = new CA;
    foo(pA);
    bar(pA);
```

```
    delete pA;
    return 0;
}
```

foo需要一个指向合法的IX的虚拟函数表的指针，而bar则需要一个指向IY虚拟函数表的指针。当然IX和IY的虚拟函数表中的内容是不一样的。因此在将一个IX vtbl传给bar时，此函数将不能正常工作。因此编译器将同一指针传给foo和bar是不可能的，它必须对CA的指针进行修改以便它指向一个合适的vabl指针。图3-3显示了类CA对象的内存结构：

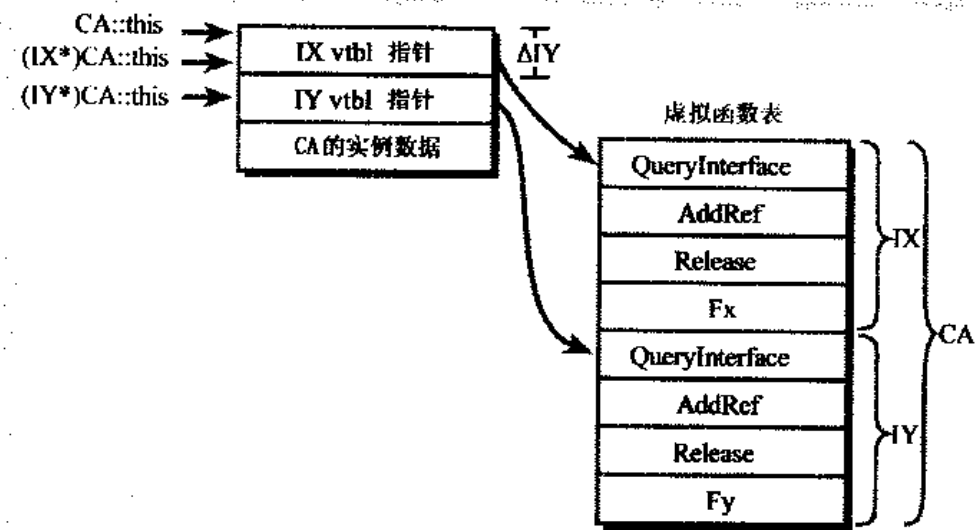


图3-3 同时继承了IX和IY的类CA的内存结构

从图3-3可以看到，CA的this指针指向IX的虚拟函数表。因此可以在不改变CA的this指针的值的情况下用它来代替IX指针。但是从图中可以很明显地看出，类CA的this指针并没有指向IY的虚拟函数表指针。因此在将指向类CA的指针传给一个接收IY指针的函数之前，其值必须修改。为完成此种修改，编译器将把IY虚拟函数表指针的偏移量(ΔIY)加到CA的this指针上，因此编译器将把下面的代码：

```
IY* pC = pA;
```

转换成与下面类似的代码：

```
IY* pC = (char*)pA + ΔIY;
```

有关更为详细的讨论请参阅Margaret A. Ellis和Bjarne Stroustrup所著《The Annotated C++ Reference Manual》中的Section 10.3c，"Multiple Inheritance and Casting"。但应说明的是，C++编译器并不需要实现如图3-3所示的多重继承虚拟函数表。

3.1.7 一个完整的例子

下面我们将把前面所提到过和各代码段组合起来，以构成一个说明 QueryInterface 实现及使用的完整例子。此例子的代码如程序清单 3-1 所列。总的来说可以将这些代码分成三部分。

第一部分是接口 IX、IY 和 IZ 的定义部分。接口 IUnknown 的定义在 Win32 SDK 的头文件 UNKNWN.H 中。

第二部分是组件的实现。类 CA 实现了一个支持 IX 和 IY 接口的组件。QueryInterface 的实现同前一节中给出的实现是一样的。在类 CA 的末尾给出了 CreateInstance 的定义。客户可以使用此函数来创建类 CA 所代表的组件并返回一个指向其 IUnknown 接口的指针。

在定义好 CreateInstance 函数之后，下面定义的是各接口的 IID 结构。从这些定义可以看出 IID 结构是一个相当大的结构。在第 7 章中我们还将再讨论这些结构。在链接此示例程序时，需要将其同 UUID.LIB 一块链接以获取 IUnknown 的接口标识符 IID_IUnknown 的定义。

清单中的第三部分也就是最后一部分是 main 函数，它表示示例程序中的客户。

在使用 Microsoft Visual C++ 编译此程序时可以使用命令 cl IUnknown.cpp UUID.lib。

程序清单 3-1 QueryInterface 的使用。

IUNKNOWN.CPP

```
//
// IUnknown.cpp
// To compile use: cl IUnknown.cpp UUID.lib
//
#include <iostream.h>
#include <objbase.h>

void trace(const char* msg) { cout << msg << endl; }

// Interfaces
interface IX : IUnknown
{
    virtual void __stdcall Fx() = 0;
};
interface IY : IUnknown
{
    virtual void __stdcall Fy() = 0;
};
interface IZ : IUnknown
{
    virtual void __stdcall Fz() = 0;
```

```
};
// Forward references for GUIDs
extern const IID IID_IX;
extern const IID IID_IY;
extern const IID IID_IZ;
//
// Component
//
class CA : public IX,
           public IY
{
    //IUnknown implementation
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
    virtual ULONG __stdcall AddRef() { return 0; }
    virtual ULONG __stdcall Release() { return 0; }
    // Interface IX implementation
    virtual void __stdcall Fx() { cout << "Fx" << endl; }
    // Interface IY implementation
    virtual void __stdcall Fy() { cout << "Fy" << endl; }
};
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
    if (iid == IID_IUnknown)
    {
        trace("QueryInterface: Return pointer to IUnknown.");
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IX)
    {
        trace("QueryInterface: Return pointer to IX.");
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IY)
    {
        trace("QueryInterface: Return pointer to IY.");
        *ppv = static_cast<IY*>(this);
    }
    else
    {
        trace("QueryInterface: Interface not supported.");
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef(); // See Chapter 4.
```

```
    return S_OK;
}

//
// Creation function
//
IUnknown* CreateInstance()
{
    IUnknown* pI = static_cast<IX*>(new CA);
    pI->AddRef();
    return pI;
}

//
// IIDs
//
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IX =
    {0x32bb8320, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};

// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IY =
    {0x32bb8321, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};

// {32bb8322-b41b-11cf-a6bb-0080c7b2d682}
static const IID IID_IZ =
    {0x32bb8322, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};

//
// Client
//
int main()
{
    HRESULT hr;

    trace("Client:      Get an IUnknown pointer.");
    IUnknown* pIUnknown = CreateInstance();

    trace("Client:      Get interface IX.");

    IX* pIX = NULL;
    hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
    if (SUCCEEDED(hr))
    {
        trace("Client:  Succeeded getting IX.");
        pIX->Fx();      // Use interface IX.
    }
```

```
trace("Client:        Get interface IY.") ;
IY* pIY = NULL ;
hr = pIUnknown->QueryInterface(IID_IY, (void**)&pIY) ;
if (SUCCEEDED(hr))
{
      trace("Client:   Succeeded getting IY.") ;
      pIY->Fy() ;      // Use interface IY.
}
trace("Client:        Ask for an unsupported interface.") ;
IZ* pIZ = NULL ;
hr = pIUnknown->QueryInterface(IID_IZ, (void**)&pIZ) ;
if (SUCCEEDED(hr))
{
      trace("Client:   Succeeded in getting interface IZ.") ;
      pIZ->Fz() ;
}
else
{
      trace("Client:   Could not get interface IZ.") ;
}
trace("Client:        Get interface IY from interface IX.") ;
IY* pIYfromIX = NULL ;
hr = pIX->QueryInterface(IID_IY, (void**)&pIYfromIX) ;
if (SUCCEEDED(hr))
{
      trace("Client:   Succeeded getting IY.") ;
      pIYfromIX->Fy() ;
}
trace("Client:        Get interface IUnknown from IY.") ;
IUnknown* pIUnknownFromIY = NULL ;
hr = pIY->QueryInterface(IID_IUnknown, (void**)&pIUnknownFromIY) ;
if (SUCCEEDED(hr))
{
      cout << "Are the IUnknown pointers equal?   " ;
      if (pIUnknownFromIY == pIUnknown)
      {
            cout << "Yes, pIUnknownFromIY == pIUnknown." << endl ;
      }
      else
      {
            cout << "No, pIUnknownFromIY != pIUnknown." << endl ;
      }
}
```

```
    // Delete the component.
    delete pIUnknown ;

    return 0 ;
}
```

此程序的输出如下：

```
Client:          Get an IUnknown pointer.
Client:          Get interface IX.
QueryInterface: Return pointer to IX.
Client:          Succeeded getting IX.
Fx
Client:          Get interface IY.
QueryInterface: Return pointer to IY.
Client:          Succeeded getting IY.
Fy
Client:          Ask for an unsupported interface.
QueryInterface: Interface not supported.
Client:          Could not get interface IZ.
Client:          Get interface IY from interface IX.
QueryInterface: Return pointer to IY.
Client:          Succeeded getting IY.
Fy
Client:          Get interface IUnknown from IY.
QueryInterface: Return pointer to IUnknown.
Are the IUnknown pointers equal?   Yes, pIUnknownFromIY == pIUnknown.
```

客户程序启动之后它将调用 CreateInstance 函数。此函数将返回一个指向组件的 IUnknown 接口的指针。客户将通过此 IUnknown 接口的 QueryInterface 函数而查询一个 IX 接口的指针。在决定查询是否成功时我们使用了 SUCCEEDED 宏。若客户成功地获取了 IX 接口的指针，它将使用返回的指针调用相应接口中的 Fx 函数。

接下来客户将使用 IUnknown 指针来获取一个 IY 接口的指针。若成功，它将使用返回的指针。由于类 CA 同时实现了 IX 和 IY，因此在查询这些接口时肯定会成功。但是类 CA 并没有实现接口 IZ，因此当客户查询 IZ 接口时，QueryInterface 将会失败，此时它将返回 E_NOINTERFACE。此时 SUCCEEED 宏的返回值将为 FALSE，这种情况下客户将不会用 pIZ 来访问 IZ 的成员函数。

这样一来似乎就比较有趣了。客户可以通过一个 IX 接口指针 pIX 来查询 IY 接口的指针。因此组件确实是支持 IY 接口，此查询将会成功，客户将可以象使用第一个指针那样使用返回的 IY 接口指针。

最后客户通过 IY 接口指针来查询 IUnknown 接口。由于所有的 COM 组件都是从 IUnknown 接口继承的，因此，此查询请求也将成功。但最有趣之处在于所返回的 IUnknown 接

口指针 pIUnknownFromIY 同 IUnknown 接口指针 pIUnknown 是相同的。下面我们将会看到这是 COM 的需求之一：QueryInterface 对所有的 IUnknown 接口查询请求都必须返回相同的指针。

这个例子表明，从任何其他的接口都可以使用 QueryInterface 来获取 CA 所实现的任意接口。这是指导 QueryInterface 接口实现的重要规则之一。下面我们将具体讨论一下 QueryInterface 的实现有一些什么样的规则。

3.2 关于 QueryInterface 的实现规则

本节将给出一些 QueryInterface 的所有实现都必须遵循的一些规则，以便客户能够获取关于组件的足够多的知识并对之施实一些控制和其他有用的处理。如果没有这些规则，是不可能编写出组件的，因为在这种情况下，QueryInterface 的行为将是不确定的。具体来讲，这些规则是：

- QueryInterface 返回的总是同一 IUnknown 指针。
- 若客户曾经获取过某个接口，那么它将总能获取此接口。
- 客户可以再次获取已经拥有的接口。
- 客户可以返回到起始接口。
- 若能够从某个接口获取某特定接口，那么可以从任意接口都将可以获取此接口。

下面我们将详细讨论这些规则。

3.2.1 同一 IUnknown

组件的实例只有一个 IUnknown 接口。因为当查询组件实例的 IUnknown 接口时，不论通过哪个接口，所得到的均将是同一指针值。为确定两个接口是否指向同一组件，可以通过这两个接口查询 IUnknown 接口，然后将返回值进行比较。例如，下面代码中的 SameComponents 函数可以决定 pIX 和 pIY 是否指向同一组件中的接口：

```
BOOL  SameComponents(IX* pIX, IY* pIY)
{
      IUnknown* pI1 = NULL;
      IUnknown* pI2 = NULL;

      // Get IUnknown pointer from pIX.
      pIX->QueryInterface(IID_IUnknown, (void**)&pI1);

      // Get IUnknown pointer from pIY.
      pIY->QueryInterface(IID_IUnknown, (void**)&pI2);

      // Are the two IUnknown pointers equal?
      return pI1 == pI2;
}
```

这条规则是极为重要的。如果 QueryInterface 的实现不遵循这条规则的话，则将没法决定两个接口是否指向同一组件。

3.2.2 客户可以获取曾经得到过的接口

若对于某个给定的接口，QueryInterface 曾经成功过，那么对于同一组件的后续 QueryInterface 将总是成功的。若对于某个给定的接口，QueryInterface 调用是失败的，那么后续的调用也将会失败。这一规则适用于组件的某个特定实例。当创建组件的一个新实例时，这条规则并不适用。

为什么需要这条规则呢？可以想象一下如果组件所支持的接口会不时的发生变化将会导致一些什么样的后果。这样一来，客户代码的编写将是极为困难的。例如，客户应该在何时发出查询请求呢？应该按何种频率发出此种请求呢？当客户不再能获取它曾经使用过的某个接口时会发生什么情况呢？因此，若组件实现的接口集不是固定的，客户将无法通过编程的方法来决定一个组件到底具有一些什么样的功能。

3.2.3 可以再次获取已经拥有的接口

若客户拥有一个 IX 接口，则可以通过它来查询 IX 接口指针，并且一定可以成功。例如在下面的代码中：

```
void f(IX* pIX)
{
    IX* pIX2 = NULL;

    // Query IX for IX.
    HRESULT hr = pIX->QueeryInterface(IID_IX, (void**)&pIX2);
    assert(SUCCEEDED(hr)); //Query must succeed.
}
```

这条规则看起来可能会显得有些奇怪。客户为什么需要再次获取它已经拥有的接口呢？记住，所有的接口都继承了 IUnknown，而许多函数都需要一个 IUnknown 指针作为参数。它们应该能够使用任何 IUnknown 指针来获取任何接口。如下面的代码所示：

```
void f(IUnknown* pI)
{
    HRESULT hr;
    IX* pIX = NULL ;

    // Query pI for IX.
    hr = pI->QueryInterface(IID_IX, (void**)&pIX);

    // Do something creative here.
}
```

```
void main()
{
    // Get an IX pointer for somewhere.
    IX* pIX = GetIX();
    // Pass it to a function.
    f(pIX);
}
```

函数 f 将能够使用传给它的接口指针来获取一个 IX 指针，即使此指针已经是一个 IX 指针。

3.2.4 客户可以从任何接口返回到起始接口

若客户拥有一个 IX 接口指针并成功地使用它来查询了一个 IY 接口，那么它将可以使用此 IY 接口来查询一个 IX 接口。不论客户所拥有的接口是什么，它都可以返回起始时所用的接口。如下面的代码所示。

```
void f(IX* pIX)
{
    HRESULT hr;
    IX* pIX2 = NULL;
    IY* pIY = NULL;
    // Get IY from IX.
    hr = pIX->QueryInterface(IID_IY, (void**)&pIY);
    if (SUCCEEDED(hr))
    {
        // Get an IX from IY.
        hr = pIY->QueryInterface(IID_IX, (void**)&pIX2);
        // QueryInterface must succeed.
        assert(SUCCEEDED(hr));
    }
}
```

3.2.5 若能够从某接口获取某特定接口，则从任意接口都将能够获取此接口

若能够从某个组件获取某特定接口，那么客户将可以通过此组件所支持的任意接口获取此接口。例如，若可以通过接口 IX 得到接口 IY，通过 IY 可以得到 IZ，那么通过 IX 也将可以得到 IZ。如下面的代码所示：

```
void f(IX* pIX)
{
    HRESULT hr;
```

```
        IY* pIY = NULL;
        // Query IX for IY.
        hr = pIX->QueryInterface(IID_IY, (void**)&pIY);
        if (SUCCEEDED(hr))
        {
            IZ* pIZ = NULL;
            // Query IY for IZ.
            hr = pIY->QueryInterface(IID_IZ, (void**)&pIZ))
            if (SUCCEEDED(hr))
            {
                // Query IX for IZ.
                hr = pIX->QueryInterface(IID_IZ, (void**)&pIZ);
                // This must succeed.
                assert(SUCCEEDED(hr));
            }
        }
}
```

这条规则使得 QueryInterface 是可用的。可以想象一下若在获取某个特定接口的指针时,必须依赖于所使用的到底是哪个接口会发生什么情况。例如,当修改客户的代码并将其中的某个函数移到另外一个函数之前,那么整个系统可能就无法运行了。这样一来想开发出能够同此组件一块使用的客户几乎是不可能的。

制定上述规则的目的完全是为了使 QueryInterface 使用起来更为简单、更富有逻辑性、更一致以及更具确定性。幸运的是在实现组件的 QueryInterface 时遵循这些规则时并不是什么困难的事情。并且只有当组件按照这些规则正确地实现了 QueryInterface 时,客户才不用为这些规则担心。值得注意的是 QueryInterface 使用和实现上的简单性并不会减少 QueryInterface 对于 COM 的重要性。事实上再没有其他什么像 QueryInterface 对于 COM 那样重要了。

3.3 QueryInterface 定义了组件

我们说 QueryInterface 是 COM 最为重要的部分,这主要是因为一个组件实际上就是由 QueryInterface 定义了。组件所支持的接口集就是 QueryInterface 能够为之返回接口指针的那些接口。这一点是由 QueryInterface 的实现决定的,而不是由实现组件的 C++ 类决定的。实现组件的类的继承层次关系也不能决定组件。一个组件仅仅是由 QueryInterface 的实现决定的。

由于客户并不知道 QueryInterface 的实现,因此,它将无法知道一个组件所支持的所有接口。客户了解组件所支持接口的唯一方法是进行查询。这一点同 C++ 有着很大的不同。在那里,某个类的客户可以知道该类的所有成员,这是因为它拥有此类的头文件。从

某种意义上,COM 更类似于在某次社交聚会上同某人会面,而与对他们进行工作面试有很大的不同。当进行工作面试时,被试者将提交一份介绍他们情况的个人简历。这份个人简历是类似于 C++ 类的定义。而当在社交聚会上会面时,没有人会给对方提供个人简历。为了解对方的情况,必须向他们提问。这一点是类似于 COM 组件的。

3.3.1 接口集

作者开始学习 COM 时所考虑的第一个问题是为什么不能向组件询问它所支持的所有接口呢? 关于这个问题作者得到的回答是:你想对组件所支持的接口列表进行什么处理 呢? 虽然这一回答并没有从正面回答问题,但却不失为一个好的回答。

例如对于前面例子中的组件,它能够支持接口 IX 和 IY。假定客户的编码是在此组件的编码之前完成的,并且此客户并不知道接口 IY。当客户创建此组件并查询其所有接口,然后组件返回 IX 和 IY。由于客户并不能识别 IY,因此,对于此接口不能进行任何处理。为使客户能够对它所不能识别的接口进行一些有意义的处理,客户需要重新阅读关于组件的文档,然后编写处理此接口的代码。从目前的技术水平而言,这一点是不可能的。因此,一个组件所能支持的接口只是那些接口程序员所知道的接口。与此类似,客户所支持的接口只是其程序员所知道的那些接口。

但 COM 也确实提供了一个名为类型库的手段,供客户在运行时确定组件所提供的接口。客户可以使用类型库来决定接口中某个函数的参数,但客户仍然不可能知道如何编写出能够使用这些函数的代码,这项工作仍然需要程序员来完成。关于类型库我们将在第 11 章中讨论。

在许多情况下,客户可以使用只实现某个特定接口集的组件。创建一个组件,然后一个个地查询其组件,以最终找出此组件是否支持某个所需的接口是非常浪费时间的。为节省时间开销,某个特定的接口集可以用一个*组件类别* 来标识。各组件可以声明它是否属于某个特定的组件类别。客户可以在不创建组件的情况下获取此种信息。这些内容将在第 6 章详细讨论。

QueryMultipleInterfaces

分布式 COM(DCOM)定义了一个新接口 IMultiQI。此接口有一个新的成员函数 QueryMultipleInterfaces。使用此函数,客户可以通过一次调用而查询组件的多个接口。这主要是为了减少数据在网络上来回传输的次数,以提高应用程序的效率。

下面我们来看一下 QueryInterface 非常规的用途之一,即处理新版本的组件。

3.4 新版本组件的处理

我们知道在 COM 中接口是不会发生变化的。当组件发布一个接口并被某个客户使

用之后，此接口将决不会发生任何变化，而将永远保持不变。但我们说接口不会发生变化到底是什么含义呢？每一个接口都有一个唯一的接口标识符(IID)。一般情况下，我们不会改变接口，而可以建立一个新接口并为之指定一个新的 IID。当 QueryInterface 接收到对老 IID 的查询时，它将返回老的接口。而当它收到对新的 IID 的查询时，它将返回新的接口。就 QueryInterface 而言，一个 IID 就是一个接口。

所以同某个 IID 相应的接口将绝不会发生变化。新接口可以继承老接口，它也可以同老接口完全不同。由于老的接口仍然保持不变，已有客户的运行将不会受到任何影响。而新客户则可以自行决定是使用老接口还是新接口，因它可以自由决定到底是查询哪个接口。

这种处理多个版本的方法最有效的地方在于它是无缝的。客户不需要做任何附加的工作即可确信它使用的是接口的正确版本。若它能够查找到某个接口，那么这个接口一定是正确的。接口的标识同其版本是完全绑在一块的。若版本发生了变化，那么接口也将发生变化。这里不会发生混淆。

例如假定有一个名为 Pilot 的飞行模拟程序。它使用了由许多不同厂商开发的组件所实现的模拟飞行器。为同 Pilot 一块工作，飞行器组件必须实现一个名为 IFly 的接口。假定某个公司开发了一个名为 Bronco 的飞行器组件，并且它支持 IFly 接口。随后我们决定将 Pilot 升级成一个名为 FastPilot 的新版本。在 FastPilot 中，通过提供另外一个名为 IFlyFast的接口来扩展一架飞机所能表演出的动作。与此相应，销售 Bronco 的公司在其中加入了 IFlayFast 接口，从而得到一个新版本的组件 FastBronco。

由于 FastPilot 仍然支持 IFly，所以如果用户有一份 Bronco，FastPilot 也将能够使用它。此种情况下，FastPilot 将首先查询 IFly，若被查询的组件不支持此接口，那么它将继续查询 IFly。同样，由于 FastBronco 仍然支持 IFly，因此当某些用户仍然在使用原有的 Pilot 时，FastBronco 也将能够同它一块运行。图 3-4 显示出了各种可能的互连关系。

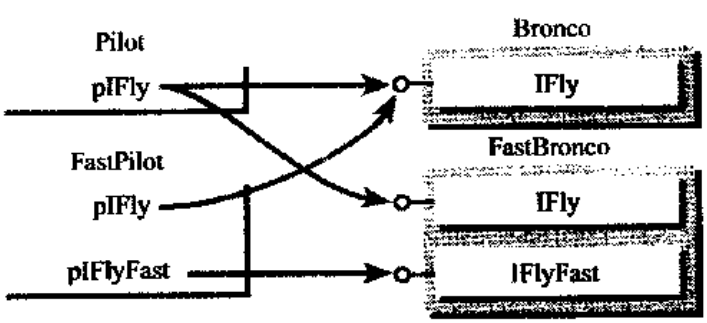


图 3-4 新老版本的客户及组件的各种不同组合

可以看到不论按何种组合方式，客户和组件都将能够正常工作。

在某些情况下，新版本的组件或客户要保持后向的兼容或许是不可能的，诸如需要付出极大的工作量，或具有极高的技术难度。但 COM 处理多个版本的手段在保持后向兼容

不可能的情况下同样是有效的。因为接口的 IID 决定了它的版本。当客户获取某个接口时，由于不同版本的接口实际上是不同的接口，它们各自具有不同的 ID，因此客户仍将能够得取正确版本的接口。

3.4.1 何时需要建立一个新版本

为使 COM 处理多个版本的机制能够起作用，程序员在给某个已有接口的新版本指定新的 ID 时应非常严格。当改变了下列条件中的任何一个时，就应给新接口指定新的 ID：

- 接口中函数的数目。
- 接口中函数的顺序。
- 某个函数的参数。
- 某个函数参数的顺序。
- 某个函数参数的类型。
- 函数可能的返回值。
- 函数返回值的类型。
- 函数参数的含义。
- 接口中函数的含义。

总之，只要是所做的修改将会导致已有客户的正常运行，都应为接口指定新的 ID。（当然若能够同时修改客户和组件，则对于上述规则可以灵活掌握。）

3.4.2 不同版本接口的命名

在建立了某个接口的新版本时，也应相应地修改其名称。COM 关于新版本名称的约定是在老名称的后面加上一个数字。在此种约定下，IFly 的新版本名称将是 IFly2 而不是 IFlyFast。当然对于用户自己建立的接口，可以按自己的喜好指定名称。若老接口是其他人建立的，那么在建立其新版本或指定一个新名称之前应先询问一下别人是否允许。

3.4.3 隐含合约

仅保证函数名称及参数不变并不足以保证对组件的修改不会妨碍客户的正常运行。因为客户将按照一定的方式或次序来使用接口中的函数。若组件的实现发生变化之后，此种调用方式或次序不再能正常工作，则客户也将因此而不能正常运行。

关于这个问题我们可以从另外一个角度来考虑。法律合约一般被认为是非常精确的文档，它精确地说明了所涉及到的合约各方的责任。但不论合同怎么短小或平常，其中总会有一些破绽。这些破绽就是那些当事人在签约时没有仔细考虑，而后又因此损失成千上万财富的句子。在这里合约所使用的字体大小是无关紧要的——合同的法律约束力并不会因此减少什么。

接口就是客户同组件之间的一个合约。同所有合约一样，接口也会有破绽。只不过对于接口而言，容易出破绽的地方在于接口被使用的方式。客户使用接口中函数的方式

定义了它同实现此接口的组件之间的合约。若组件对接口的实现被修改了，它应保证客户仍然能够按同一方式来使用此接口中的函数。否则客户将因无法正常工作而不得不重新编译。例如，若客户按如下次序调用函数 Foo1、Foo2 及 Foo3。若组件被修改之后，Foo3 必须首先被调用，那么此组件实际上就违反了指定接口中函数可以被如何以及按何种次序被使用的隐含合约。

所有的接口都有一些隐含合约。这种合约只是在想用一种将会妨碍对接口已有使用的方法来实现它时才会引起问题。为避免违反隐含合约，可以使用两种方法。第一种方法是使得接口不论在其成员函数怎么被调用都能正常工作。第二种方法是强制客户按一定的方式来使用此接口并在文档中将这一点说明清楚。这样一来当因组件的变化而使客户无法正常运行时，所违反的将不再是隐含规则而是那些明确的规则。不论使用哪种方法，都需要开发人员具有高度的洞察力与仔细的规划。

3.5 本章小结

现在读者知道了 COM 是什么了。QueryInterface 是将编写 COM 组件的过程同编写 C++ 类的过程区分开的一种特性。COM 组件大部分的灵活性及其封装的能力都是由 QueryInterface 提供的。它使得客户能够在运行时决定一个组件所能提供的功能，并最大限度地利用动态链接。通过将组件的功能完全地向客户隐藏起来，QueryInterface 可以尽可能地防止组件实现的变化对客户造成的影响。同时 QueryInterface 也是一种极好的实现对组件版本无缝处理的机制。这种机制使得组件的新旧不同版本可以互操作，从而能够一块工作。

在这一章中我们也讨论了 IUnknown 这一所有其他接口都支持的根接口。而 QueryInterface 只不过是 IUnknown 三个成员函数之一。在下一章中我们将讨论如何使用 IUnknown 的另外两个成员函数 AddRef 和 Release 来代替在我们的例子中一直在使用的 delete 操作符。但在开始阅读下一章的内容之前，可以再看一下游戏 Animal。

你需要完成将组件从内存中释放之类的处理吗?
>是
你需要处理引用计数吗?
>是
你是 AddRef 吗?
>否
你是谁?
>Release
你同 AddRef 有什么区别呢?
>减少引用计数值
谢谢。

第4章 引用计数

作者小时候经常梦想能成为一名消防队员。同大多数小孩一样，作者也同样为那种令人着迷、兴奋以及危险的情形所吸引。但这些同真正想成为一名消防队员的原因比较起来，简直可以说是不值一提。首先，消防队员可以威风凛凛地站在消防车的上面。这一点似乎也是作者那时想成为一名清洁工的原因，当然这是另外一回事了。其次消防队员可以穿上非常酷的消防服装：如漂亮的头盔、长筒靴、非常大的套装以及氧气罐等。作者那时是多么希望有朝一日能够将这些东西穿在身上啊！不论他们走到哪里，似乎总是穿着这一身行头，即使是在将一只可怜的猫从树上救下来也是如此。显然人们一眼就能看出消防队员同其他人的区别。

C++类在许多方面同消防队员非常类似。例如同消防队员的服装标识出他们的身份类似，C++类的头文件表示了这个类所提供的服务及函数。但这一点同 COM 组件是非常不同的。同消防队员或 C++类比较起来，COM 组件更具隐蔽性。例如客户无法得知一个 COM 组件的全部细节，从而也就无法知道一个组件的实现细节。相反客户必须向组件提出一系列的问题，例如：你使用氧气罐吗？你使用斧头吗？你穿防水服装吗？等等。

实际上对于客户而言，它所询问的组件是否真的是一个消防队员是无关紧要的。客户所真正关心的问题是组件对它所提出问题的回答是什么。例如，若客户询问：你穿防水服装吗？若被询问的组件是一个爱斯基摩人、一个水中呼吸器或一个伐木工人，对于客户来说都是可以的。若客户询问：你使用氧气罐吗？那么对于所询问的对象为水中呼吸器或消防队员的情形，客户均可以得到所需的回答。但当客户询问：你使用斧头吗？那么将只有消防队员能够给出令客户满意的答复。当然如果有带着水中呼吸器的潜水伐木工，那么客户也将可以得到满意的回答。

这种不让客户了解组件身份特征的能力可以使得客户能够尽可能少地受组件变化的影响。但由于客户对组件知道的仅仅是其接口，因此客户也就无法直接控制整个组件的生命期。在这一章中，我们就来介绍一种通过直接控制单个接口生命期来间接控制组件生命期的方法。

4.1 生命期控制

首先，我们来看一下客户为什么不应直接控制组件的生命期。假定某个组件正在使

用前面我们提到过的消防队员组件。在客户的代码中,可能会有若干个指向此组件接口的指针。例如客户的某一部分可能会通过一个 IUseOxygen 来"吸取"氧气罐中的氧气,而另外一部分则使用接口 IUseAnAxe 来"砍一间房子"。在使用完 IUseOxygen 之后可能还希望继续使用 IUseAnAxe。这种情况下,当使用完一个接口而仍要使用另外一个接口时,是不能将组件释放掉的。由于很难知道两个接口指针是否指向同一组件,因此决定何时可以安全地释放一个组件将是极为复杂的。得知两个接口指针是否是指向同一对象的唯一方法是查询这两个接口的 IUnknown,然后对结果进行比较。当程序越来越复杂时,决定何时可以释放一个组件将是极为复杂的。解决这个问题的最为简单的方法是在整个程序的运行期内一直装载相应的组件。但这种方法的效率可以想见不是很高。

所以就有如下的方法:我们可以通知组件何时需要使用它的某个接口以及何时用完此接口,而不是直接将接口删除。一般我们总能精确地知道何时开始使用一个接口,并且在大多数情况下,可以精确地知道何时将用完此接口。但正如前面所看到的,决定何时将用完整个组件并不是一件容易的事情。因此,当用完某个接口后给组件发出指示将比在使用完整个组件更有意义。对组件的释放可以由组件在客户使用完其各个接口之后自己完成。

IUnknown 的另外两个成员函数 AddRef 和 Release 的作用就是给客户提供一种让它指示何时处理完一个接口的手段。前一章中我们曾经给出过 IUnknown 的定义。为清楚起见,在此我们再次给出此定义。

```
interface IUnknown
{
    virtual HRESULT _stdcall QueryInterface(const IID& iid, void** ppv) = 0;
    virtual ULONG _stdcall AddRef() = 0;
    virtual ULONG _stdcall Release() = 0;
}
```

本章我们将讨论 AddRef 和 Release 如何可以实现让组件自己来管理其生命期,以及客户又如何可以借助于这两个函数以使它们只需关心接口。首先,我们将简要介绍引用计数的概念,然后我们再讨论客户可以如何使用 AddRef 和 Release。在熟悉了 AddRef 和 Release 之后,我们将实现一个组件中的这两个成员。最后我们将讨论在什么情况下可以省去对 AddRef 和 Release 的调用以提高性能,并将我们所讨论的结果总结成一些简单的规则。

4.2 引用计数简介

AddRef 和 Release 实现的是一种名为引用计数的内存管理技术。引用计数是使组件能够自己将自己删除的最简单同时也是效率最高的方法。COM 组件将维护一个称作是

引用计数的数值。当客户从组件取得一个接口时,此引用计数值将增1。当客户使用完某个接口后,组件的引用计数值将减1。当引用计数值为0时,组件即可将自己从内存中删除。当创建某个已有接口的另外一个引用时,客户也将会增大相应组件的引用计数值。读者大概已经可以猜出,AddRef可增大引用计数值,而Release将减少这一值。

为正确地使用引用计数,读者需要了解以下三条简单的规则:

1 在返回之前调用AddRef。对于那些返回接口指针的函数,在返回之前应用相应的指针调用AddRef。这些函数包括QueryInterface及CreateInstance。这样当客户从这种函数得到一个接口后,它将无需调用AddRef。

2 使用完接口之后调用Release。在使用完某个接口之后应调用此接口的Release函数。

3 在赋值之后调用AddRef。在将一个接口指针赋给另外一个接口指针时,应调用AddRef。换句话说,在建立接口的另外一个引用之后应增加相应组件的引用计数。

这些就是关于引用计数的三条简单规则。下面我们来看两个例子。第一个例子将说明前面两条规则。在下面的代码中,我们建立了一个组件并取得了一个指向接口IX的指针。由于在CreateInstance和QueryInterface中已经调用了AddRef,因此这里不再调用它。但对于CreateInstance所返回的IUnknown接口以及QueryInterface所返回的IX接口,却都需要调用Release。

```
// Create a new component.
IUnknown* pIUnknown = CreateInstance();

// Get interface IX.
IX* pIX = NULL;
HRESULT hr =
pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
if (SUCCEEDED(hr))
{
    pIX->Fx();              // Use interface IX.
    pIX->Release();         // Done with IX.
}
pIUnknown->Release();       // Done with IUnknown.
```

在上面的例子中,实际上在调用完QueryInterface之后,对IUnknown接口的使用也就结束了。因此下面需要将IUnknown释放。

```
// Create a new component.
IUnknown* pIUnknown = CreateInstance();

// Get interface IX.
IX* pIX = NULL;
HRESULT hr =
```

```
pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
// Done with IUnknown.
pIUnknown->Release();
// Use IX if we succeeded in getting it.
if (SUCCEEDED(hr))
{
    pIX->Fx();          // Use interface IX.
    pIX->Release();     // Done with IX.
}
```

第三条规则是许多人都会忘记的。在下面的代码段中,我们建立了对接口 IX 的另外一个引用。一般而言,每当复制一个接口的指针时,都应像前面的第三条规则所说的那样相应地增加引用计数。

```
IUnknown* pIUnknown = CreateInstance();
IX* pIX = NULL;
HRESULT hr =
pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
pIUnknown->Release();
if (SUCCEEDED(hr))
{
    pIX->Fx();          // Use interface IX.
    IX* pIX2 = pIX;     // Make a copy of pIX.
    pIX2->AddRef();     // Increment the reference count.
    pIX2->Fx();         // Do something with pIX2.
    pIX2->Release();    // Done with IX2.
    pIX->Release();     // Done with IX.
}
```

看完上述代码之后,读者的第一个反应也许是:在本例中必须对 pIX2 调用 AddRef 和 Release 吗?或怎么才能记住在每次复制指针时都去调用 AddRef 和 Release 呢?某些读者可能会同时产生这样两个疑问。对于第一个问题的答案是:不,在上例中并不一定要对 pIX2 调用 AddRef 和 Release。在像本例这样一个简单的例子中,很容易看出通过 pIX2 增大然后又减少引用计数并不是必要的。因为 pIX2 的生命期和 pIX 的生命期是一样的。在本章的后面我们将介绍一些如何优化引用计数的指导性原则。但是一般而言,在给某个接口指定一个别名时,都应调用 AddRef。在实际程序中,很难决定某些没有加上的 AddRef 及 Release 调用到底是优化还是程序中的错误。由于很多人经常花费许多天的时间以试图找出程序中不正确计数的原因,因此,可以肯定地讲引用计数引起的问题并不是那么容易就能够找出来的。

但在第 10 章中我们将看到,智能指针类可以把引用计数完全封装起来。在使用智能指针时,程序员几乎可以完全将引用计数的问题忘掉。

我们知道当客户调用 QueryInterface 时，它实际上是告诉相应的组件它想要使用某个接口。在上面的代码中我们看到，QueryInterface 将调用所请求接口上的 AddRef。当客户用完某个接口之后，它将调用此接口上的 Release。只要组件的引用计数值不为零，那么它将一直处于内存中。当引用计数为零时，组件将把它从内存中删除。

4.2.1 引用计数接口

在此有一个细节问题需要提醒一下。在客户看来，引用计数是处于接口级上而不是组件级上的。不知读者是否还记得关于消防队员例子。客户并不了解组件的全部，它看到的只是组件的一个个接口。因此在客户看来，每一个接口都有一个引用计数。

虽然从客户的角度看来是接口而不是组件被记录下其引用计数(参见图 4-1)，但从实现的角度来看，谁的引用计数被记录下来实际上并没有什么关系。组件可以对其每一个接口分别维护一个引用计数，也可以对整个组件维护单个的引用计数。只要能够使客户相信组件将记录每个接口本身维护引用计数值，使用何种实现方法是无关紧要的。由于在组件的实现中可能会选择对每一个接口分别维护一个引用计数，因此 客户不应作相反的假设。

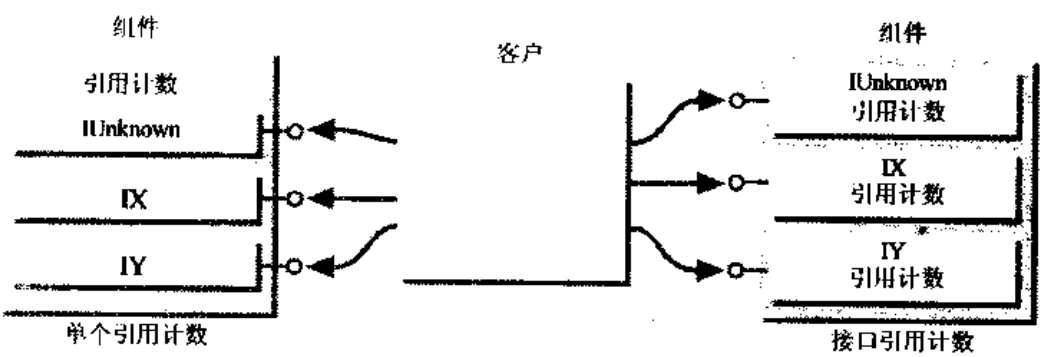


图 4-1 组件可以选择是对整个组件维护一个引用计数还是对每一个接口分别维护一个引用计数

对于客户而言，每一个接口被分别维护一个引用计数意味着什么呢？它意味着客户应该对它将要使用的那个指针调用 AddRef，而不是其他的什么指针。对于使用完了指针，客户应调用其 Release。例如，不要编写如下形式的代码：

```
IUnknown * pIUnknown = CreateInstance() ;
IX * pIX =  NULL ;
pIUnknown -> QueryInterface(IID_IX, &pIX) ;
pIX -> Fx() ;
pIX * pIX2 = pIX ;
pIUnknown -> AddRef() ;     // Should be pIX2 -> AddRef();
```

```
pIX2->Fx();
pIX2->Release();
pIUnknown->Release();   // Should be pIX->Release();
pIUnknown->Release();
```

上面的代码假定像使用 IX 接口指针那样对 pIUnknown 接口调用 AddRef 和 Release。根据组件实现方式的不同,这可能会引起一些问题。

那么为什么要选择为每一个接口单独维护一个引用计数而不是针对整个组件维护引用计数呢?这主要有两个原因:一是使程序调试更为方便;另外一个原因是支持资源的按需获取。

一、程度调试

假定在程序中忘了对某个接口调用 Release——实际上这一点是很多人经常会犯的一个错误。这样组件将永远不会被删除掉,因为只是在引用计数值为 0 时 delete 才会被调用。这时就需要找出接口在何时何处应该被释放掉。当然找起来是相当困难的。在只对整个组件维护一个接口的情况下,进行这种查找就更为困难了。此时必须检查使用了此组件所提供的所有接口的代码。但若组件支持对每一个接口分别维护一个引用计数,那么可以把查找的范围限制在某个特定的接口上。在某些情况下这可以节省大量的时间。

二、资源的按需获取

在实现某个接口时可能需要大量的内存或其他资源。对于此种情况,可以在 QueryInterface 的实现中,在客户请求此接口时完成资源的分配。但若只对整个组件维护一个引用计数,组件将无法决定何时可以安全地将同此接口相关联的内存释放。但若对每一个接口分别维护一个引用计数,那么决定何时可以将此内存释放将会容易得多。

另外一种作法是(大多数情况下这种作法更优)是在另外一个单独的组件中实现此种需要大量资源的接口,然后将指向此组件的接口传给客户。这种被称作是聚合的技术将在第 8 章中详细讨论。

为使本书中所用的例子尽量地简单,我们将对整个组件维护单个的引用计数值。下面我们来看一下如何实现这一点。

4.2.2 AddRef 和 Release 的实现

AddRef 和 Release 的实现实际上是非常简单的。实际上可以通过增大和减少某个数值而实现之。例如下面的代码给出了一个实现的例子。

```
ULONG _stdcall AddRef()
{
```

```
    return ++m_cRef;
}
ULONG _stdcall Release()
{
    if (--m_cRef == 0)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}
```

AddRef将记录有组件引用计数值的成员变量m_cRef增大，而Release则将此值减少，并在此值为0时将组件删除。在许多情况下，许多程序员可能会使用Win32的函数InterlockedIncrement和InterlockedDecrement来实现AddRef和Release。这两个函数可以确保在同一时刻只会有同一个线程来访问成员变量。根据COM对象所使用的线程模型的不同，多线程可能会引起一些问题。关于线程，我们将在第12章中讨论。

```
ULONG __stdcall AddRef()
{
    return InterlockedIncrement(&m_cRef);
}
ULONG _stdcall Release()
{
    if (InterlockedDecrement(&m_cRef) == 0)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}
```

另外要注意的是AddRef和Release的返回值没有什么意义，只是在程序调试中才可能会用得上。客户不应将此值当成是组件或其接口的精确引用数。

若读者仔细阅读过第3章中给出的示例代码，可以发现在QueryInterface和CreateInstance中都使用了AddRef。

```
HRESULT _stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
    {
        *ppv = static_cast<IX*>(this);
    }
  else if (iid == IID_IX)
```

```
        {
            *ppv = static_cast<IX*>(this);
        }
        else if (iid == IID_IY)
        {
            *ppv = static_cast<IY*>(this);
        }
        else
        {
            *ppv = NULL;
            return E_NOINTERFACE;
        }
    static_cast<IUnknown*>(*ppv)->AddRef();
        return S_OK;
}

IUnknown* CreateInstance()
{
    IUnknown* pI = *ppv = static_cast<IX*>(new CA);
    pI->AddRef();
    return pI;
}
```

这样当建立一个新组件时，实际上也将建立一个对此组件的引用。因此创建组件时，在将指针返回给客户之前，应该增大组件的引用计数值。这使得程序员可以不必在调用 CreateInstance 或 QueryInterface 之后记着去调用 AddRef。

在某些情况下可以省去对 AddRef 和 Release 的调用。在介绍如何将那些不是必需的 AddRef 和 Release 调用去掉之前，我们先来看一下程序清单 4-1。此清单将前面给出的代码片段组合成了一个完整的例子。

程序清单 4-1　关于引用计数的一个完整例子。

REFCOUNT.CPP

```
//
// RefCount.cpp
// To compile, use : cl RefCount.cpp UUID.lib
//
#include <iostream.h>
#include <objbase.h>
void trace(const char* msg) { cout << msg << endl; }
// Forward references for GUIDs
extern const IID IID_IX;
extern const IID IID_IY;
extern const IID IID_IZ;
// Interfaces
```

```
interface IX : IUnknown
{
  virtual void __stdcall Fx() = 0;
};
interface IY : IUnknown
{
  virtual void __stdcall Fy() = 0;
};
interface IZ : IUnknown
{
  virtual void __stdcall Fz() = 0;
};
//
// Component
//
class CA : public IX,
           public IY
{
  // IUnknown implementation
  virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall AddRef();
  virtual ULONG __stdcall Release();

  // Interface IX implementation
  virtual void __stdcall Fx() { cout << "Fx" << endl; }

  // Interface IY implementation
  virtual void __stdcall Fy() { cout << "Fy" << endl; }
public:
  // Constructor
  CA() : m_cRef(0) {}

  // Destructor
  ~CA() { trace("CA:     Destroy self."); }
private:
  long m_cRef;
};
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
  if (iid == IID_IUnknown)
  {
    trace("CA QI:  Return pointer to IUnknown.");
    *ppv = static_cast<IX*>(this);
  }
```

```
    else if (iid == IID_IX)
    {
        trace("CA QI:  Return pointer to IX.");
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IY)
    {
        trace("CA QI:  Return pointer to IY.");
        *ppv = static_cast<IY*>(this);
    }
    else
    {
        trace("CA QI:  Interface not supported.");
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}
ULONG __stdcall CA::AddRef()
{
    cout << "CA:     AddRef = " << m_cRef+1 << '.' << endl;
    return InterlockedIncrement(&m_cRef);
}
ULONG __stdcall CA::Release()
{
    cout << "CA:     Release = " << m_cRef-1 << '.' << endl;
    if (InterlockedDecrement(&m_cRef) == 0)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}
//
// Creation function
//
IUnknown* CreateInstance()
{
    IUnknown* pI = static_cast<IX*>(new CA);
    pI->AddRef();
    return pI;
}
```

```
//
// IIDs
//
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
const IID IID_IX =
    {0x32bb8320, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;
// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
const IID IID_IY =
    {0x32bb8321, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;
// {32bb8322-b41b-11cf-a6bb-0080c7b2d682}
const IID IID_IZ =
    {0x32bb8322, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;
//
// Client
//
int main()
{
    HRESULT hr ;
    trace("Client: Get an IUnknown pointer.") ;
    IUnknown* pIUnknown = CreateInstance() ;
    trace("Client: Get interface IX.") ;
    IX* pIX = NULL ;
    hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX) ;
    if (SUCCEEDED(hr))
    {
        trace("Client: Succeeded getting IX.") ;
        pIX->Fx() ;          // Use interface IX.
        pIX->Release() ;
    }
    trace("Client: Get interface IY.") ;
    IY* pIY = NULL ;
    hr = pIUnknown->QueryInterface(IID_IY, (void**)&pIY) ;
    if (SUCCEEDED(hr))
    {
        trace("Client: Succeeded getting IY.") ;
        pIY->Fy() ;          // Use interface IY.
        pIY->Release() ;
    }
    trace("Client: Ask for an unsupported interface.") ;
```

```
    IZ * pIZ = NULL ;
    hr = pIUnknown -> QueryInterface(IID_IZ, (void* * )&pIZ) ;
    if (SUCCEEDED(hr))
    {
        trace("Client: Succeeded in getting interface IZ.") ;
        pIZ -> Fz() ;
        pIZ -> Release() ;
    }
    else
    {
        trace("Client: Could not get interface IZ.") ;
    }
    trace("Client: Release IUnknown interface.") ;
    pIUnknown -> Release() ;
    return 0;
}
```

上述程序的输出如下：

```
Client: Get an IUnknown pointer.
CA:     AddRef = 1.
Client: Get interface IX.
CA QI:  Return pointer to  IX.
CA:     AddRef = 2.
Client: Succeeded getting IX.
Fx
CA:     Release = 1.
Client: Get interface IY.
CA QI:  Return pointer to IY.
CA:     AddRef = 2.
Client: Succeeded getting IY.
Fy
CA:     Release = 1.
Client: Ask for an unsupported interface.
CA QI:  Interface not supported.
Client: Could not get interface IZ .
Client: Release IUnknown interface.
CA:     Release = 0.
CA:     Destroy Self.
```

上例同第 3 章中给出的例子是一样的，只不过在其中加入了引用计数的功能。就组件而言，它们实现了 AddRef 和 Release。从客户的角度来讲，唯一的不同在于加上了一些 Release 调用以告知组件客户用完了各接口。另外还可以注意到，客户不再使用 delete 操作符了。并且在上例中，由于 CreateInstance 和 QueryInterface 用它们所返回的指针调用了

AddRef,因此客户无需再去调用此函数。

4.3 何时进行引用计数

下面我们来看一下在什么情况下需要进行引用计数。我们将看到在某些情况下可以省略对 AddRef 和 Release 的调用,从而可以优化程序代码。将前面给出的关于引用计数的一些规则同此处关于优化的一些考虑结合起来,将可以得到关于引用计数一些一般性规则。

4.3.1 引用计数的优化

前面曾提到过这样一个问题:是否在复制一个接口指针时都需要增大其引用计数。为说明这个问题我们先看一下下面的代码:

```
HRESULT hr ;
IUnknown * pIUnknown = CreateInstance() ;
IX * pIX = NULL;
hr = pIUnknown -> QueryInterface(IID_IX, (void * *)&pIX) ;
pIUnknown -> Release() ;
if (SUCCEEDED)hr))
{
IX * pIX2 = pIX ;  // Make a copy of pIX.
    // The life of pIX2 is nested in the life of pIX.
    pIX2 -> AddRef() ;      // Increment the reference count.
    pIX -> Fx() ;           // Use interface IX.
    pIX2 -> Fx() ;          // Do something with pIX2.
    pIX2 -> Release() ;     // Done with pIX2.
pIX -> Release() ;          // Done with IX.
                            // Also done with the component.
}
```

对于上面的这段代码,只有当客户将 pIX 释放时组件才会被从内存中删除。而客户只是在用完了 pIX 和 pIX2 之后才会将 pIX 释放。正是由于组件只是在 pIX 被释放之后才被从内存中删除掉,因此可以保证在 pIX2 的生命期内相应的组件将一直在内存中。因此对 pIX2 并不真的需要调用 AddRef 和 Release。从而上面代码中加粗的那两行可以删除掉。

为将上例中的组件保持在内存中,pIX 的单个引用计数就足够了。之所以会这样,关键之处在于 pIX2 的生命期包含在 pIX 的生命期内。为强调这一点,在上面的代码中我们故意增大了使用 pIX2 的那些代码行的缩进量。图 4-2 图示了 pIX 和 pIX2 生命期的嵌套关系。图中各灰色条表示各接口指针及组件本身的生命期,时间是由上到下的。图的左

边列出的是那些将影响各生命期的操作。水平虚线则表示出这些操作如何开始和结束各接口的生命期。

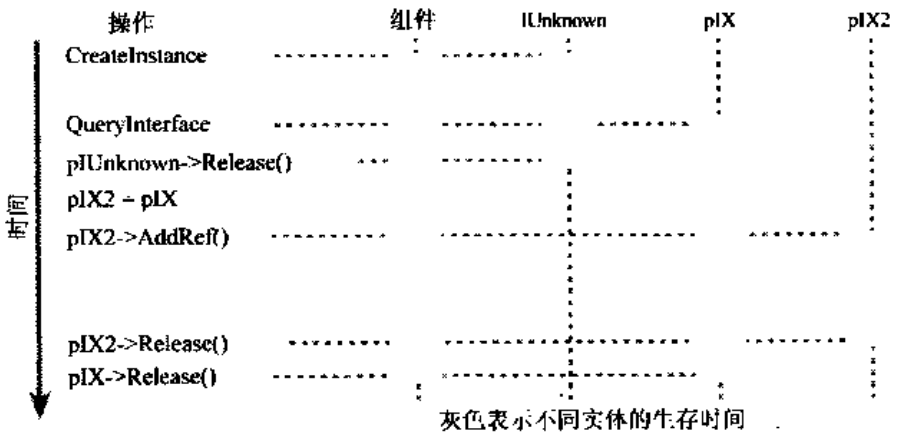


图 4-2 生命期嵌套的接口指针图示。对于生命期嵌套在其他接口指针中的接口指针，无需进行引用计数

从图 4-2 中可以很容易地看出，接口 pIX2 的生命期起于 pIX 之后而止于 pIX 之前。因此 pIX 的引用计数将足以保证在 pIX2 的生命期内，组件将一直被保持在内存中，若 pIX2 的生命期不包含在 pIX 的生命期内，那么就需要对 pIX2 进行引用计数。例如在下面的代码中，pIX2 的生命期同 pIX 的生命期是重叠的。

```
IUnknown * pIUnknown = CreateInstance() ;
IX * pIX = NULL;
HRESULT hr =
pIUnknown -> QueryInterface(IID_IX, (void* * )&pIX) ;
pIUnknown -> Release() ;

if (SUCCEEDED)hr))
{
IX * pIX2 = pIX ;          // Make a copy of pIX.
pIX2 -> AddRef() ;         // pIX2 starts life.

pIX -> Fx() ;
pIX -> Release() ;         // pIX ends life.

pIX2 -> Fx() ;
pIX2 -> Release() ;        // pIX2 ends life.
                           // Also done with the component.
}
```

在上例中，必须对 pIX2 调用 AddRef，因为 pIX2 释放是在 pIX 的释放之后。图 4-3 图示了这种情况。

在上面给出的这些简单的例子中，我们一眼就能看出是否需要进行引用计数。但实

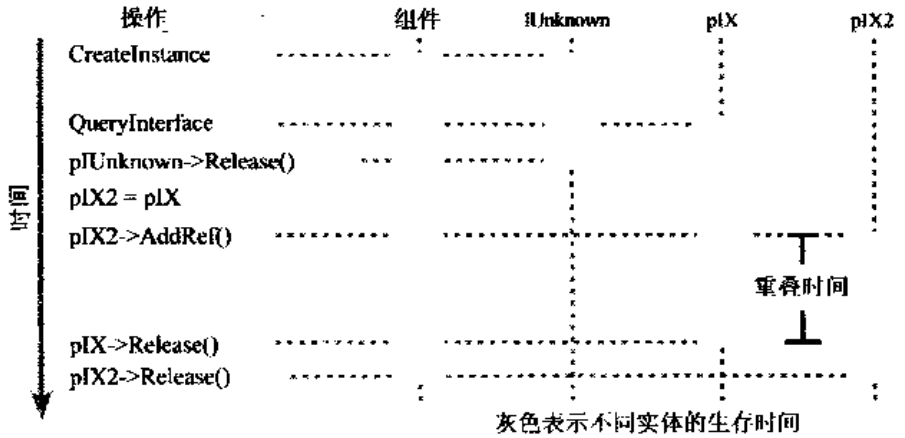


图 4-3 生命期出现重叠的接口指针图示。此种情况下对这两个接口指针都必须进行引用计数

际开发过程中要找出嵌套的生命期则不是一件容易的事情。但不管怎么样,在某些情况下,如函数等,接口的生命期还是比较明显的。在下面的代码中,显然 foo 的生命期包含在 pIX 的生命期中。因此对于传递给 foo 的接口指针,无需调用 AddRef 和 Release。

```
void foo(pIX* pIX2)
{
    pIX2->Fx();              // Use interface IX.
}

void main()
{
    HRESULT hr;
    IUnknown* pIUnknown = CreateInstance();
    IX* pIX;
    hr = pIUnknown->QueryInterface(IID_IX, (void**)&pIX);
    pIUnknown->Release();

    if (SUCCEEDED)hr))
    {
        foo(pIX);            // Pass pIX to procedure.
        pIX->Release();      // Done with IX.
                             // Also done with the component.
    }
}
```

在函数中,无需对存在于局部变量的接口指针进行引用计数。因为局部变量的生命期同函数的生命期是一样的,因此也将包含在调用者的生命期内。但当从某个全局变量或向某个全局变量复制一个指针时,则需要对此指针进行引用计数,这是因为全局变量可以从任意函数中的任意地方被释放。

从上面的讨论可以看到，为对引用计数进行优化，关键是找出那些生命期嵌套在引用同一接口的指针生命期内的接口指针。在实际的代码中，找出这种嵌套生命期可能是比较困难的。下一节列出的规则将给出在哪些地方可以省略 AddRef/Release 对，而不会冒太大的风险。

4.3.2 引用计数规则

本节给出的规则将综合前一节及本章开头给出的一些引用计数优化策略。在学习这部分内容时，应记住客户必须像每一个接口具有一个单独的引用计数值那样来处理各接口。因此，客户必须对不同的接口分别进行引用计数，即使它们的生命期是嵌套的。

一、输出参数规则

输出参数指的是给函数的调用者传回一个值的函数参数。在函数体中将设置此输出参数的值而不会使用调用者传进来的值。从这一点上讲，输出参数的作用同函数的返回值是类似的。例如 QueryInterface 的第二个参数就是一个输入参数。

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

任何在输出参数中或作为返回值返回一个新的接口指针的函数必须对此接口指针调用 AddRef。这一规则同本章开头给出的"在返回之前调用 AddRef"规则是一样的，只不过是换了一种提法。QueryInterface 函数的实现就遵循了这一规则，它在返回之前对接口指针调用了 AddRef。我们所实现的组件的 CreateInstance 函数也遵循了这一规则。

二、输入参数规则

输入参数指的是给函数传递某个值的参数。在函数体中将会使用这个值但却不会修改它或将其返回给调用者。在 C++ 中，输入参数实际上就是那些按值传递的参数或常量。下面的代码就是将接口指针作为输入参数的一个例子。

```
void foo(pIX * pIX)
{
    pIX -> Fx() ;
}
```

对传入函数的接口指针，无需调用 AddRef 和 Release，这是因为函数的生命期嵌套在调用者的生命期内。为记住这一点，可以想象将函数的代码以内联方式展开的情形。例如：

```
IX * pIX = CreateInstance () ; //Automatic AddRef
foo(pIX) ;
pIX -> Release() ;
```

在将 foo 以内联方式展开时，上面的代码将变成：

```
IX* pIX = CreateInstance();  //Automatic AddRef
//foo(pIX);
pIX->Fx();   //Expanded foo function
pIX->Release();
```

如此将可以清楚地看出 foo 的生命期是嵌套在调用者的生命期之内的。

三、输入-输出参数规则

输入-输出参数同时具有输入参数及输出参数的功能。在函数体中可以使用输入-输出参数的值，然后可以对这些值进行修改并将其返回给调用者。

在函数中，对于用输入-输出参数传递进来的接口指针，必须在给它赋另外一个接口指针值之前调用其 Release。在函数返回之前，还必须对输出参数中所保存的接口指针调用 AddRef。如下例所示。

```
void ExchangeForCachedPtr(int i, IX** ppIX)
{
    (*ppIX)->Fx();          // Do something with in-parameter.
    (*ppIX)->Release();     // Release in parameter.
    *ppIX = g_Cache[i];     // Get cached pointer.
    (*ppIX)->AddRef();      // AddRef pointer.
    (*ppIX)->Fx();          // Do something with out-parameter.
}
```

四、局部变量规则

对于局部复制的接口指针，由于它们只是在函数的生命期内才存在，因此无需调用 AddRef 和 Release。这条规则实际是输入参数规则的直接结果。在下面的例子中，pIX2 只是在函数 foo 的生命期内才存在，因此可以保证其生命期将嵌套在所传入的 pIX 指针的生命期内，因此无需对 pIX2 调用 AddRef 和 Release。

```
void foo(IX* pIX)
{
    IX* pIX2 = pIX;
    pIX2->Fx();
}
```

五、全局变量规则

对于保存在全局变量中的接口指针，在将其传递给另外一个函数之前，必须调用其 AddRef。由于此变量是全局性的，因此任何函数都可以通过调用其 Release 来终止其生命

期。对于保存在成员变量中的接口指针，也应按此种方式进行处理。因为类中的任何成员函数都可以改变此种接口指针的状态。

六、不能确定时的规则

对于任何不能确定的情形，都应调用 AddRef 和 Release 对。省略 AddRef 和 Release 对的调用不会对性能或内存的分配带来什么改善，却极容易使得某个组件永远不会被从内存中释放掉。并且这样做还会使程序员在跟踪由于引用计数错误而引起的错误方面花费大量的时间。在这种情况下可以用一个测试程序来测试一下此种优化是否真的有什么效果。另外，在决定要进行优化时，应给那些没有进行引用计数的指针加上相应的注释。否则其他程序员在修改代码时，将可能会增大接口指针的生命期，从而使引用计数的优化遭到破坏。

忘记调用 Release 造成的错误可能比不调用 AddRef 造成的错误更难检测。而这一点正是 C++ 程序员易犯的一个毛病，大多在某些情况下他们可能会使用 delete 操作符。在第 9 章中我们将看到智能指针如何将引用计数完全封装起来。

4.4 本章小结

IUnknown 的成员函数使得程序员可以对接口进行完全的控制。在上一章中，我们已经看到如何通过 QueryInterface 来获取某个组件所支持的接口。在这一章中，我们重点讨论了如何通过 AddRef 和 Release 来控制得到的接口的生命期。AddRef 实际上是告诉组件我们想要使用某个接口，而 Release 则告诉组件我们用完了某个接口。同时 Release 也给组件提供了一些控制其生命期的能力。客户并不需要直接地将组件从内存中删除，相反 Release 将告诉组件客户使用完了某个接口。若没有任何客户还在使用组件的任何一个接口，那些组件可以将自己从内存中删除。

虽然对于接口我们现在可以进行完全的控制了。但另外有一个非常重要的问题我们还没有讨论到：动态链接。一个没有动态链接能力的组件就如同一个没有套装、头盔及长筒靴的消防队员。在下一章中我们将讨论如何给组件加上动态链接的能力。

第5章 动态链接

回顾一下第1章,在那里,我们特别强调了动态链接对于用组件来构造软件系统的重要性。但现在已经讲到了第5章,我们所构造的客户和组件不但是静态链接在一起的,而且可以看出它们实际上是在同一文件中。这样安排的原因在于只有在完全实现了IUnknown之后,才可以改变客户和组件之间的那种密不可分的联系。

开始时,在不要求客户发生变化的情况下,组件不可能发生变化。然后我们使用QueryInterface将组件抽象成由多个相互独立的接口构成的集合。将组件分成多个接口只是将铁板一块状的应用程序分割成多个部分的第一步。下面我们需要使用某种方法来管理组件的生命期。这一点是通过对接口的引用记数实现的。通过引用记数,客户可以控制接口的生命期,而组件本身可以决定何时将它从内存中删除。在实现了IUnknown之后,组件和客户之间只是一种非常松散的连接,这种松散的连接关系使得组件和客户各自可以发生变化而不会对对方造成什么影响。

在这一章中我们将讨论如何将组件放入到动态链接库(DLL)中。需要提醒读者注意的是,这并不是我们要将一个组件变成为一个DLL。一个组件实际上并不是一个DLL,将组件当成DLL来看待是非常肤浅的。DLL只是一个组件服务器,或者说是一种发行组件的方式。组件实际上应看成是在DLL中所实现的接口集。DLL只是一种形式,而组件才是实质。

在这一章中我们将通过介绍客户如何建立在一个DLL中实现的组件来讨论什么是动态链接。然后我们将对第4章中的程序清单4-1进行处理,以将客户和组件划分到不同的文件中。在给出实际的代码之后,我们将建立使用三种不同接口不同组合形式的三个不同客户及组件。最后我们将把客户和组件混合起来。

若读者已经熟悉了DLL的概念,那么对于本章所讨论的大部分内容应该有所了解了。但是在"5.2 客户和组件的划分"一节中给出的内容仍然是需要看一看的。另外在"5.3 对象串"一节中给出的内容也是相当有趣的。

5.1 组件的创建

在这一节中我们将介绍如何将组件动态地链接到客户中。首先我们将讨论一下客户将如何创建组件。当然这里所给出的创建组件的方法也是临时性的。在后续的章节中我们将看到如何将客户和组件进一步分隔开来。

在客户可以获取某个组件接口指针之前，它必须先将相应的 DLL 装载到其进程空间中并创建此组件。第 3 章中给出的 CreateInstance 可以建立一个组件的实例并给客户返回一个 IUnknown 接口指针。这是 DLL 中唯一需要客户显式链接的函数。由于组件中客户所需的所有函数都可以通过一个接口指针而访问到，因此需要在 DLL 中输出 CreateInstance 函数以便客户可以调用它。

5.1.1 从 DLL 中输出函数

从 DLL 中输出一个函数并不是什么困难的事情。为此，可先将此需要输出的函数用 extern "C"进行标记。例如在 CMPNT1.CPP 中，CreateInstance 函数的定义如下：

```
//
// Creation function
//
extern "C"
IUnknown * CreateInstance()
{
   IUnknown * pI = (IUnknown *)(void *)new CA;
   pI->AddRef();
   return pI;
}
```

在函数的定义前加上 extern "C"可防止 C++编译器在函数名称上加上类型信息。例如在不加 extern "C"时，Microsoft Visual C++将把 CreateInstance 变成：

```
? CreateInstance@@YAPAUUnknown@@XZ
```

其他的编译器可能会使用另外某种名称修改方法。对此并没有什么标准，因此变换以后的函数是不可移植的。并且对这种变换以后的名称处理起来也非常困难。

输出函数定义的转储

在使用 Microsoft Visual C++时，可以用 DUMPBIN.EXE 来得到某个 DLL 中所输出的符号的清单。例如下面的命令：

```
dumpbin -exports Cmpnt1.dll
```

将产生如下的输出：

```
Microsoft (R) COFF Binary File Dumper Version 4.20.6281
Copyright (C) Microsoft Corp 1992-1996. All rights reserved.

Dump of file component1.dll

File Type: DLL
```

```
    Section contains the following Exports for Compnt1.dll
            0 characteristics
     325556C5 time date stamp Fri Oct 04 11:26:13 1996
         0.00 version
            1 ordinal base
            1 number of functions
            1 number of names

     ordinal hint   name
           1   0    CreateInstance  (00001028)

Summary

   7000 .data
   1000 .idata
   3000 .rdata
   2000 .reloc
  1D000 .text
```

当然从DLL中输出函数仅仅是将函数加上extern "C"标记是不够的,还需要告诉链接程序需要输出什么函数。为此,需要建立一个"讨厌"的DEF函数。之所以这么讲主要是因为程序员经常忘记将要输出的函数的名称加入到此文件中。这样造成的后果是客户将无法使用此函数。作者在这方面是有深切体会的。

DEF文件建立起来也是比较麻烦的。为此可以在某个例子中复制一个DEF文件,然后修改其中的某些行。在程序清单5-1中给出了CMPNT1.DLL的DEF文件CMPNT1.DEF。

程序清单5-1 在模块定义文件中列出了动态链接库所输出的函数的名称。

```
CMPNT1.DEF
;
; Compnt1 module-definition file.
;
LIBRARY         Compnt1.dll
DESCRIPTION     '(c) 1996-1997 Dale E.Rogerson'

EXPORTS
                CreateInstance @1 PRIVATE
```

对上述文件所需要做的全部工作就是在EXPORTS段中列出待从DLL中输出的函数的名称。对每一个名称,还可以加上一个序号。在LIBRARY行上必须加上DLL的实际名称。

从DLL中输出函数所需要做的工作就是这些。下面我们看一下如何装载所需的DLL并调用其中的函数。

5.1.2 DLL 的装载

在文件 CREATE.H 和 CREATE.CPP 中实现了函数 CallCreateInstance。这个函数以 DLL 的名称作为参数。其功能是装载此 DLL 并调用其中所输出的函数 CreateInstance。其代码如程序清单 5-2 所示。

程序清单 5-2 使用 LoadLibrary 和 GetProcAddress 客户可以动态地连接到组件上。

CREATE.CPP

```
//
// Create.cpp
//
#include <iostream.h>
#include <unknwn.h>     //Declare IUnknown.
#include "Create.h"
typedef IUnknown* (*CREATEFUNCPTR)();
IUnknown* CallCreateInstance(char* name)
{
  // Load dynamic link library into process.
  HINSTANCE hComponent = ::LoadLibrary(name);
  if (hComponent == NULL)
  {
     cout << "CallCreateInstance:\tError: Cannot load component." << endl;
     return NULL;
  }
  // Get address for CreateInstance function.
  CREATEFUNCPTR CreateInstance
     = (CREATEFUNCPTR)::GetProcAddress(hComponent, "CreateInstance");
  if (CreateInstance == NULL)
  {
     cout  << "CallCreateInstance:\tError: "
           << "Cannot find CreateInstance function."
           << endl;
     return NULL;
  }
  return CreateInstance();
}
```

为装载指定的 DLL，CallCreateInstance 调用了 Win32 的 LoadLibrary 函数：

```
HISTANCE LoadLibrary(
    LPCTSTR lpLibFileName // filename of DLL
  );
```

LoadLibrary 以被装载的 DLL 的名称作为参数并返回一个指向所装载的 DLL 的句柄。Win32 的 GetProcAddress 函数可以使用此句柄以及待调用的函数的名称，然后返回一个指向此函数的指针：

```
FARPROC GetProcAddress(
    HMODULE hModule,   // handle to DLL module
    LPCSTR lpProcName  // name of function
  );
```

使用上述两个函数，客户可以将所需的 DLL 装载到其地址空间中并获取 CreateInstance 的地址。有了此地址之后，创建所需的组件并获取其 IUnknown 接口的指针就是一件简单的事情了。然后 CallCreateInstance 将把此地址转换成一个可以使用的类型，并像其名称所说明的那样调用 CreateInstance 函数。

但 CallCreateInstance 仍使客户和其组件的实现之间是一种紧密的联系。因为客户并不需要知道实现组件的 DLL 的名称。按照我们的设想，用户应该能够将组件从一个 DLL 中移到另外一个 DLL 中，或将此 DLL 从某个目录移到另外一个目录中。在第 6 章和第 7 章中我们将讨论一种更为一般的、更为灵活的创建组件的方法，以便将客户和组件彻底地分开。在第 7 章中，CallCreateInstance 将被一个名为 CoCreateInstance 的 COM 库函数所代替。但在此之前，我们将一直使用 CallCreateInstance 来创建组件。

使用 DLL 的原因

为什么可以使用 DLL 来实现组件呢？其原因在于 DLL 可以共享它们所链入的应用程序的地址空间。

前面我们讲过，客户和组件是通过接口进行交互的。一个接口实际上是一个指向函数的指针列表(vtbl)。组件将为 vtbl 分配内存并用每一个函数的地址来初始化此表格。为使用 vtbl，客户应该能够访问组件为其 vtbl 所分配的内存。同时它还必须能够理解组件放入到 vtbl 中的各个地址。在 Windows 中，由于动态链接库与客户使用的是同一地址空间，因此客户访问 vtbl 是不成问题的。

在 Windows 中一个正在被执行的程序被称作是一个进程。每一个应用程序(EXE)都将以一个单独的进程运行，每一个进程都有一个 4GB 的地址空间。一个进程中的一个地址同另外一个进程中的某个地址是不同的。由于指针是在不同的地址空间中起作用的，因此不能将一个指针从一个进程传到另外一个进程。为理解这一点，可以类比一下街道地址。例如，对于地址 369 Peachtree St.，它可能位于亚特兰大的一个商业大街上，也可能是位于西雅图的某片咖啡馆中。若没有指定城市名称，这个地址实际上是没有意义的。计算机中的进程就相当于是城市。两个不同进程中的指针可以包含相同的地址值，但它们实际上指向的是不同的物理内存。

所幸的是，动态链接库将驻留在所链接的应用程序的地址空间中，由于 DLL 和 EXE

共享同一进程,因此它们也可以共享同一地址空间。由于这个原因,DLL 经常也被称作是*进程中服务器*(*in-process server*)。在第 10 章中我们将讨论以 EXE 方式实现的"进程外服务器",或"本地及远程服务器"。进程外服务器具有与其客户不同的地址空间,但我们将仍然使用 DLL 来帮助实现进程外服务器同其客户之间的交流。图 5-1 显示了 DLL 是如何映射到其客户的地址空间的。

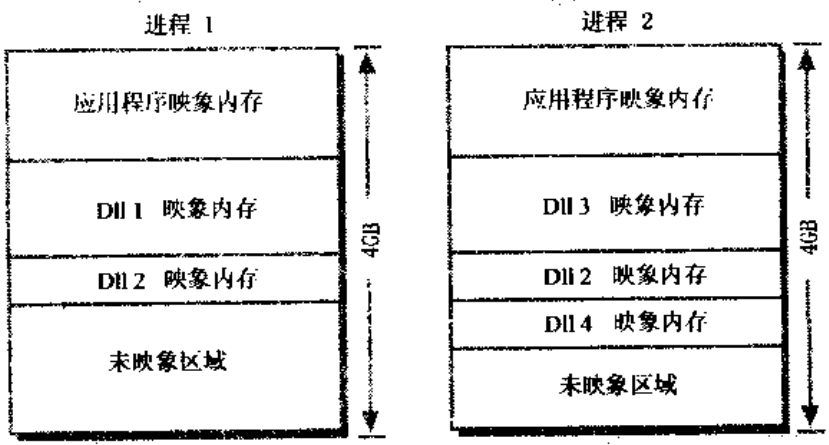


图 5-1 动态链接库将被映射到它们所链接的应用程序的进程空间中

此处的关键之处在于,当客户得到组件的一个接口指针时,连接客户和组件的唯一中介是接口的二进制结构。当客户查询组件的某个接口时,它所请求的实际上是具有特定格式的一块内存。当组件返回一个接口指针时,它告诉客户的实际上是此块内存的地址。由于接口是在客户和组件都能够访问的内存中,因此这种情况实际上与当客户和组件在同一 EXE 文件中是相同的。对于客户而言,唯一的区别在于在静态链接及动态链接的情况下它获取接口指针的方式不同。

5.2 客户和组件的划分

当作者说"大巨人"时,作者父亲经常揶揄道:你能肯定它不是一个小巨人吗?为使作者父亲高兴一下,在此作者将把前面给出的铁板状的应用程序划分到不同的文件中去。在这一节中我们将讨论如何将程序清单 4-1 给出的代码进行划分。然后我们再来看一下各部分的内容。图 5-2 显示了包含有单个客户及单个组件的文件名称。

在此,客户将在 CLIENT1.CPP 中实现。客户文件还包括 CREATE.H 并将同 CREATE.CPP 一块链接起来。这两个文件将把创建包含在 DLL 中组件的过程封装起来。(CREATE.CPP 如程序清单 5-2 所列。)在第 7 章中我们将看到这些文件将被 COM 库中的函数所取代。

组件在此是在一个名为 CMPNT1.CPP 的文件中实现的。由于动态链接需要一个列出

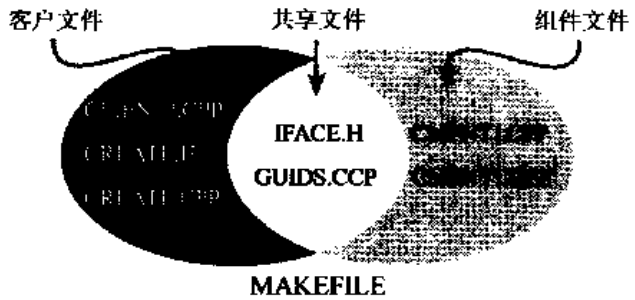


图 5-2 构成客户及组件的文件

从 DLL 中输出的函数名称的模块定义文件,因此在组件文件中还包括一个名为 CMPNT1.DEF 的文件。关于 CMPNT1.DEF,可参见程序清单 5-1。

组件和客户共享了两个文件。其中之一 IFACE.H 包含 CMPNT1 支持的所有接口的声明,其中还包含这些接口的接口 ID 声明。关于这些接口 ID 的定义则在文件 GUIDS.CPP 中。对 GUID 的详细讨论请参阅下一章,在此读者可以不必深究。

为建立客户和组件,可以使用如下的命令:

```
cl Client1.cpp Create.cpp GUIDS.cpp UUID.lib
cl /LD Cmpnt1.cpp GUIDS.cpp UUID.lib Cmpnt1.def
```

但是由于有三个客户及三个组件,为此使建立过程更加方便,作者编写了一个 makefile 文件,并尽量使之具有较高的可读性。有了此 makefile,输入如下的命令即可建立好所有的组件及客户:

```
nmake -f makefile
```

关于示例中文件的说明就这么多了。在本书的后面这些文件的名称及用途将基本上保持不变(当然其内容可能会发生相应的变化)。

5.2.1 程序清单

下面我们来看一下实际的代码,特别需要关注一下客户是如何实现的,因为这是本章关键之所在。程序清单 5-3 列出了客户的实现代码。在此,客户首先要求用户输入 DLL 的名称,然后将此名称传递给 CallCreateInstance。CallCreateInstance 将调用指定 DLL 中输出的 CreateInstance 函数来建立相应的组件。

程序清单 5-3 此客户询问包含待建立的 DLL 的名称。然后它将装载此 DLL,在建立相应的组件之后使用组件中的接口。

```
CLIENT1.CPP
//
// Client1.cpp
// To compile, use: cl Client1.cpp Create.cpp GUIDs.cpp UUID.lib
```

```
//
#include <iostream.h>
#include <objbase.h>

#include "Iface.h"
#include "Create.h"

void trace(const char * msg) { cout << "Client 1: \t" << msg << endl ;}

//
// Client1
//
int main()
{
    HRESULT hr ;

    // Get the name of the component to use.
    char name[40] ;
    cout << "Enter the filename of a component to use [Cmpnt?.dll]: " ;
    cin  >> name ;
    cout << endl ;

    // Create component by calling the CreateInstance function in the DLL.
    trace("Get an IUnknown pointer.") ;
    IUnknown * pIUnknown = CallCreateInstance(name) ;
    if (pIUnknown == NULL)
    {
        trace("CallCreateInstance Failed.") ;
        return 1 ;
    }

    trace("Get interface IX.") ;
    IX * pIX = NULL ;
    hr = pIUnknown -> QueryInterface(IID_IX, (void * *)&pIX) ;

    if (SUCCEEDED(hr))
    {
        trace("Succeeded getting IX.") ;
        pIX -> Fx() ;           // Use interface IX.
        pIX -> Release() ;
    }
    else
    {
        trace("Could not get interface IX.") ;
    }

    trace("Release IUnknown interface.") ;
    pIUnknown -> Release() ;

    return 0 ;
}
```

程序清单5-4给出了实现组件的代码。除了组CreateInstance加上了extern "C"标记外，此组件基本上同前面关于此组件的代码是一样的。只不过现在它是在一个单独的文件CMPNT1.CPP中。在编译此文件时加上了/LD选项，并同程序清单5-1所示的CMPNT1.DEF一块链接起来了。

程序清单5-4 在一个单独的文件中所实现的组件。

```
CMPNT1.CPP
//
// Cmpnt1.cpp
// To compile, use: cl /LD Cmpnt1.cpp GUIDs.cpp UUID.lib Cmpnt1.def
//
#include <iostream.h>
#include <objbase.h>

#include "Iface.h"

void trace(const char* msg) { cout << "Component 1: \t" << msg << endl; }

//
// Component
//
class CA : public IX
{
    // IUnknown implementation
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
    virtual ULONG __stdcall AddRef();
    virtual ULONG __stdcall Release();

    // Interface IX implementation
    virtual void __stdcall Fx() { cout << "Fx" << endl; }

public:
    // Constructor
    CA() : m_cRef(0) {}

    // Destructor
    ~CA() { trace("Destroy self."); }

private:
    long m_cRef;
};

HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
{
    if (iid == IID_IUnknown)
    {
        trace("Return pointer to IUnknown.");
```

```
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IX)
    {
        trace("Return pointer to IX.");
        *ppv = static_cast<IX*>(this);
    }
    else
    {
        trace("Interface not supported.");
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}
ULONG __stdcall CA::AddRef()
{
    return InterlockedIncrement(&m_cRef);
}
ULONG __stdcall CA::Release()
{
    if (InterlockedDecrement(&m_cRef) == 0)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}
//
// Creation function
//
extern "C" IUnknown* CreateInstance()
{
    IUnknown* pI = static_cast<IX*>(new CA);
    pI->AddRef();
    return pI;
}
```

下面我们只需看一下两个共享的文件 IFACE.H 和 GUIDS.CPP。在 IFACE.H 中声明了客户和组件所使用的所有接口。

程序清单5-5 接口声明。

IFACE.H

```
//
// Iface.h
//
// Interfaces
interface IX : IUnknown
{
    virtual void __stdcall Fx() = 0;
};
interface IY : IUnknown
{
    virtual void __stdcall Fy() = 0;
};
interface IZ : IUnknown
{
    virtual void __stdcall Fz() = 0;
};
// Forward references for GUIDs
extern "C"
{
    extern const IID IID_IX;
    extern const IID IID_IY;
    extern const IID IID_IZ;
}
```

可以看到,接口和组件仍然使用了IX、IY和IZ接口。在IFACE.H的末尾声明了这些接口的ID。关于接口ID我们将在下一章中详细讨论。这些接口ID的定义在文件GUIDS.CPP中,如程序清单5-6所列。

程序清单5-6 在GUIDS.CPP中定义了接口ID。客户和组件将与此GUIDS.CPP链接起来。

GUIDS.CPP

```
//
// GUIDs.cpp - Interface IDs
//
#include <objbase.h>
extern "C"
{
    // {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
    extern const IID IID_IX =
        {0x32bb8320, 0xb41b, 0x11cf,
        {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

```
// {32bb8321-b41b-11cf-a6bb-0080c7b2d682}
extern const IID IID_IY =
    {0x32bb8321, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;
// {32bb8322-b41b-11cf-a6bb-0080c7b2d682}
extern const IID IID_IZ =
    {0x32bb8322, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;
// The extern is required to allocate memory for C++ constants.
```

上面就是如何在 DLL 中实现一个组件的所有细节。下面我们来看一下可以对这些组件进行一些什么有趣的处理。

5.3 对象串

下面我们将对一些组件进行处理并看一下它们是如何动态链接的。我们可以仿照 CLIENT1 实现另外两个客户 CLIENT2 和 CLIENT3，并仿照 CMPNT1 实现另外两个组件 CMPNT2 和 CMPNT3。在此客户 2 和组件 2 将支持接口 IX 和 IY，而客户 3 和组件 3 则支持所有的三个接口。表 5-1 列出了各接口及组件各自支持哪个接口。

表 5-1 各客户和组件所支持的接口

	IX	IY	IZ	
客户 1	√			组件 1
客户 2	√	√		组件 2
客户 3	√	√	√	组件 3

所有这些客户和组件可以使用如下的命令进行编译：

```
nmake -f makefile
```

在运行各客户程序时，它将首先询问待运行的组件的名称。输入一外组件名称之后按 Enter 键。然后客户将链接到指定的组件上，并查询它所知道的指定组件的每一个接口。若指定组件支持所查询的接口，客户将调用接口中的一个函数。若组件不支持所查询的接口，客户将会崩溃。

但与上述情形相反的是，客户并不会崩溃。相反，组件将显示出一条消息说明它不支持所请求的接口。下面是客户 2 连接到组件 2 及客户 3 连接到组件 1 的情形。

```
C:\ > client2
Enter the filename of a component to use [Cmpnt?.dll]: cmpnt2.dll
```

```
Client 2:   Get an IUnknown pointer.
Client 2:   Get interface IX.
Component 2:        Return pointer to IX.
Client 2:   Succeeded getting IX.
Fx
Client 2:   Ask for interface IY.
Component 2:        Return pointer to IY.
Client 2:   Succeeded getting IY.
Fy
Client 2:   Release IUnknown interface.
Component 2:        Destroy self.
c:\>client3
Enter the filename of a component to use [Cmpnt2.dll] cmpnt1.dll
```

```
Component 1:        Return pointer to IX.
Client 3:   Succeeded getting IX.
Fx
Client 3:   Ask for interface IY.
Component 1:        Interface not supported.
Client 3:   Could not get interface IY.
Client 3:   Ask for interface IZ.
Component 1:        Interface not supported.
Client 3:   Could not get interface IZ.
Client 3:   Release IUnknown interface.
Component 1:        Destroy self.
```

组件2实现了客户2所需的所有接口,而组件1只实现了接口IX。但客户3需要全部三个接口IX、IY和IZ。读者还可以自己试一下其他组合情形。

不知读者是否觉得这些例子比较有趣,实际上我们成功地建立了一种可以在运行时

第 6 章　关于 HRESULT、GUID、注册表及其他细节

莱特兄弟的精神今天仍然具有强大的生命力。每年都有成百上千的人用他们车库中的工具制造他们自己的飞机。这些飞机并不是那种塑料模型、无线电控制或织物制造的超轻型飞机，而是一种用高科技合成材料制造成的现代化的、双人全封闭的飞机。为满足联邦航空局(FAA)的一些要求，飞机装备制造商们只能完成飞机建造工作的 49%。其他 51%的工作仍要由飞机制造者自己完成。

根据所制造的飞机的规格的不同，制造者们需要花去 250 到 5000 小时的时间。大多数装备制造商都会提供一些已经装配好许多部件的装备，如已经完全焊接好的机架及其他构成部件。使用这些部件，制造者们用不着花太多的时间就可以使他们的飞机初具雏形。但要把这个雏形变成一架飞机还需要花许多的精力。例如，安装仪表板、仪表、座位、安全带、灭火器、地毯、保险丝、公告板、控制电缆、电线、灯光、电池、防火墙、通风孔、全压管、加热器、窗口、门闩、门把以及其他许多设备。

正因如此，许多机迷们在这个漫长的过程中逐渐由失望而沮丧，最终幻想破灭而放弃了努力。与此类似的是，许多人最初雄心勃勃地想要攻克一些比较困难的内容，如 COM，可由于一些细节问题而被弄得焦头烂额，最后不得不放弃。在本书的前五章中，作者尽了最大的努力将一些细节问题隐藏起来，并将我们的重点放在了对整体轮廓的讨论上。在这一章中我们将讨论一些前面被跳过或被隐藏的细节问题，同时还将讨论一些在学习下面的章节中需要了解的细节问题。

首先我们将讨论 HRESULT。关于它我们在第 3 章中讨论 QueryInterface 时曾有所提及。接下来讨论的是 GUID。传递给 QueryInterface 的 IID 结构就是 GUID 的一个例子。在讨论了这些类型之后，我们将讨论组件如何在 Windows 的注册表中发布其位置信息，以便客户能够找到并创建它们。最后我们将介绍 COM 库中的一些有用的函数及实用程序。

6.1　HRESULT

任何一架飞机都会有许多仪表，用配套元件制造的飞机也不例外。虽然某些飞机的仪表中可能使用了具有彩色图形显示器的计算机(它们可能是运行着 Windows NT!)，但大多数使用的可能是一些较为便宜一些的仪表。例如可以用一块金属片来作一个简单的空气速度指示器。金属片弯曲得越厉害，飞机飞得也就越快。

这些仪表可以告诉飞行员飞机及其各系统的运行情况，但其主要作用却是在某个地方出了问题时进行报警。例如在飞行速度指示器上有一个红色的表示速度过快或过慢的标记。其他各种仪表各有其警告指示灯或蜂鸣声。

但COM组件却没有仪表。相反，它们将使用HRESUTL来向其用户报告各种情况。例如QueryInterface返回的就是一个HRESULT值。在后面的章节中我们将会看到，许多COM接口函数所返回的均是HRESULT值。HRESULT这个名称容易使人将这种类型的值当成是一个指向某个结果的句柄，实际上并不是这样。HRESULT值实际上是一个可分成三个域的32位值。图6-1显示了这三个部分的含义。这个名称的形成是有其历史原因的，读者不要将它想成是"结果的句柄"(Handle to RESULT)，而应将其当成是"这就是结果"(Here's the RESULT)。

系统所生成的HRESULT值在Win32的头文件WINERROR.H中有定义。在此文件的前面列出的是Win32的错误代码。HRESULT值同Win32的错误代码有些类似，但读者不应将它们等同起来，也不要将其混淆。

图6-1所示的HRESULT值中最重要的比特位表示函数调用是否成功。这种设计使得函数可以返回多个成功代码及失败代码。HRESULT的低16位包含的就是函数的返回代码，其余的15位包含的是此类型及返回值起源的更详细的信息。

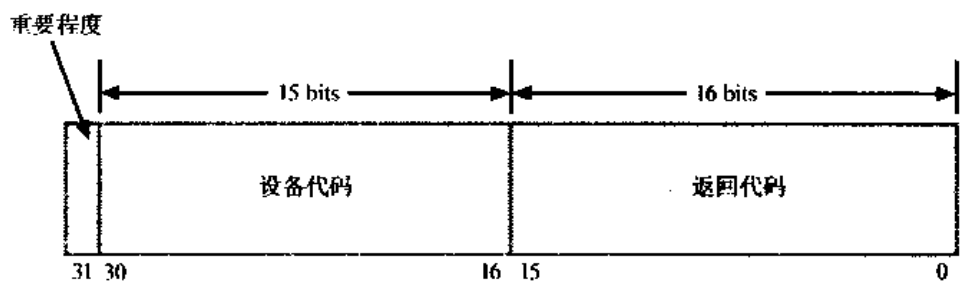


图6-1 HRESULT的结构

表6-1列出了那些最常用的错误代码。其中表示成功返回的代码在其名称前都有一个S_前缀，而表示失败返回的代码其名称前的前缀则是E_。

表6-1 常用HRESULT值

名称	含义
S_OK	函数成功。在某些情况下，它还表示函数返回了一个布尔真值。S_OK被定义为0。
NOERROR	同S_OK相同。
S_FALSE	函数成功并返回一个布尔假值。S_FALSE被定义为1。
E_UNEXPECTED	无法预知的失败。
E_NOIMPLE	成员函数未被实现。
E_NOINTERFACE	组件不支持所请求的接口。此值将由QueryInterface函数返回。

续表

名称	含　义
E_OUTOFMEMORY	组件无法分配所需的内存。
E_FAIL	没有指定的失败。

注意 S_FALSE 被定义成 1 而 S_OK 被定义成 0。这一点同 C/C++ 编程原则正好是相反的。在那里 0 表示 FAISE 而非 0 表示 TRUE。因此在使用 HRESULT 值时，应将返回的值直接同 S_FALSE 或 S_OK 进行比较。

HRESULT 值中 16 到 30 这 15 个比特位包含的是设备代码。设备代码标识的是可以返回 HRESULT 中返回代码的操作系统部分。由于 Windows 操作系统是由 Microsoft 开发的，因此它保留有定义设备代码的权力。当前所定义的设备代码如表 6-2 所列。

表 6-2　当前所定义的设备代码

FACILITY_WINDOWS	8
FACILITY_STORAGE	3
FACILITY_SSPI	9
FACILITY_RPC	1
FACILITY_WIN32	7
FACILITY_CONTROL	10
FACILITY_NULL	0
FACILITY_ITF	4
FACILITY_DISPATCH	2
FACILITY_CERT	11

使用设备代码，编写 RPC 代码的 Microsoft 开发人员将无需将他们用作返回代码的值同开发 AcitveX 控件的 Microsoft 开发人员进行协调。这样由于各不同的小组使用了不同的设备代码，因此从不同设备中返回的代码将不会出现冲突。编写定制接口的开发人员就不会这么幸运了。

所有的设备代码，除 FACILITY_ITFPPQH，都标识了一些由 COM 定义的返回代码。所有的设备代码都是通用的，它们对任何人、在任何地方都是相同的。只不过 FACILITY_ITF 是一个例外：它标识的是与返回此代码的接口特定相关的返回代码。为决定某个 HRESULT 值的设备，可以使用在 WINERROR.H 中定义的 HRESULT_FACILITY 宏。在“6.1.3 用户自己代码的定义”一节中，我们将会看到 HRESULT_FACILITY 宏返回的代码并不是唯一的，它们可以根据返回此代码的接口不同而具有不同的含义。在定义我们自己的返回代码之前，下面我们先来看一下如何使用 HRESULT 值。

6.1.1 HRESULT 值的查找

前面曾提到过，WINERROR.H 中包含当前由系统产生的所有 COM(及 OLE，现在被称

作是 AcitiveX)状态代码。大多数这种代码是用十六进制的形式给出的。如 E_NOINTERFACE 的定义如下：

```
//
// MessageId: E_NOINTERFACE
//
// MessageText:
//
//  No such interface supported
//
#define E_NOINTERFACE                    0x80004002L
```

但是若某个具有 FACILITY_WIN32 设备代码 HRESULT 值，我们将会发现将它不在 HRESULT 值列表中。通常它将是一个被映射成 HRESULT 值的 Win32 错误代码。为查明其含义，可查找低 16 位与之相同的 Win32 错误代码。例如某个接口可能返回值 0x80070103。其中的 7 就是设备代码 FACILITY_WIN32。当在 WINERROR.H 中查找时，将会发现并没有被列在 HRESULT 值中。为查明其含义，可将其低 16 位从十六进制数转换成十进制数 259，然后我们将可以在错误代码列表中找到它。

```
//
// MessageId: ERROR_NO_MORE_ITEMS
//
// MessageText:
//
//  No more data is available.
//
#define ERROR_NO_MORE_ITEMS              259L
```

在程序开发过程中在 WINERROR.H 中查找 HRESULT 值的含义是可以的。但在某些情况下，应用程序需要一种获取与某个特定 HRESULT 值相应的错误消息并将其显示出来的方法。以显示标准 COM(以及 AcitveX，其前身为 OLE 和 Win32)错误消息，可以使用 Win32 API 中的 FormatMessage 函数：

```
void ErrorMessage(LPCTSTR str, HRESULT hr)
{
  void* pMsgBuf;
  ::FormatMessage(
      FORMAT_MESSAGE_ALLOCATE_BUFFER |
      FORMAT_MESSAGE_FROM_SYSTEM,
      NULL,
      hr,
      MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
      (LPTSTR) &pMsgBuf,
```

```
        0,
        NULL);
    // Display the string.
    cout << str << "\r\n";
    cout << "Error (" << hex << hr << "): "
        << (LPTSTR)pMsgBuf << endl;
    // Free the buffer.
    LocalFree( pMsgBuf );
}
```

6.1.2 HRESULT 值的使用

读者将会发现使用 HRESULT 值比使用典型的布尔值要稍微麻烦一些。如果对下面的一些问题不注意,可能会导致"阴沟里翻船"。

- 成功的代码有多个,失败的代码也有多个。
- 失败代码可能会发生变化。

一、多状态代码

一个函数在各种情况下返回的状态代码通常将包含多个成功代码及多个失败代码。这就是我们为什么要使用 SUCCEEDED 及 FAILED 宏的原因。一般不能直接将 HRESULT 值同某个成功代码(如 S_OK)进行比较以决定某个函数是否成功,也不能直接将其同某个失败代码(如 E_FAIL)进行比较以决定函数调用是否失败。因此下面的代码是有问题的:

```
HRESULT hr = CoCreateInstance(...);
if (hr == E_FAILED) // Don't do this!
  return;
hr =  pI->QueryInterface(...);
if (hr == S_OK)    // Don't do this!
{
  pIX->Fx();
  pIX->Release();
}
pI->Release();
```

为纠正上述代码中的错误,可以不将 HRESULT 值同 S_OK 或 E_FAIL 值直接进行比较,而可使用 SECCEEDED 及 FAILED 宏。

```
HRESULT hr = CoCreateInstance(...);
if (FAILED(hr))
  return;
hr =  pI->QueryInterface(...);
```

```
if (SUCCEEDED(hr))
{
  pIX->Fx();
  pIX->Release();
}
pI->Release();
```

二、错误可能会发生变化

在编写好客户程序之后，其他人可能会定义新的 HRESULT 错误代码，客户程序也可能会遇到相应的错误。由于客户使用的组件可能会发生变化，因此组件返回的错误代码也可能会发生变化。假定我们将一个组件当成是一个进程中服务器来实现，然后对之进行升级以使之变成在另外一个系统上运行的远程服务器。显然，组件的第一个版本将不会返回任何网络错误代码，但第二个版本则可能会。由于客户可能并不知道它所能遇到的所有错误代码，因此它应该能够处理那些无法预知的错误代码。为此可在程序中像处理 E_UNEXPECTED 那样处理所有的不可预知的错误。

成功代码则没有这些，这是因为函数返回的成功代码必须是静态的，它们将被当成是接口的一部分，因此不能发生变化。使用某个接口的客户必须能够理解、预知并处理所有可能的成功返回值，这样才能在各种情况下都可以继续运行。但客户并不需要处理所有可能的错误代码，这是因为它并不需要在遇到某个无法预料的错误代码之后继续运行。

HRESULT 和网络

同远程机器的连接经常莫名其妙地就断开了。对于同远程组件进行通信的客户，它必须能够妥善地处理网络连接的中断。这意味着任何可能会跨越网络的函数调用都应该具有指示网络连接中断的方法。由于这个原因，所有期望能够在远程机器上运行的成员都应返回一个 HRESULT 值。为此需要避免返回其他类型的值，如对于：

```
double GetCordLength(double BladeSection);
```

可将其修改为返回一个 HRESULT 值而将其他结果在输出参数中返回：

```
HRESULT GetCordLength(/*in*/ double BladeSection,
                      /*out*/double* pLength);
```

返回的 HRESULT 值将给客户提供检查网络错误所需的信息。自动化(以前称作是 OLE 自动化)中的函数调用将可以满足这种需求。关于远程组件，我们将在第 10 章中更详细地讨论。

6.1.3 用户自己代码的定义

所有通用的返回代码都是由 COM 定义的，如 S_OK 和 E_UNEXPECTED。与接口特定

相关的错误代码的定义则由接口设计人员负责。包含特定于某接口返回代码的 HRESULT 值中也应包含 FACILITY_ITF 设备代码。此设备代码实际上是告诉客户告诉此返回代码是与特定的接口相关的。

虽然用 FACILITY_ITF 标记的返回代码的含义是同返回它的接口相关的，但代码值本身一般而言并不是唯一的，因为只有 2^{16} 个可能的返回代码，而有成千上万的开发人员在开发定义有他们自己返回代码的 COM 组件。所有这些错误代码均被标记为 FACILITY_ITF。实际上 AcitveX(OLE)定义有许多接口，而所有这些接口都定义了用 FACILITY_IFT 标记的返回代码。因此不同的接口将不同的含义赋予同一值不仅是可能的，而且几乎可以肯定会出现这种情况。虽然 32 个比特所提供的空间并不足以使每个开发人员都能够拥有其自己的设备代码，但使用更大的位串将会导致性能的下降。使用 GUID 作为返回代码并不是一种可行的替代方法，因 GUID 结构实在是太大了。但是由于 FACILITY_ITF 标明了返回的错误代码是特定于某个接口的，因此返回代码实际上是同接口的 IID 关联起来了。

对于调用某个接口的客户，返回值出现冲突的可能性并不会成为一个问题。由于客户知道它所调用的接口，因此它也就知道此接口可能会返回的所有成功代码。并且客户也将知道大多数失败代码。对于那些客户不能识别的失败代码，都应将其当成是 E_UNEXPECTED 那样进行处理。但是，当某个接口的调用者本身是一个组件并试图将所得到的成功或失败返回值传播给其客户时，就可能会出现问题了。这是由于调用者的客户无法理解这个返回值，因它最初是由客户并不知道的那个客户返回的。

例如假定最初客户 A 调用 IX::Fx，在 IX::Fx 中调用了 IY::Fy。若 IY::Fy 返回一个具有 FACILITY_ITF 设备代码的 HRESULT 值，IX::Fx 不能将此值传递给客户 A。这是因为客户 A 仅知道接口 IX，因此它将认为所得到的 HRESULT 值是特定于 IX 而非 IY 的。因此 IX::Fx 必须将所有 IY 返回的具有 FACILITY_ITF 设备代码的值转换成客户 A 能够识别的 HRESULT 值。对于不知道的错误代码，IX 将只能返回 E_UNEXPECTED。对于成功代码，IX 必须返回写在文档中的 IX 返回代码。

下面给出关于定义自己的 HRESULT 的一些一般性规则：

- 不要将 0x0000 及 0x01FF 范围内的值作为返回代码。这些值是为 COM 所定义的 FACILITY_ITF 代码而保留的。只有遵循这一规则，才不致使用户自己定义的代码同 COM 所定义的代码相混淆。
- 不要传播 FACILITY_ITF 错误代码。
- 尽可能地使用通用的 COM 成功及失败代码。
- 避免定义自己的 HRESULT，而可以在函数中使用一个输出参数。

在我们明白了 HRESULT 值的一些含义之后，可以使用 MAKE_HRESULT 宏来定义一个 HRESULT 值。此宏可以根据所提供的严重级别、设备代码及返回代码生成一个 HRESULT 值。如下面的两例所示：

```
MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF, 100);
MAKE_HRESULT(SEVERITY_SUCCESS, FACILITY_ITF, 101);
```

对于自己定义的返回代码的命名,有一个约定是在其名称前面加上组件或接口的名称。例如可以将上面所定义的 HRESULT 值命名如下:

```
AIRPLANE_E_LANDINGWITHGEARUP
HELICOPTER_S_ROTORRPMGREEN
```

关于 HRESULT,了解这么多应该就足够了。下面我们来看一下 GUID。

6.2 GUID

在美国,所有的常规飞机都要由 FAA 用一个数进行登记。正如汽车的执照号码是标识汽车的一样,这个数也是标识各飞机的。飞行员同空中交管部门联系时也将使用这一号码。在这一章中我们将讨论 GUID,它实际上是组件和接口的标识号。

在第 3 章中,作者曾告诉读者可以将 IID 当成是一个标识接口的常量。但从 IID_IX 的定义中可以看出,IID 实际上是一种特殊类型的常量:

```
extern "C" const IID IID_IX =
{0x32bb8320, 0xb41b, 0x11cf,
{0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}};
```

实际上 IID 是一个 128 比特(16 字节)的一个 GUID 结构。GUID 是英文 Globally Unique Identifier(全局唯一标识符)的首字母缩写,其发音同单词 geoduck① 的前一半及 druid 的后一半类似——有点像是 goo-id。

6.2.1 为什么要使用 GUID

为什么要使用 GUID 来标识接口而不用一个长整数呢?一个长整数不也是可以标识多达 2^{32} 个接口吗?作者甚至对是否需要多于 2^{32} 个接口表示严重的怀疑。但问题的实质并不在于我们可以标识多少个接口,而在于怎样保证每一个接口标识符都是唯一的。若两个接口具有相同的标识符,那么客户在调用 QueryInterface 时将可能会得到错误的接口指针。由于编写 COM 组件的开发人员是遍布全世界的,因此这个问题将更加复杂。例如若 Ako 的 Sarah 和 Tucson 的 Lynne 都在开发一个新的组件,那么如何保证他们所定义的接口标识符不会出现冲突呢?对于这种情况,一种可能的解决方法是像美国所有飞机的标识号是由 FAA 来分配的那样,由一个中央授权机构来管理所有接口的 ID 号。但与 FAA 所管理的数目相对而言有一定限度的飞机标识号相比,用一个中央软件授权机构来管理

① 通常许多人将其读成 gooey duck。程序员一般将它念成"GUI duck"。

所有程序所需的接口标识号几乎是一件不可能的事情。

GUID 则提供了一种更好的解决方案。在不同任何中央授权机构进行协调的情况下，可以用编程的方法来生成具有唯一性的 GUID。Microsoft Visual C++ 提供了两个建立 GUID 的程序，一个是 UUIDGEN.EXE，该程序是命令行方式的；另一个则是 GUIDGEN.EXE，是一个示例性的 VC++ 对话框应用。例如在运行 UUIDGEN.EXE 时将可能会得到如下的表示 GUID 的串：

```
{166769E1-88E8-11cf-A6BB-0080C7B2D682}
```

每次运行 UUIDGEN 时，都将得到一个不同的 GUID。当读者在自己的机器上运行此程序时，所得到的 GUID 肯定与上面给出的不同。若本书的所有读者都使用这个程序来生成一个 GUID，他们所得到的 GUID 将各不相同。

Microsoft Visual C++ 提供有 GUIDGEN.EXE 的源文件。它实际上调用了 Microsoft COM 库中的 CoCreateGuid 函数，CoCreateGuid 将调用 RPC 函数 UuidCreate。

关于 GUID 的一些背景

根据定义，GUID 在时间和空间上都是唯一的。为保证空间唯一性，每一个 GUID 需要一个对于生成它的计算机来说是单一的 48 位值。这个值一般就是机器上所装网卡的地址。使用这个值可以保证在作者的机器上所生成的 GUID 值与在读者机器上所生成的 GUID 值是不相同的。对于没有安装网卡的计算机，此单一值的生成将使用其他算法来完成。为保证时间上的唯一性，每一个 GUID 值具有一个 60 位的时间戳。这个时间戳表示的是自 1852 年 10 月 15 号 00:00:00.00 以来以 100 纳秒为间隔的记数值。目前用以生成 GUID 的算法将可以保证至大约公元 3400 年 GUID 值仍然是唯一的。（实际上现在所开发的程序，除了某些 FORTRAN 程序之外，很少有能在公元 3400 年还在使用的。但到那时 Windows 2000 肯定已经推出了。）

GUID 的概念最初是由开放软件基金会（OSF）的专家们定义的，只不过在那里他们称通用唯一标识符罢了（UUID）。UUID 是作为分布式计算环境（DCE）的一部分而定义的。DEC 的远程过程调用（RPC）将使用 UUID 来标识被调用者。这一点同我们使用 GUID 的目的是类似的。

有关生成 UUID 或 GUID 算法的详细讨论，请参阅 CAE Specification X/Open DEC: Remote Procedure Call。

6.2.2 GUID 的声明和定义

由于 GUID 结构比较大，因此一般我们不希望将其在程序中到处复制。在第 5 章中，在文件 GUIDS.CPP 中用类似如下的代码定义了一些 GUID：

```
// {32bb8320-b41b-11cf-a6bb-0080c7b2d682}
```

```
extern "C" const IID IID_IX =
    {32bb8320, 0xb41b, 0x11cf,
    {0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82}} ;
```

在文件 IFACE.H 中它们被按如下的方式声明：

```
extern "C" const IID IID_IX ;
```

当然用两个文件来维护 GUID 是一件比较痛苦的事情。若希望用一条语句来声明并定义 GUID，可以使用 OBJBASE.H 中所定义的 DEFINE_GUID 宏。为使用此宏，可以使用 GUIDGEN.EXE 来生成一个 GUID。GUIDGEN.EXE 可以按几种不同的格式来生成 GUID，读者可选择其中的第二种格式，下面的例子就是用这种格式给出的。

```
// {32bb8320-0xb41b-11cf-A6BB-0080C7B2D682}
DEFINE_GUID( << name >> ,
    0x32bb8320, 0xb41b, 0x11cf,
    0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82) ;
```

将所生成的 GUID 粘贴到一个头文件中，然后用实际的名称——例如 IID_IX——来取代 << name >> ：

```
DEFINE_GUID(IID_IX,
    0x32bb8320, 0xb41b, 0x11cf,
    0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82) ;
```

在 OBJBASE.H 中所定义的 DEFINE_GUID 将产生与下类似的输出：

```
extern "C" const GUID IID_IX
```

但是，如果在 OBJBASE.H 之后包含 INITGUID.H，那么 DEFINE_GUID 所生成的代码将与下面的类似：

```
extern "C" const GUID IID_IX =
    { 0x32bb8320, 0xb41b, 0x11cf,
    { 0xa6, 0xbb, 0x0, 0x80, 0xc7, 0xb2, 0xd6, 0x82 } } ;
```

从图 6-2 中可以看出这一点是如何工作的。在头文件 IFACE.H 中使用了 DEFINE_END 来声明 IID_IX，而在 GUIDS.CPP 中则定义了标识符 IID_IX。之所以会这样是因为在文件 OBJBASE.H 和 IFACE.H 之间包含了头文件 INITGUID.H。另一方面在 CMPNT.CPP 中只是声明了 IID_IX 而并没有定义它，这是因为在 CMPNT.CPP 中并没有包含 INITGUID.CPP。

为使本书中的例子尽可能地清晰，作者将不使用宏 DEFINE_GUID，而是明确地声明和定义所用到的 GUID。

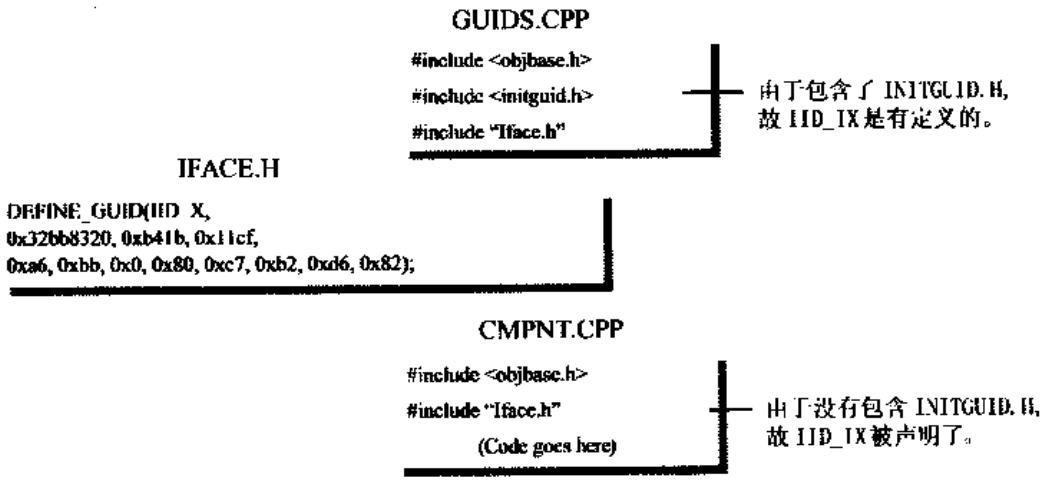


图 6-2 由于包含 INITGUID.H 使 DEFINE_GUID 宏定义所需的 GUID

6.2.3 GUID 的比较

为方便 GUID 值的比较,在 OBJBASE.H 中定义了一个 == 操作符。

```
inline BOOL operator==(const GUID& guid1, const GUID& guid2)
{
    return ! memcmp(&guid1,&guid2,sizeof(GUID));
}
```

在 QueryInterface 的实现中我们已经使用了此操作符。若不喜欢用操作符,则可以使用在 OBJBASE.H 中所定义的等价的函数 IsEqualGUID、ISEqualIID 以及 IsEqualCLSID。

6.2.4 将 GUID 作为组件标识符

除了可以用 GUID 来唯一地标识接口之外,还可以使用 GUID 来唯一地标识组件。在第 5 章中,我们定义了一个创建组件的 CallCreateInstance 函数。此函数将接收一个实现组件的 DLL 名称串作为参数。

```
IUnknown * CallCreateInstance(char * name);
```

在下一章中我们将用一个名为 CoCreateInstance 的 COM 库函数来代替 CallCreateInstance。CoCreateInstance 将使用一个 GUID 而不是一个串来标识组件。在 COM 中用以标识组件的 GUID 被称作是类标识符。为将类标识符同 IID 区别开来,与类标识符相应的类型为 CLSID。

同接口一样,所有的组件都有一个不同的 ID。两个组件可以实现相同的接口集,但每一个组件仍然必须具有各不相同的 CLSID。在不改变类标识符的情况下,可以向组件

中加入新的接口。但是若对某个组件的修改将会妨碍已有应用程序的正常运行，那么必须为组件分配一个新的 CLSID。

6.2.5 通过引用传递 GUID 值

由于一个 GUID 值占用了 16 个字节，因此一般不用值传递 GUID 参数，而大量使用的是按引用传递，这就是为什么 QueryInterface 接收一个常量引用参数的原因。若读者不想输入 const IID&，则可以使用等价的 REFIID。类似地，在传递类标识符时，可以使用 REFCLSID；而在传递 GUID 值时可以使用 REFGUID。

下面我们来看一下组件如何在系统中注册它们以便客户可以找到并使用它。

6.3 Windows 注册表

FAA 维护了一个所有常规飞机的注册表。在其中记录了哪架飞机是属于谁的。在这一节中我们将介绍另外一种类型的注册表，在其中可以记录在哪个 DLL 中包含哪个组件。

在第 5 章中，当创建一个组件时，我们需要将包含有待创建组件的 DLL 的名称传递给 CallCreateInstance。在下一章中，我们将用 COM 库函数 CoCreateInstance 来取代 CallCreateInstance。此函数将不再使用包含组件的 DLL 名称作为参数，而是使用了一个 CLSID。组件可以用 CLSID 作为索引在 Windows 的注册表中发布包含它们的 DLL 文件名称。CoCreateInstance 将用 CLSID 作为关键字在注册表中查找所需的文件名称。

现实生活中的注册表是一种记录有某些项、名称或动作的正式本子。在 Windows 中，注册表是 Windows 操作系统的一个正式的共享系统数据库。注册表中包含关于系统中硬件、软件、配置以及用户的各种信息。所有基于 Windows 的程序都可以向注册表中加入若干信息并可从中读出所需的信息。客户可以在注册表中搜索它们所需的组件。在介绍如何在注册表中发布信息之前，首先我们讨论一下注册表是如何组织的。

6.3.1 注册表的组织

注册表是一个由许多元素构成的层次结构。每一个元素均被称作是一个关键字。每一个关键字可以包含一系列子关键字、一系列命名的值及/或一个未命名的值(缺省值)。子关键字也可以包含其他的子关键字及值，但值却不能有子关键字或值。每一个值都有其相应的类型，大多数情况下，关键字值都是字符串。图 6-3 显示了注册表的结构。

6.3.2 注册表编辑器

注册表中充斥着关于程序的信息。但幸运的是，一般我们只是对注册表信息的一个子集感兴趣。学习注册表的最好的方法是使用注册表编辑器，使用它可以浏览和编辑注

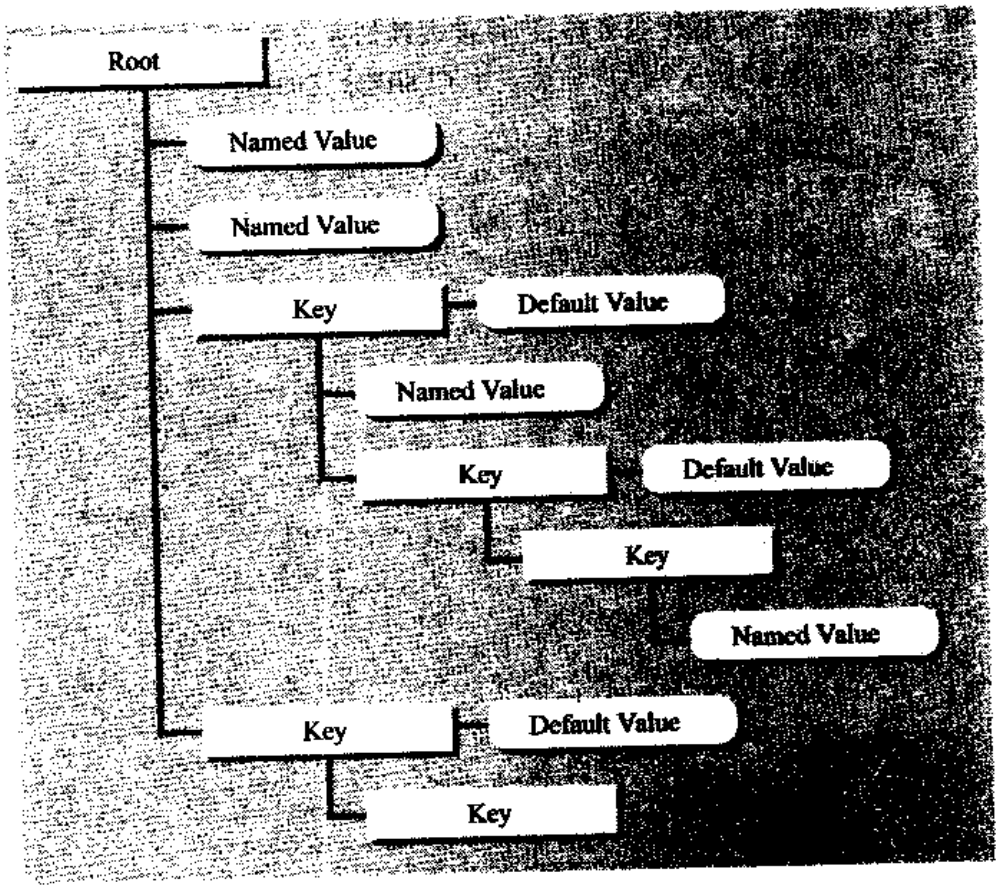


图 6-3 Windows 注册表的结构

册表中的内容。在 Windows NT 中,注册表编辑器的名称为 REGEDIT32.EXE;在 Windows 95 中的名字则为 REGEDIT.EXE。另外要提醒的是:修改编辑器中的内容极易将整个系统搞崩溃,因此应多加小心。

下面我们来看一下注册表中最重要的信息:CLSID 及 DLL 的文件名称。之后我们将看一下注册表中保存的其他信息。

6.3.3 CLSID 关键字结构

COM 只使用了注册表的一个分支:HKEY_CLASSES_ROOT。在此关键字之下,可以看到有一个 CLSID 关键字。在 CLSID 关键字之下列有系统中安装的所有组件的 CLSID。注册表 CLSID 是一个具有如下格式的串:{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}。很快我们将会看到,按 CLSID 来查看注册表将是一件极不轻松的事。这就是为什么每一个 CLSID 关键字都用组件的有意义的名称来作为其缺省值。

对于每一个 CLSID 关键字,我们现在关心的只是它的一个子关键字 InprocServer32。此子关键字的缺省值是组件所在的 DLL 文件名称。使用 InprocServer32 这个名称是比较

合适的,因 DLL 就是一个进程中服务器;它将被装载到客户的进程空间中,并给客户提供所需的服务。图 6-4 显示了一个示例注册表中 CLSID 分支的结构。

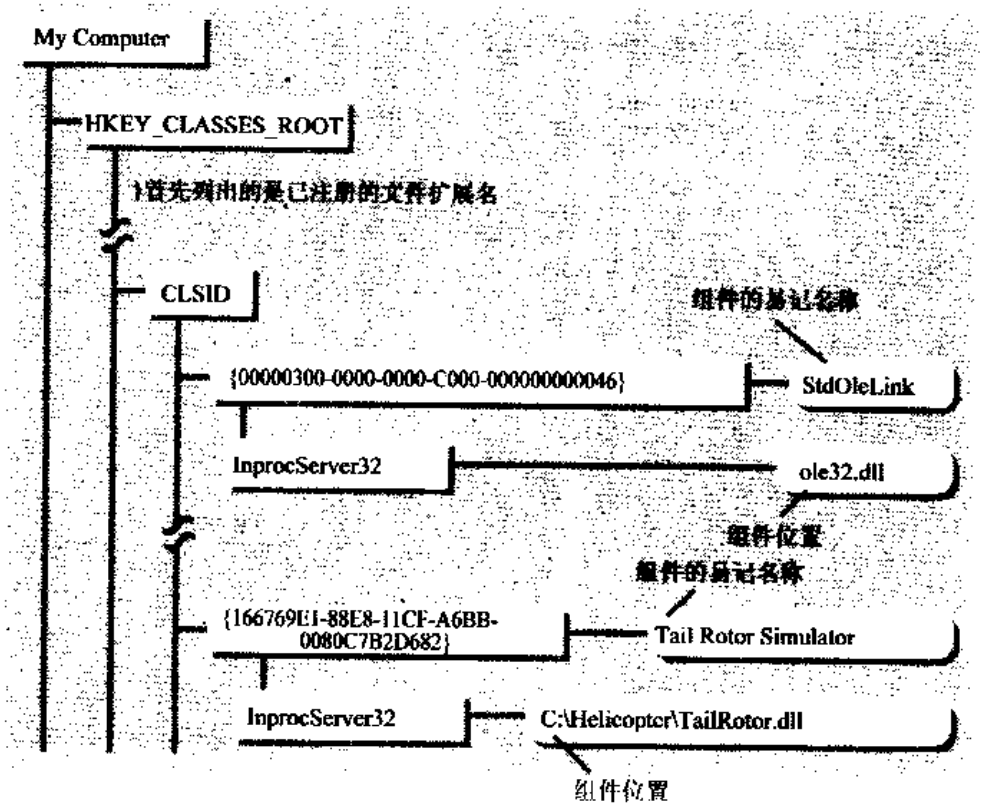


图 6-4 注册表中 CLSID 子关键字的结构

从图 6-4 中可以看出,Tail Rotor Simulator 组件的 CLSID 被保存在 HKEY_CLASSES_ROOT\CLSID 关键字下。组件的这个有意义的名称被注册为组件的 CLSID 缺省值。在此 CLSID 下的 InprocServer32 子关键字包含实现 Tail Rotor Simulator 组件的 DLL 名称 C:\Helicopter\TailRotor.dll。

CLSID 和文件名称是注册表中记录的两种最重要的信息。开发人员可以在不向注册表中加入任何有关组件信息的情况下编写出大量的组件。但是在某些情况下将需要用到注册表中更详细的信息。

6.3.4 关于注册表的其他细节

下面我们简要地看一下 HKEY_CLASSES_ROOT 子关键字的结构。前面我们已经介绍了 CLSID,在本章的后面我们将看到在此关键字中可以为类保存更多的信息。在 KKEY_CLASSES_ROOT 的开头,列出的将是各种应用程序所注册的文件扩展名。在扩展名之后,可以看到许多其他的名字。此类名字的大多数被称作是 ProgID,表示是程序员定

义的标识符。稍后我们将讨论 ProgID。某些名称表示的不是 ProgID 而是一些特殊的关键字，如 CLSID 关键字。这些关键字可以将一个 GUID 映射为其他某块信息，如一个文件名称。如下所列均为一些特殊的关键字。

- AppID——此关键字下的子关键字的作用是将某个 APPID(应用程序 ID)映射成某个远程服务器名称。分布式 COM(DCOM)将用到此关键字。关于此关键字我们将在第 10 章中讨论。
- 组件类别——注册表的这一分支可以将 CATID(组件类别 ID)映射成某个特定的组件类别。关于组件类别，我们将在“6.3.7 组件类别”中讨论。
- Interface——此关键字用于将 IID 映射成与某个接口相关的信息。这些信息主要用于在跨越进程边界使用接口的情况。关于此关键字我们将在第 10 章中讨论。
- Licenses——Licenses 保存的是授权使用 COM 组件的一些许可信息。本书中我们将不讨论关于许可的问题。
- TypeLib——类型库关键字所保存的是关于接口成员函数所用参数的信息，另外还有其他一些信息。此关键字可以将一个 LIBID 映射成存储类型库的文件名称。关于类型库我们将在第 11 章中讨论。

6.3.5 ProgID

下面我们来详细讨论一下 ProgID。

注册表中 HKEY_CLASSES_ROOT 分支的大多数子关键字都是 ProgID。所谓 ProgID 指的是程序员给某个 CLSID 指定的一个程序员易记的名称。某些计算机语言，如 Visual Basic，将用 ProgID 而不是 CLSID 来标识组件。但没有办法来保证 progID 是唯一的，因此名字冲突将是一个潜在的问题。但对 ProgID 处理起来却容易得多。(由于某些编程语言并不能处理结构，因此它们需要使用字符串格式的 CLSID。)

一、ProgID 命名约定

根据约定，ProgID 具有如下的格式：

```
<Program>.<Component>.<Version>
```

例如在作者机器上的注册表中有如下一些 ProgID：

```
Visio.Application.3
Visio.Drawing.4
RealAudio.RealAudioõ ActiveX Control (32-bit).1
Office.Binder.95
MSDEV.APPLICATION
JuiceComponent.RareCat.1
```

上述格式只是一种约定而不是什么规则。在作者机器上的注册表中就有许多并不遵

循上述约定的 ProgID。

在许多情况下，客户并不关心它所连接的组件的版本究竟是什么。因此组件经常会有一个与版本号无关的 ProgID。此 ProgID 被映射成所安装的最新版本的组件。与版本无关的 ProgID 的命名约定是将上述约定中的版本号去掉。例如 MSDEV.APPLICATION 就是一个与版本号无关的 ProgID。

二、ProgID 注册表格式

ProgID 及与版本无关的 ProgID 被列在组件 CLSID 下面。但是 ProgID 的主要作用是获取相应的 CLSID。在每一个 CLSID 项中查找某个 ProgID 将是非常低效的。因此在 HKEY_CLASSES_ROOT 的下面也将直接列出 ProgID。由于 ProgID 并不是针对最终用户而定义的，因此 ProgID 关键字的缺省值为用户易记的名称。在 ProgID 之下有一个名为 CLSID 的关键字，其缺省值为组件的 CLSID。与版本号无关的 ProgID 也直接被列在 HKEY_CLASSES_ROOT 下面。它们还将有另外一个关键字 CurVer，其缺省值为组件当前版本的 ProgID。

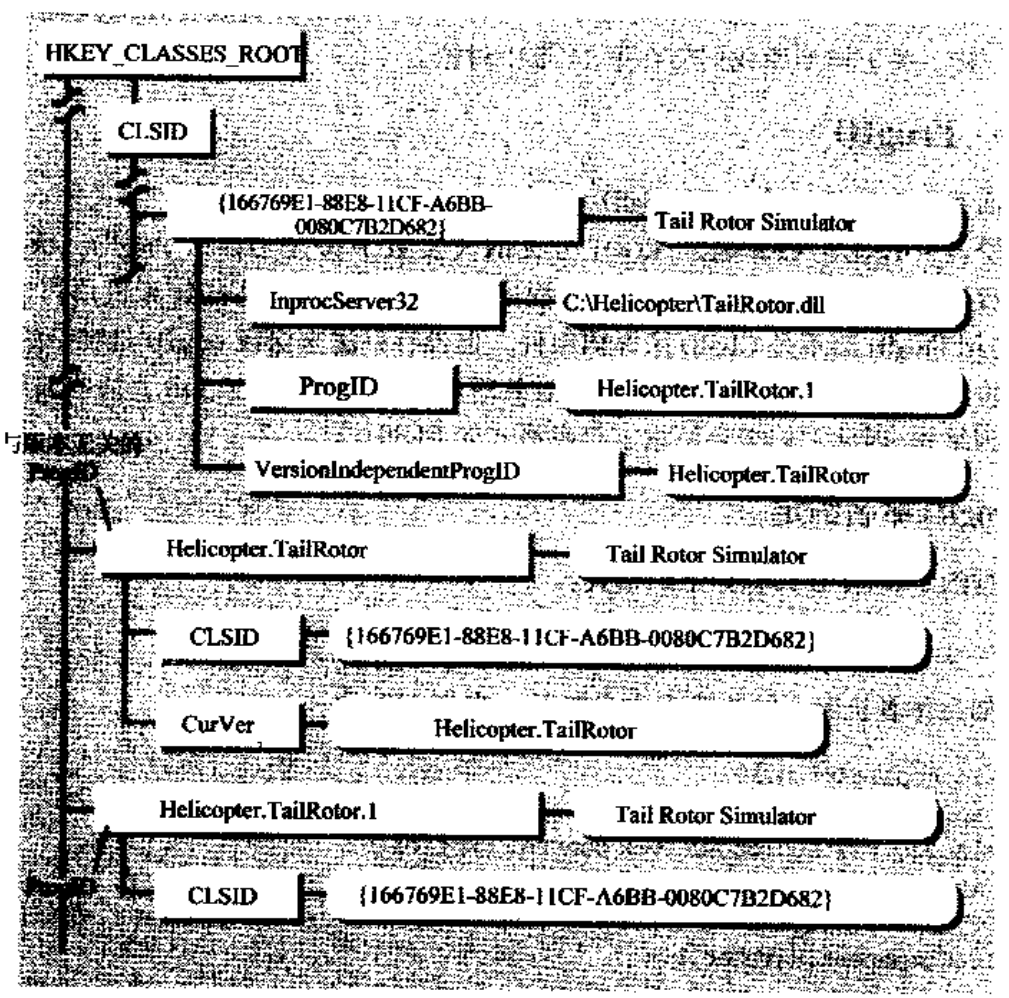


图 6-5 包含与 ProgID 相关信息的登记簿关键字的组织情况

图 6-5 所示是对图 6-4 所示的 Tail Rotor Simulator 一例的扩展，在此加上了一个 ProgID。在此组件的 CLSID 关键字下加上了一个名为 ProgID 的关键字，其值为 Helicopter.TailRotor.1，也就是组件的 ProgID。与版本号无关的 ProgID 被保存在名为 VersionIndependentProgID 的关键字中。本例中此关键字的值为 Helicopter.TailRotor。

另外该例也显示了名为 Helicopter.TailRotor.1 以及 Helicopter.TailRotor 的关键字直接挂在了 HKEY_CLASSES_ROOT 关键字之下。Helicopter.TailRotor.1 包含一个子关键字 CLSID，其中包含相应组件的 CLSID。与版本号无关的 ProgID 关键字 Helicopter.TailRotor 包含两个关键字 CLSID 及 CurVer。其中 CurVer 子关键字的缺省值为相应组件当前版本的 ProgID，即 Helicopter.TailRotor.1。

三、从 ProgID 到 CLSID 的转换

在将所需的信息注册到注册表中之后，进行 CLSID 与 ProgID 之间的转换将是非常容易的。COM 库为此提供了两个函数 CLSIDFromProgID 和 ProgIDFromCLSID，以完成所需的注册表处理：

```
CLSID clsid ;
CLSIDFromProgID(L"Helicopter.TailRotor", &clsid) ;
```

6.3.6 自注册

有了注册表之后，又如何在其中发布关于组件的信息呢？由于 DLL 知道它所包含的组件，因此 DLL 可以完成这些信息的注册。但由于 DLL 本身并不能完成任何事情，因此在 DLL 中一定要输出如下两个函数：

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

其中 STDAPI 在 OBJBASE.H 中被定义如下：

```
#define STDAPI EXTERN_C HRESULT STDAPICALLTYPE
```

展开后它将是：

```
extern "C" HRESULT _stdcall
```

用户可以使用程序 REGSVR32.EXE 来注册某个组件，它实际上是通过调用上述那些函数来完成组件的注册的。用户的系统中一般都带有这一程序。在本书后面许多章节的例子中，相应的 makefile 都将调用这一程序而完成组件的注册。

作为安装过程的一部分，大多数安装程序也将调用 DllRegisterServer。这一过程实际上是非常简单的，只需用 LoadLibrary 装载相应的 DLL，然后用 GetProcaddress 获取此函数的地址，再调用此函数即可。

一、DllRegisterServer的实现

DllRegisterServer的实现实际上是通过直接调用注册表函数来完成的。Win32 API中包含许多向注册簿中加入新的关键字及删除某个关键字的函数。为注册某个组件或取消某个组件的注册，只需要用到其中的6个函数，如下所列：

```
RegOpenKeyEx
RegCreateKeyEx
RegSetValueEx
RegEnumKeyEx
RegDeleteKey
RegCloseKey
```

在其他的书籍中对这些函数的介绍已经非常详尽了，故在此作者不准备对其做更多的讨论。为使用这些函数，需要在源文件中包含WINREG.H或WINDOWS.H头文件，并将其同ADVAPI32.LIB一块链接。在下一章给出的例子中的REGISTRY.H及REGISTRY.CPP中，读者将可以看到这些函数是如何被使用的。

6.3.7 组件类别

在Windows的注册表中包含一个很长的CLSID列表，其中每一个CLSID都有一个相关的文件名称。客户可以自由地查询这些CLSID并选择所需的组件来完成相应的任务。这样就产生一个问题：客户根据什么来决定选择使用哪个组件呢？一种方法是向用户显示出每一个组件的有意义的名称，然后由用户加以选择。但这样有一个问题就是客户只应显示出那些它实际上能够使用的组件的名称。因为如果用户从一个很长的组件名称列表中选择出某个之后客户实际上却无法使用，那么对用户而言，这将是非常难受的。另外有一种方法就是依次装载注册表中的每一个组件，然后向其查询所需的接口。但可以想见这种方法的开销将非常大。开发人员需要的是一种无需创建组件实例就能决定它是否提供所需接口的方法。

组件类别就是解决这个问题的一种方案。一个组件类别实际上就是一个接口集合。该集合将被分配给一个GUID，此GUID此时将被称作是CATID。对于某个组件而言，若它实现了某个组件类别的所有接口，那么它就可以将其注册成该组件类别的一个成员。这样，客户就能够通过从注册表中选择只属于某个特定组件类别的组件而准确找到它所需的组件。

组件类别也是组件和客户之后的一个合约。对于将自己注册为某个组件类别的组件，它将保证它可以支持此组件类别的所有成员接口。因此可以使用组件类别对组件加以分类。组件类别的使用方法同C++中的抽象基类的用户是类似的。一个抽象基类是一组其派生类所必须实现的函数的集合，所以可以说此派生类是相应抽象基类的一个特

定实现。与此类似,一个组件类别是属于此类别的组件所必须实现的接口集合。属于某组件类别的组件是该组件类别的一个特定实现。

对一组件而言,它可以是任意多个组件类别的成员。属于某个组件类别并不是说此组件只能实现该组件类别中的那些接口。实际上它可以实现任何其他所需的接口。

组件类别的一种用途是指定某个组件必须实现的接口集合。另外一种用途是用于指定组件需要其客户提供的接口集合。这是由于某些组件需要客户提供某些服务才能正常运行。例如一个三维图形对象在其能够运行之前可能需要某个特定的图形引擎。

一、组件类别的实现

组件类别最吸引人的地方可能是开发人员无需自己完成注册表的处理。这些工作可以由 Windows 系统附带的一个名为 Component Category Manager 来完成。Comomponent Category Manager,CLSID_StdComponentCategoriesMgr 是一个实现了两个主要接口的 COM 组件:ICatRegister及 ICatInformation。ICatRegester 可以完成新组件类别的登记或取消注册,它也可以将某个组件加入到某个组件类别中,可将其从组件类别中删除。ICatInformation 则可用于获取系统中某个组件类别的数据。如:

- 系统中注册的所有类别。
- 属于某个特定类别的所有组件。
- 某个特定组件所属的所有类别。

当使用 Component Category Manager 时,向组件类别中加入组件或删除其中的组件都将是相当容易的。

6.3.8 OleView

Registry Editor 显示出来的是关于 Windows 注册表的一个原始的、未经任何加工处理的视图,此视图对于了解注册表的结构将是非常有好处的。为了实现必须将其注册在注册表中的组件或实现必须查询注册表的客户,就需要知道注册表是如何组织的。但是如果想对系统中安装的组件进行更为详尽的了解,那么使用注册表将是极为费时的。另外 Registry Editor 基本上是按一个 CLSID 列表来显示所有的内容。

Win32 SDK 中的另外一个程序 OleView 给用户提供了关于注册表中信息的一个更为高级的视图。在此视图下显示出来的将不再是一个长长的 CLSID 或其他 GUID 列表,而是一个由组件的有意义的名称构成的树形视图。另外在 OleView 中还可以看到系统中注册的所有组件类别。学习 OleView 的最好的方法是实际使用它。作者经常用它来检查自注册代码的正确性,例如若 OleView 能够找到这些信息,那么注册过程基本上就是正确的了。

6.4 COM 库函数

所有的 COM 组件和客户都需要完成一些相同的操作。为保证这些操作是按照标准的并且是兼容的方法完成的，COM 定义了一个函数库以实现所有这些操作。

此函数库是在 OLE32.DLL 中实现的。在使用静态链接时，可以使用 OLE32.LIB。本节将介绍其中的若干个重要的操作。

6.4.1 COM 库的初始化

首先我们来看一下 COM 库本身是如何初始化的。在使用 COM 库中的其他函数(除 CoBuildVersion 外，此函数将返回 COM 库的版本号)之前，进程必须先调用 CoInitialize 来初始化 COM 库函数。当进程不再需要使用 COM 库函数时，必须调用 CoUninitialize。这些函数的原型定义如下：

```
HRESULT CoInitialize(void* reserved);  //Argument must be NULL.
void CoUninitialize();
```

对每一个进程，COM 库函数只需初始化一次。这并不是说不能多次调用 CoInitialize 函数，但需要保证每一个 CoInitialize 都有一个相应的 CoUninitialize 调用。当进程已经调用过 CoInitialize 后，再次调用此函数所得到的返回值将是 S_FALSE 而不再是 S_OK。

由于对每一个进程，COM 库只需初始化一次，并且由于 COM 库是用于创建组件的，因此进程中组件无需初始化 COM 库。COM 库的初始化一般只是在 EXE 中进行，而在 DLL 中则无需进行。

一、OleInitialize 的使用

OLE 是建立在 COM 基础之上的，它增加了对类型库、剪贴板、拖放、AcitveX 文档、自动化以及 AcitveX 控件的支持。在 OLE 库中包含对这些特性的额外的支持。在需要使用这些特性时，应调用 OleInitialize 及 OleUninitialize，而不是 CoInitialize 和 CoUninitialize。一般而言，最简单的处理方法是调用 Ole* 函数然后就不用理会其他的问题了。Ole* 函数将调用 Co* 函数。但若程序中没有用到那些额外的功能，使用 Ole* 将会造成资源的浪费。

> **CoInitializeEx**
>
> 在支持分布式 COM(DCOM)的 Windows 操作系统中，可以使用 CoInitializeEx 将某个组件标记为所有线程均可访问的。关于 CoInitializeEx 的详细介绍，请参阅第 12 章。

6.4.2 内存管理

在组件中分配一块内存，然后将其通过一个输出参数传递给客户是一种非常常见的

作法。但这样就产生一个问题:谁来释放这块内存? 如何释放它? 第一个问题可能更为严重,这主要是由于客户和组件可能分别是由不同的程序员实现的,他们所用的编程语言也可能不同,甚至可能是在不同的进程中运行。因此,必须有一种分配和释放内存的标准方法。

COM 解决这一问题的方案是提供一个任务内存分配器。使用此分配器,组件可以给客户提供一块可以由客户删除的内存。并且 COM 的任务内存分配器的设计考虑到了线程之间的同步问题,因此可以在多线程应用程序中使用。

一般程序员可以通过一个接口来使用任务内存分配器,在这种情况下,此接口为 IMalloc,它可以由 CoGetMalloc 返回。为分配一块内存,可以使用 IMalloc::Alloc;而由 IMalloc::Alloc 所分配的内存的释放则可以由 IMalloc::Free 完成。但大多数情况下,调用 CoGetMalloc来获取接口指针,然后使用此指针来调用某个函数,再将此指针释放是一个比较麻烦的过程。为此,COM 库实现了一些方便的帮助函数,如 CoTaskMemAlloc 和 CoTaskMemFree:

```
void* CoTaskMemAlloc(
    ULONG cb      //Size in bytes of block to be allocated
  );
void CoTaskMemFree(
    void* pv      //Pointer to memory block to be freed
  );
```

同某个输出参数相关联的内存的释放应由函数的调用者使用 CoTaskMemFree 完成。

6.4.3 将字符串转化成 GUID

在注册表中包含的是 CLSID 的字符串表示。因此需要一些函数来完成 CLSID 与表示 CLSID 的字符串之间的转换。为此,在 COM 库中应包含这样的函数。例如 StringFromGUID2 可以将某个 GUID 转换成一个字符串:

```
wchar_t szCLSID[39];
int r = ::StringFromGUID2(CLSID_Component1, szCLSID, 39);
```

传给 StringFromGUID2 的参数是一个 Unicode 串(即一个宽字符 wchar_t 类型的数组而不是 char 类型的字符数组)。在非 Unicode 的系统中,需要将结果转化为单字节字符(char)。为此可以使用 ANSI 的 wcstombs 函数,如下所示:

```
#ifndef _UNICODE
  // Convert from widechar to non-wide....
  char szCLSID_single[39];
  wcstombs(szCLSID_single, szCLSID, 39);
#end if
```

其他一些可以完成类似工作的函数如下表所列。

函数	用 途
StringFromCLSID	将CLSID转化成文本串
StringFromIID	将IID转化成文本串
StringFromGUID2	将GUID转化成文本串。此串将被存放在调用者所分配的缓冲区中
CLSIDFromString	将一个文本串转化成CLSID
IIDFromString	将一个文本串转化成IID

上表所列的某些函数需要用到前一节中介绍的任务内存分配器：

```
wchar_t * string;
// Get String from CLSID
::StringFromCLSID(CLSID_Component1, &string);
// Use string
⋮
// Free string
::CoTaskMemFree(string) ;
```

关于接口定义宏

为帮助程序员从C到C++的转换，有人编写了一些宏，以使同一接口定义对于C和C++程序员都能正常工作。在ORJBASE.H和BASETYPS.H中包含这些宏的定义。在本书前面的例子中，作者使用了如下的接口：

```
interface IX : IUnknown
{
    virtual void _stdcall Fx() = 0;
};
```

在使用接口定义宏时，上述接口的定义将如下：

```
DECLARE_INTERFACE(IX, IUnknown)
{
    // IUnknown
    STDMETHOD(QueryInterface)(THIS_ REFIID, PPVOID) PURE;
    STDMETHOD_(ULONG, AddRef) (THIS) PURE;
    STDMETHOD_(ULONG, Release) (THIS) PURE;
    // IX
    STDMETHOD(void,Fx) (THIS) PURE;
}
```

但作者编写程序时一般并不使用这些宏，而是尽量使接口的定义符合C++的风格。在需要发布组件时，可以使用编写接口的一个特殊语言IDL来编写接口。关于IDL，我们将在第10章和第11章中详细介绍。

6.5 本章小结

不论是自己制造飞机也好，还是凌晨两点仍在坚持写作也好，还是开发组件也好，实际上大多数情况下我们的时间和精力主要花在了各种各样的细节上。能不能理解这些细节并正确地处理之将是成功或失败的关键所在。在本章中，我们介绍了 COM 是如何通过 HRESULT 值来返回正确及错误代码的。同时我们还介绍了 GUID 这个神奇的数据结构，它主要是基于一种可以使得无论何时，任何人在任何地方均可以生成一个单一的标识符的算法。COM 将使用 GUID 来标识几乎所有的对象，如组件(CLSID)和接口(IID)等。

另外我们还看到组件的 CLSID 是如何通过 Windows 的注册表而映射到其文件名的。安装程序或 REGSVR32.EXE 可以调用实现组件的 DLL 中输出的 DllRegisterServer 函数来注册相应的组件。在注册时，组件至少要将其 CLSID 和文件名记录在注册表中。

下一章我们将讨论 COM 如何使用某个组件的 CLSID 来创建之。当然这比制造飞机要简单得多。

第7章 类 厂

随着作者年龄的增长，自己的理想也逐渐发生了变化：不再想当一名消防队员，而想成为一名 Lego 积木块的设计人员。那时作者头脑里充满了各种新奇的想法，并且坚信根据这些设想，将可制造能够搭出极漂亮模型的灵活的积木块。作者那时甚至还将自己的一些想法寄给了 Lego 公司(但遗憾的是没有引起他们的重视)。虽然如此，但现在却有了一些新的设备，使得任何人均可以在家里制造出与 Lego 兼容的积木块。

这些设备被称作是三维打印机。这种名符其实的设备同喷墨打印机类似，但喷出的却不是墨而是精细的塑料层。这些塑料层的厚度不超过一个毫米。通过不断地将塑料喷射在某个区域中，可以构造出极其复杂的三维形状。这些形状可以作为某些原型的一部分，建立部件的原型，或直接用作某些应用的一部分。有了这样的一台设备就可以开办一个室内塑料积木厂了。使用一个 CAD 软件包，可以毫不费力地设计出新的积木块并生成它，然后用我们的室内塑料积木厂，就可以制造出建造模型所需的积木块了。这样我们手头将永不会缺积木块，虽然 Lego 公司更希望人们去买他们制造的积木块。

在本章中我们将介绍另外一种类型的工厂，在这种“工厂”中我们可以制造出所需的组件。这种工厂被称作是类厂。这种工厂同那种需花费 50 000 美元的三维打印机相比要便宜得多，也简单得多，因为类厂实际上就是一个带有能够创建其他组件的组件。

在介绍类厂之前，我们先来看一下创建组件的最简单的方法，这就是使用 CoCreateInstance 函数。正因如此，CoCreateInstance 也是创建组件时使用得最多的一种方法。但不幸的是，CoCreateInstance 的灵活性却不足以使之能够满足所有组件的需求。这就是为什么要引入类厂的原因。所有的组件都是用类厂创建的。CoCreateInstance 实际上也是按照一般的方法通过类厂来创建组件的。当客户直接使用类厂时，将能够获取更高的灵活性。这一点同在我们的室内塑料积木块厂制造所需的积木块要比从市场上购买更灵活是类似的。

7.1 CoCreateInstance

在 COM 库中包含一个用于创建组件的名为 CoCreateInstance 的函数。此函数需要一个 CLSID 参数，在此基础上创建相应组件的一个实例，并返回此组件实例的某个接口。本节将讨论如何使用 CoCreateInstanc 以及它有一些什么样的限制。下面我们先来看一下 CoCreateInstance 的定义。

7.1.1 CoCreateInstance 的声明

CoCreateInstance 的声明如下：

```
HRESULT _stdcall CoCreateInstance(
  const CLSID& clsid ,
  IUnknown * pIUnknownOuter ,      // Outer Component
  DWORD dwClsContext ,             // Server Context
  const IID& iid ,
  void** ppv
);
```

可以看到，CoCreateInstance 有四个输入参数和一个输出参数。第一个参数是待创建组件的 CLSID。第二个参数是用于聚合组件的，我们将在下一章详细讨论它们。第三个参数 dwClsContext 的作用是限定所创建的组件的执行上下文。稍后我们就来讨论 dwClsContext。

第四个参数 iid 为组件上待使用的接口的 IID。CoCreateInstance 将在最后一个参数中返回此接口的指针。通过将一个 IID 传给 CoCreateInstance，客户将无需在创建组件之后去调用其 QueryInterface 函数。

7.1.2 CoCreateInstance 的使用

CoCreateInstance 的使用同 QueryInterface 的使用是一样简单的：

```
// Create Component.
IX* pIX = NULL ;
HRESULT hr = ::CoCreateInstance( CLSID_Component1,
                    NULL,
                    CLSCTX_INPROC_SERVER,
                    IID_IX,
                    (void**)&pIX ) ;
if (SUCCEEDED(hr))
{
  pIX->Fx() ;
  pIX->Release() ;
}
```

在上例中我们创建了一个由 CLSID_Component1 标识的组件。由于此处不需要聚合组件，因此第二个参数为 NULL。在下一章中我们将给此参数一个非 NULL 值。CLSCTX_INPROC_SERVER 值告诉 CoCreateInstance 只装载包含进程中服务器或 DLL 中的组件。

传给 CoCreateInstance 的最后两个参数同传给 QueryInterface 的参数是一样的。在本例中，我们使用 IID_IX 来表示希望得到接口 IX 的指针，CoCreateInstance 将把此指针返回在

pIX 变量中。若 CoCreateInstance 成功返回，那么 IX 接口就可以使用了。调用 Release()函数表示客户对接口和组件的使用已经完成了。

7.1.3 类上下文

CoCreateInstance 的第三个参数 dwClsContext 可以控制所创建的组件是在与客户相同的进程中运行，还是在不同的进程中运行，或者是在另外一台机器上运行。此参数的值可以是如下所列各值的组合。

CLSCTX_INPROC_SERVER	客户希望创建在同一进程中运行的组件。为能够同客户在同一进程中运行，组件必须是在 DLL 中实现的。
CLSCTX_INPROC_HANDLER	客户希望创建进程中处理器。一个进程中处理器实际上是一个只实现了某个组件一部分的进程中组件。该组件的其他部分将由本地或远程服务器上的某个进程外组件实现。
CLSCTX_LOCAL_SERVER	客户希望创建一个在同一机器上的另外一个进程中运行的组件。本地服务器是由 EXE 实现的(请参阅第 10 章)。
CLSCTX_REMOTE_SERVER	客户希望创建一个在远程机器上运行的组件。此标志需要分布式 COM 正常工作(我们将在第 10 章中讨论)。

客户可以在三种不同的上下文中使用某个组件：进程中、本地以及远程。在某些情况下，客户可能希望只使用进程中组件，以获取更高的速度。在另外一些情况下，客户可能并不希望组件在它们的进程空间中运行，因进程中组件将可以读写进程空间中的任意内存地址，而这可能是极不安全的。但在大多数情况下，客户对组件所处理的执行上下文可能并不太关心。因此在 OBJBASE.H 中用上述值的组合(按位或)定义了一些方便的常量，如表 7-1 所列。

表 7-1 执行上下文标记的一些预定义组合

常量名称	值
CLSCTX_INPROC	CLSCTX_INPROC_SERVER CLSCTX_INPROC_HANDLER
CLSCTX_ALL	CLSCTX_INPROC_SERVER CLSCTX_INPROC_HANDLER CLSCTX_LOCAL_SERVER CLSCTX_REMOTE_SERVER
CLSCTX_SERVER	CLSCTX_INPROC_SERVER CLSCTX_LOCAL_SERVER CLSCTX_REMOTE_SERVER

另外要说明的是，CLSCTX_REMOTE_SERVER 值只是在包含 OBJEBASE.H 之前将 _WIN32_WINNT的值定义为大于或等于 0x0400 时才会被加入到 CLSCTX_ALL 和 CLSCTX_SERVER 中。（在包含 OBJBASE.H 之前定义 _WIN32_DCOM 的效果将是一样的。）但应注意的是：若在某个不支持 DCOM 的系统中将 CLSCTX_REMOVE_SERVER 值传给 CoCreateInstance，此函数将会失败并返回一个 E_INVALIDARG 值。有些人经常在编译程序时将 _WIN32_WINNT值设成 0x0400，然后将编译好的程序放到 Microsoft Windows NT3.51 或 Windows 95 上运行，但不幸的是，这些系统并不支持 DCOM。在第 10 章中我们将对 CLSCTX_LOCAL_SERVER 和 CLSCTX_REMOTE_SERVER 进行更为详细的讨论。

7.1.4 客户程序清单

在本章的示例代码中我们将创建我们的第一个纯 COM 客户和组件。程序清单 7-1 给出了客户的代码。此客户同第 5 章中给出的客户的真正的差别在于客户创建组件时使用的是 CoCreateInstance。另外它还使用了 CoInitialize 和 CoUninitialize 来初始化 COM 库（请参阅第 6 章）。

程序清单 7-1 客户的完整代码。

CLIENT.CPP

```
//
// Client.cpp - client implementation
//
#include <iostream.h>
#include <objbase.h>
#include "Iface.h"
void trace(const char* msg) { cout << "Client: \t\t" << msg << endl; }
//
// main function
//
int main()
{
    // Initialize COM Library
    CoInitialize(NULL);
    trace("Call CoCreateInstance to create");
    trace("  component and get interface IX.");
    IX* pIX = NULL;
    HRESULT hr = ::CoCreateInstance(CLSID_Component1,
                                    NULL,
                                    CLSCTX_INPROC_SERVER,
                                    IID_IX,
                                    (void**)&pIX);
```

```
    if (SUCCEEDED(hr))
    {
        trace("Succeeded getting IX.") ;
        pIX->Fx() ;          // Use interface IX.
        trace("Ask for interface IY.") ;
        IY* pIY = NULL ;
        hr = pIX->QueryInterface(IID_IY, (void**)&pIY) ;
        if (SUCCEEDED(hr))
        {
            trace("Succeeded getting IY.") ;
            pIY->Fy() ;       // Use interface IY.
            pIY->Release() ;
            trace("Release IY interface.") ;
        }
        else
        {
            trace("Could not get interface IY.") ;
        }
        trace("Ask for interface IZ.") ;
        IZ* pIZ = NULL ;
        hr = pIX->QueryInterface(IID_IZ, (void**)&pIZ) ;
        if (SUCCEEDED(hr))
        {
            trace("Succeeded in getting interface IZ.") ;
            pIZ->Fz() ;
            pIZ->Release() ;
            trace("Release IZ interface.") ;
        }
        else
        {
            trace("Could not get interface IZ.") ;
        }
        trace("Release IX interface.") ;
        pIX->Release() ;
    }
    else
    {
        cout << "Client: \t\tCould not create component. hr = "
            << hex << hr << endl ;
    }
    // Uninitialize COM Library
    CoUninitialize() ;
    return 0 ;
}
```

7.1.5 CoCreateInstance 的不灵活性

在任何一个面向对象的系统中，对象的创建都是非常重要的。因为在能够使用某个对象之前，必须先建立它。若各对象的创建过程不相同，那么想要对对象进行一致的处理将是一件极为困难的事情。因此我们希望对象的创建过程应尽可能地灵活，以便能够按相似的方式创建所有的组件。创建过程越灵活，那么它将越能够适于更多组件的需求。

前面我们看到 CoCreateInstance 创建组件的过程是：传给它一个 CLSID，然后它创建相应的组件，并返回指向所请求的接口的指针。大多数情况下，CoCreateInstance 确实是可以完成它的工作的。但 CoCreateInstance 却没有给客户提供一种能够控制组件创建过程的方法。当 CoCreateInstance 完成之后，组件实际上已经建立好了。在建立好一个组件之后，想要控制将组件装载到内存中何处或检查客户是否有权限来创建该组件基本上已经不可能了。这充分说明，CoCreateInstance 的灵活性是不够的。

问题的关键在于如何控制组件的创建过程。在此我们并不需要关心组件的初始化，因为在创建好组件之后，可以通过一个接口来初始化它。但在建立好一个组件之前是无法获取它的某个接口指针的。因此也就无法对组件的创建加上条件限制。

解决这个问题的方法就是使用另外一个专门用于创建所需组件的组件。这个组件就是我们下面将要介绍的类厂。

7.2 类厂

CoCreateInstance 实际上并没有直接创建 COM 组件，而是创建了一个被称作是类厂的组件。而所需的组件正是由此类厂创建的。类厂组件的唯一功能就是创建其他的组件。更精确地讲，某个特定的类厂将创建只同某个特定的 CLSID 相应的组件。客户可以通过类厂所支持的接口来对类厂创建组件的过程加以控制。创建组件的标准接口是 IClassFactory，用 CoCreateInstance 创建的组件实际上是通过 IClassFactory 创建的。

下面我们来看一下客户如何通过一个类厂直接创建所需的组件。这个过程的第一步是创建类厂本身，然后再使用一个接口如 IClassFactory 来创建所需的组件。

7.2.1 CoGetClassObject

CoCreateInstance 将接收一个 CLSID 作为参数并返回指向所创建的组件中某个接口的指针。为创建同某个 CLSID 相应的类厂，需要一个与 CoCreateInstance 等价的、也可接收一个 CLSID 作为参数并返回相应类厂中某个接口指针的函数。这个函数就是 COM 库中的 CoGetClassObject。

此函数的声明如下：

```
HRESULT _stdcall CoGetClassObject(
    const CLSID& clsid,
    DWORD dwClsContext,
    COSERVERINFO * pServerInfo,      // Reserved for DCOM
    const IID& iid,
    void ** ppv
  );
```

可以看到，CoGetClassObject 同 CoCreateInstance 是非常相似的。这两个函数的第一个参数都是待创建组件的 CLSID。第二个参数则均为所创建组件的执行上下文 dwClsContext。另外它们的最后两个参数也是相同的。但 CoGetClassObject 返回的是指向类厂中某个接口的指针，而 CoCreateInstance 返回的则是指向组件中某个接口的指针。这两个函数只有一个参数不同。CoCreateInstance 将接收一个 IUnknown 指针，而 CoGetClassObject 则将接收一个 COSERVERINFO 指针。COSERVERINFO 指针将被 DCOM 用于控制对远程组件的访问。关于这个问题我们将在第 10 章中讨论。

当然最大的区别在于 CoGetClassObject 返回是的指向所需组件的类厂而不是指向 组件本身的一个指针。客户可以用 CoGetClassObject 所返回的指针来创建所需的组件。这个指针通常是一个 IClassFactory 指针。

7.2.2 IClassFactory

类厂所支持的用于创建组 件的标准接口是 IClassFactory。大多数组件均可使用 IClassFactory接口来创建。此接口的声明如下。

```
interface IClassFactory : IUnknown
{
HRESULT  stdcall
CreateInstance(   IUnknown * pUnknownOuter,
                  const IID& iid,
                  void ** ppv );
HRESULT  stdcall LockServer(BOOL bLock) ;
};
```

IClassFactory 具有两个成员函数。第一个成员函数为 CreateInstance，第二个为 LockServer。关于 LockServer，我们将在本章末介绍。

一、CreateInstance

IClassFactory::CreateInstance 函数的参数也是我们所熟悉的。第一个参数 pUnknownOuter 为指向某个 IUnknown 接口的指针。这一点同传给 CoCreateInstance 的 IUnknown 指针是相同的。关于此参数我们将在下一章讨论组件聚合时介绍。

此函数的其余两个参数同传给 QueryInterface 的参数是相同的。使用这两个参数，客

户可以在创建组件的同时请求此组件中某个接口的指针。这样可以省去客户的一次函数调用。当组件是在一台远程机器上运行时,它还可以省去一次来回传输的开销。

但 CreateInstance 最令人感兴趣的方面却不是它接收的那些参数——而是它所不接收的参数。可以看到 IClassFactory::CreateInstance 并没有接收一个 CLSID 参数。这意味着此函数将只能创建同某个 CLSID —— 即传给 CoGetClassObject 的 CLSID —— 相应的组件。

IClassFactory2

除了 IClassFactory 之外,Microsoft 还定义了另外一个创建接口 IClassFactory2。此接口在 IClassFactory 的基础上增加了许可或权限功能。此时,为使类厂能够创建所需的组件,客户必须通过 IClassFactory2 给类厂提供正确的关键字或许可。通过使用 IClassFactory2,类厂可以保证客户只能获得它能合法访问的组件,并具有对此组件的访问授权。但作者相信这并不是最后的创建接口。

7.2.3 CoCreateInstance 与 CoGetClassObject 的比较

在每次创建组件时,先创建相应的类厂,然后用所获取的 IClassFactory 指针来创建所需的接口需要完成的工作显然比直接调用 CoCreateInstance 来创建所需的组件要复杂一些。这也正是为什么大多数组件都使用 CoCreateInstance 来创建组件的原因。但前面提到过,CoCreateInstance 实际上是通过 CoGetClassObject 实现的。下面的代码表示了如何用 CoGetClassObject 来实现 CoCreateInstance:

```
HRESULT CoCreateInstance(const CLSID& clsid,
                         IUnknown * pUnknownDuter,
                         DWORD dwClsContext,
                         const IID& iid,
                         void ** ppv)
{
  // Set the out parameter to NULL.
  *ppv = NULL;
  // Create the class factory and
  // get an IClassFactory interface pointer.
  IClassFactory * pIFactory = NULL;
  HRESULT hr = CoGetClassObject(clsid,
                                dwClsContext,
                                NULL,
                                IID_IClassFactory,
                                (void**)&pIFactory);
  if (SUCCEEDED(hr))
  {
    //Create the component.
```

```
    hr = pIFactory->CreateInstance(pUnknownOuter,iid, ppv) ;
    // Release the class factory.
    pIFactory->Release() ;
  }
  return hr ;
}
```

可以看到 CoCreateInstance 首先调用 CoGetClassObject 来获取类厂中的 IClassFactory 接口的指针,然后再使用此指针来调用 IClassFactory::CreateInstance 完成新组件的创建。

一、为什么要使用 CoGetClassObject

大多数情况下,组件的创建均使用 CoCreateInstance 而不使用 CoGetClassObject。但是在如下两种情况下应使用 CoGetClassObject 而不应使用 CoCreateInstance。第一种情况是,若想用不同于 IClassFactory 的某个创建接口来创建组件,则必须使用 CoGetClassObject。因此如果想使用 IClassFactory2 来创建组件,就应使用 CoGetClassObject。第二种情况是,若需创建同一组件的多个实例,那么使用 CoGetClassObject 将可以获得更高的效率。因为这样只需创建相应的类厂一次,而 CoCreateInstance 则需为每一个实例分别创建并释放相应的类厂。另外 CoGetClassObject 使得客户可以对组件的创建过程进行更多的控制。

使用类厂来创建组件比使用 CoCreateInstance 更易让人困惑。但只要能记住类厂不过是创建其他组件的一个简单组件,那么对类厂的理解将会容易得多。

7.2.4 类厂的若干特性

在介绍如何实现一个类厂之前,我们先介绍一下类厂的一些特性。首先类厂的一个实例将只能创建同某个 CLSID 相应的组件。这一点从 CoGetClassObject 将接收一个 CLSID 而 IClassFactory::CreateInstance 却不接收此参数可以看出来。其次与某个特定 CLSID 相应的类厂将是由实现组件的开发人员实现的。大多数情况下,类厂组件包含在与它所创建的组件相同的 DLL 中。

由于类厂的一个实例只同某个 CLSID 相应并且由于类厂和它所创建的组件是由同一开发人员实现的,因此,类厂可以知道并且确实具有它所创建的组件的一些特殊知识。在实现组件时使用关于它所创建的组件的一些特殊知识并不是不可以的。类厂就是需要知道如何创建相应的组件并将这一点封装起来,以便客户能够尽可能地同组件所具有的特殊需求分开。

7.3 类厂的实现

本节将讨论组件的实现特别是类厂的实现。在此之前我们先来看一下类厂本身是如

何被创建的。

7.3.1 DllGetClassObject 的使用

第 5 章介绍的 CallCreateInstance 调用了 DLL 中的 CreateInstance 来创建所需的组件。类似地，CoGetClassObject 需要 DLL 中的一个特定的函数来创建组件的类厂。此函数的名称为 DllGetClassObject。CoGetClassObject 将调用此函数完成类厂的创建。DllGetClassObject 的声明如下：

```
STDAPI DllGetClassObject(
    const CLSID& clsid,
    const IID& iid,
    void** ppv
    );
```

读者对传给 DllGetClassObject 的三个参数应该已经非常熟悉了。这三个参数与 CoGetClassObject 的三个参数是相同的。第一个参数为类厂将要创建的组件的 CLSID；第二个参数为类厂中客户希望得到的接口 ID。而接口的指针将被返回在最后一个参数中。

将 CLSID 传给 DllGetClassObject 是非常有意义的。这一点使得一个 DLL 可以支持任意数目的组件，因为在这种情况下 DllGetClassObject 可以用 CLSID 来选择合适的类厂。

7.3.2 组件的创建过程

为更清楚地了解组件到底是如何被创建的，我们来看一下图 7-1。在该图中列出了组

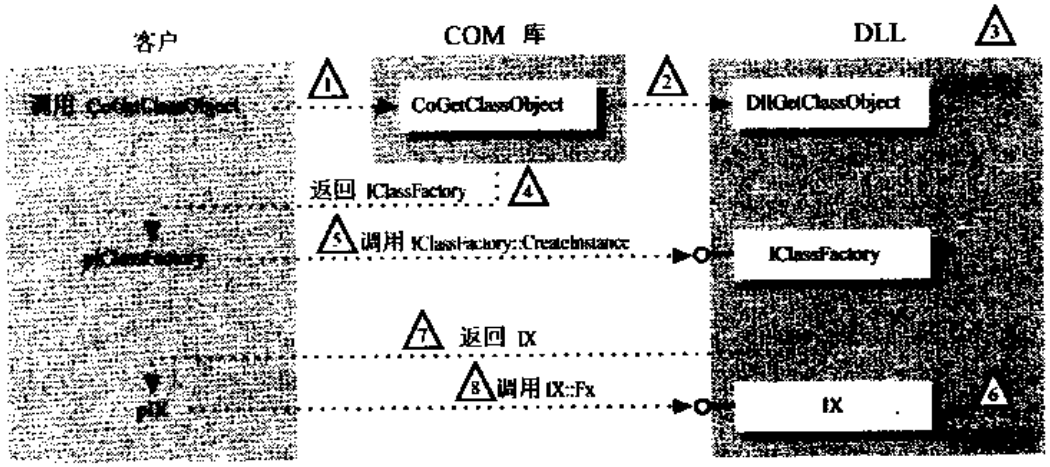


图 7-1 使用 COM 库函数和类厂创建组件的过程

件的创建过程中所涉及到的主要角色。首先是客户，它将通过调用 CoGetClassObject 来启动组件的创建过程。其次是 COM 库，它实现了 CoGetClassObject 函数。第三个角色是 DLL，其中实现了被 CoGetClassObject 调用的 DllGetClassObject 函数。DllGetClassObject 的任

务就是创建客户所请求的类厂。当然它创建此类厂的方式完全是由开发人员决定的，因这一点是需要向客户隐藏起来的。

在创建好类厂之后，客户将使用 IClassFactory 接口来创建相应的组件。IClassFactory::CreateInstance 如何创建组件也是由开发人员决定的。IClassFactory 将把这个过程封装起来以便类厂能够使用关于组件的内部知识来创建它。

下面我们就可以来具体看一下组件是如何实现的了。

7.3.3 组件代码清单

在程序清单 7-2 中实现了组件即相应的类厂。这个类厂实际上是由 C++ 类 CFactory 实现的。关于 CFactory，需要说明的第一点是，它只不过是另外一个组件而已，例如它也按与其他组件相同的方式实现了 IUnknown 接口。CFactory 和 CA 的唯一区别在于它们所支持的接口的不同。

在读者阅读此程序清单时，请特别注意 CFactory::CreateInstance 和 DllGetClassObject 的实现。

程序清单 7-2 组件、类厂及 DLL 所输出函数的完整代码。

```
CMPNT.CPP
//
// Cmpnt.cpp
//
#include <iostream.h>
#include <objbase.h>

#include "Iface.h"      // Interface declarations
#include "Registry.h"   // Registry helper functions

// Trace function
void trace(const char * msg) { cout << msg << endl ;}

//////////////////////////////////////////////////////////
//
// Global variables
//
static HMODULE g_hModule = NULL ;     // DLL module handle
static long g_cComponents = 0 ;       // Count of active components
static long g_cServerLocks = 0 ;      // Count of locks

// Friendly name of component
const char g_szFriendlyName[] = "Inside COM, Chapter 7 Example" ;

// Version-independent ProgID
const char g_szVerIndProgID[] = "InsideCOM.Chap07" ;

// ProgID
const char g_szProgID[] = "InsideCOM.Chap07.1" ;
```

```
//////////////////////////////////////////////////////////////
//
// Component
//
class CA : public IX,
           public IY
{
public:
    // IUnknown
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) ;
    virtual ULONG __stdcall AddRef() ;
    virtual ULONG __stdcall Release() ;

    // Interface IX
    virtual void __stdcall Fx() { cout << "Fx" << endl ;}

    // Interface IY
    virtual void __stdcall Fy() { cout << "Fy" << endl ;}

    // Constructor
    CA() ;

    // Destructor
    ~CA() ;
private:
    // Reference count
    long m_cRef ;
} ;
//
// Constructor
//
CA::CA() : m_cRef(1)
{
    InterlockedIncrement(&g_cComponents) ;
}
//
// Destructor
//
CA::~CA()
{
    InterlockedDecrement(&g_cComponents) ;
    trace("Component:\t\tDestroy self.") ;
}
//
// IUnknown implementation
//
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
```

```
{
    if (iid == IID_IUnknown)
    {
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IX)
    {
        *ppv = static_cast<IX*>(this);
        trace("Component:\t\tReturn pointer to IX.");
    }
    else if (iid == IID_IY)
    {
        *ppv = static_cast<IY*>(this);
        trace("Component:\t\tReturn pointer to IY.");
    }
    else
    {
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}
ULONG __stdcall CA::AddRef()
{
    return InterlockedIncrement(&m_cRef);
}
ULONG __stdcall CA::Release()
{
    if (InterlockedDecrement(&m_cRef) == 0)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}
////////////////////////////////////////////////////////
//
// Class factory
//
class CFactory : public IClassFactory
{
public:
    // IUnknown
```

```
        virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
        virtual ULONG   __stdcall AddRef();
        virtual ULONG   __stdcall Release();

        // Interface IClassFactory
        virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
                                                 const IID& iid,
                                                 void** ppv);
        virtual HRESULT __stdcall LockServer(BOOL bLock);

        // Constructor
        CFactory() : m_cRef(1) {}

        // Destructor
        ~CFactory() { trace("Class factory:\t\tdestroy self."); }

    private:
        long m_cRef;
    };

    //
    // Class factory IUnknown implementation
    //
    HRESULT __stdcall CFactory::QueryInterface(const IID& iid, void** ppv)
    {
        if ((iid == IID_IUnknown) || (iid == IID_IClassFactory))
        {
            *ppv = static_cast<IClassFactory*>(this);
        }
        else
        {
            *ppv = NULL;
            return E_NOINTERFACE;
        }
        reinterpret_cast<IUnknown*>(*ppv)->AddRef();
        return S_OK;
    }
    ULONG __stdcall CFactory::AddRef()
    {
        return InterlockedIncrement(&m_cRef);
    }
    ULONG __stdcall CFactory::Release()
    {
        if (InterlockedDecrement(&m_cRef) == 0)
        {
            delete this;
            return 0;
        }
```

```
    return m_cRef;
}

//
// IClassFactory implementation
//
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
                                           const IID& iid,
                                           void** ppv)
{
    trace("Class factory: \t \tCreate component.");
    // Cannot aggregate.
    if (pUnknownOuter != NULL)
    {
        return CLASS_E_NOAGGREGATION;
    }
    // Create component.
    CA* pA = new CA;
    if (pA == NULL)
    {
        return E_OUTOFMEMORY;
    }
    // Get the requested interface.
    HRESULT hr = pA->QueryInterface(iid, ppv);
    // Release the IUnknown pointer.
    // (If QueryInterface failed, component will delete itself.)
    pA->Release();
    return hr;
}
// LockServer
HRESULT __stdcall CFactory::LockServer(BOOL bLock)
{
    if (bLock)
    {
        InterlockedIncrement(&g_cServerLocks);
    }
    else
    {
        InterlockedDecrement(&g_cServerLocks);
    }
    return S_OK;
}
//////////////////////////////////////////////////////////
//
```

```
// Exported functions
//

//
// Can DLL unload now?
//
STDAPI DllCanUnloadNow()
{
    if ((g_cComponents == 0) && (g_cServerLocks == 0))
    {
        return S_OK;
    }
    else
    {
        return S_FALSE;
    }
}

//
// Get class factory
//
STDAPI DllGetClassObject(const CLSID& clsid,
                         const IID& iid,
                         void** ppv)
{
    trace("DllGetClassObject:\tCreate class factory.");
    // Can we create this component?
    if (clsid != CLSID_Component)
    {
        return CLASS_E_CLASSNOTAVAILABLE;
    }
    // Create class factory.
    CFactory* pFactory = new CFactory;  // No AddRef in constructor
    if (pFactory == NULL)
    {
        return E_OUTOFMEMORY;
    }
    // Get requested interface.
    HRESULT hr = pFactory->QueryInterface(iid, ppv);
    pFactory->Release();

    return hr;
}

//
// Server registration
//
```

```
STDAPI DllRegisterServer()
{
    return RegisterServer(g_hModule,
                          CLSID_Component1,
                          g_szFriendlyName,
                          g_szVerIndProgID,
                          g_szProgID);
}

//
// Server unregistration
//
STDAPI DllUnregisterServer()
{
    return UnregisterServer(CLSID_Component1,
                            g_szVerIndProgID,
                            g_szProgID);
}

//////////////////////////////////////////////////////////
//
// DLL module information
//
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD dwReason,
                      void* lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        g_hModule = hModule;
    }
    return TRUE;
}
```

```
Client:                    Call CoCreateInstance to create
Client:                      component and get interface IX.
DllGetClassObject:         Create class factory.
Class factory:             Create component.
Component:                 Return pointer to IX.
Class factory:             Destroy self.
Client:                    Succeeded getting IX.
Fx
Client:                    Ask for interface IY.
Component:                 Return pointer to IY.
```

```
Client:                    Succeeded getting IY.
Fy
Client:                    Release IY interface.
Client:                    Ask for interface IZ.
Client:                    Could not get interface IZ.
Client:                    Release IX interface.
Component:                 Destroy self.
```

上面所示的 DllGetClassObject 的实现实际上完成了三件事。首先它将检查客户所请求的类厂组件是它所能够创建的。其次它将使用 new 操作符完成组件的创建。最后它将向类厂组件查询客户所请求的接口。IClassFactory::CreateInstance 的实现方法同 DllGetClassObject 是相同的。这两个函数都将创建一个组件然后向它查询某个接口，所不同的只是 IClassFactory::CreateInstance 创建的是 CA，而 DllGetClassObject 创建的是 CFactory。

注意，DllGetClassObject 对其所创建的类厂可以说是无所不知的，而 IClassFactory::CreateInstance 对其所创建的组件也可以说是知根知底。这些函数可以将组件的实现细节同客户完全隔离开。这些函数的实现都可以作到可复用，但在某些地方它们需要知道如何创建它们所需的特定类厂或组件。DllGetClassObject 和 IClassFactory::CreateInstance 用于创建其组件的方法完全是由组件的开发人员决定的。在第 9 章中我们将给出 DllGetClassObject 和 IClassFactory 的一个可复用的实现。

7.3.4 流程控制

下面我们来详细看一下程序清单 7-1 和程序清单 7-2 中客户和组件的流程控制。图 7-2 以图形方式显示出了相应的控制流程。在此图中，时间是从上到下的。五个主要的结构化元素，即客户、COM 库、DLL、类厂和组件分别被列在不同的列中。每一个元素都有一个相应的竖直线。其中实线表示相应的元素已经被建立好了并且是活动的，而虚线则表示相应的元素不再是活动的。线上的灰色条纹分别表示每一个操作的生命期。水平线表示将流程控制从一个结构化元素传递给下一个元素的函数调用。

为简化图 7-2，我们省略了那些对 IUnknown 成员的调用及其他幕后的一些调用。下面我们对此图作一个简要解释。首先客户调用 COM 库中实现的 CoCreateInstance。CoCreateInstance 实际是用 CoGetClassObject 实现的。CoGetClassObject 将在注册表中查找指定的组件。找到之后，它将装载实现此组件的 DLL。装载成功之后，它将调用在 DLL 服务器中实现的 DllGetClassObject。此函数的作用是创建相应的类厂，在本例中我们是用 C++ 的 new 操作符实现的。另外 DllGetClassObject 还将查询 IClassFactory 接口，并将其返回给 CoCreateInstance。然后 CoCreateInstance 将使用此接口来调用 IClassFactory::CreateInstance 函数。在本例中，IClassFactory::CreateInstance 调用了 new 操作符来创建指定的组件，并查询此组件的 IX 接口。在得到了此接口之后，CoCreateInstance 将释放相应的类厂并将 IX 指针返回给客户。然后客户就可以使用此指针来调用组件中的某个方法了。

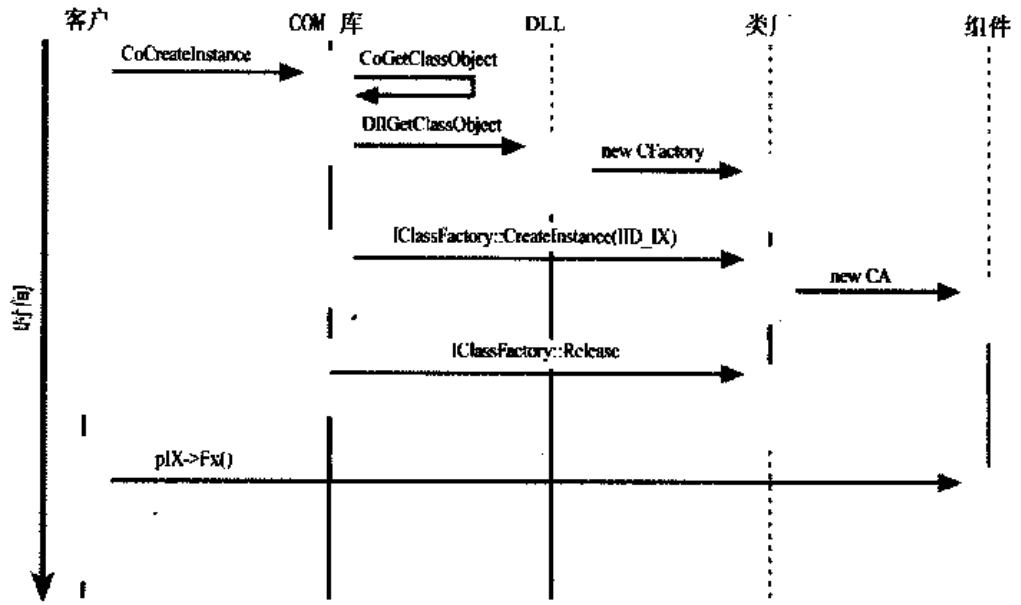


图 7-2 CoCreateInstance 将为客户完成组件创建过程中的大多数细节工作

7.3.5 组件的注册

可以注意到实现组件的 DLL 将输出四个函数。前面我们已经讨论了将被 COM 库函数用于创建类厂的 DllGetClassObject 函数。另外的三个输出函数将被用于组件的登记。

DllRegisterServer 和 DllUnregisterServer 的作用是在 Windows 的注册表中登记某个组件或取消某个组件的登记。这些函数我们在第 6 章中已作过简要的介绍。在 REGISTRY.CPP 中包含这些函数的实现。在此我们不准备对此实现进行讨论,因这些代码实际上比较简单的,读者一看基本上就能明白。在本书后面的章节中我们将继续使用此 REGISTRY.CPP 来完成组件的登记。

本章例子的 makefile 中在 CMPNT.DLL 的编译和链接之后包含如下的命令:

```
regsvr32 -s Cmptnt.dll
```

前一章中我们已经介绍过,REGSVR32.EXE 将调用 DllRegisterServer 函数完成组件的登记。若不使用此 makefile,那么需要手工来完成这一步骤。为方便起见,作者编写了一个名为 REGISTER.BAT 的批文件,其中只包含这一命令。在后续的章节中我们也将提供类似的文件来完成相应的登记工作。

一、DllMain

函数 DllRegisterServer 需要包含它的 DLL 的模块句柄,以获取 DLL 的文件名并登记它。模块句柄将被传给 DllMain。C++ 程序有一个名为 main 的函数,这是程序执行的开

始处。Windows 程序使用的是 WinMain，而 DLL 使用的是 DllMain。在实现 DllMain 的时候，获取模块的句柄是非常简单的：

```
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD dwReason,
                      void* lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        g_hModule = hModule;
    }
    return TRUE;
}
```

可以看到，DllMain 将模块的句柄保存在一个全局变量 g_hModule 中，以供 DllregisterServer 和 DllUnregisterServer 使用。

7.4 同一 DLL 中的多个组件

前面提到过 DllGetClassObject 使得我们可以在同一个 DLL 中实现多个组件。这一点的关键之处在于可以将待创建的组件的 CLSID 传给 DllGetClassObject。对于每一个 CLSID，DllGetClassObject 可以方便地创建一个不同的类厂（参见图 7-3）。

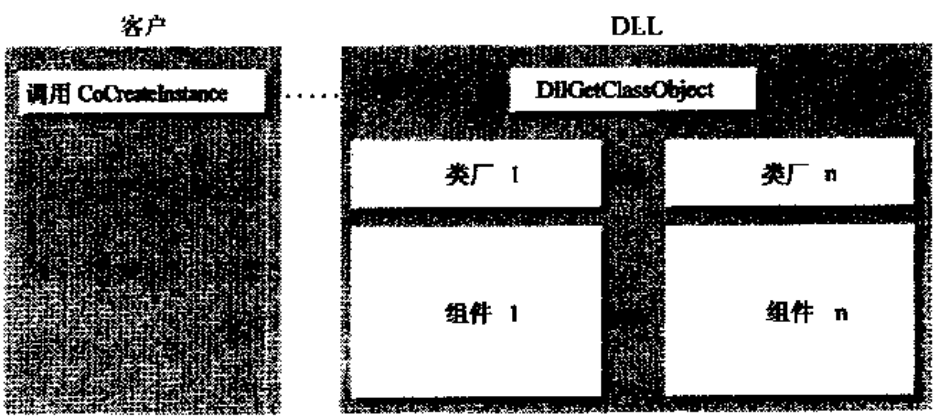


图 7-3 在一个 DLL 中可包含多个组件

一个 DLL 可以支持多个不同组件的事实正好也从一个侧面说明了 DLL 并不等价于组件，而相当于是一个组件服务器。这种将 DLL 当成是在收到客户的请求后提供相应的组件的想法是非常有效的。按照这种想法，DLL 实际上是为组件实现的分发提供了一种方法。

7.4.1 类厂实现的复用

一般而言，程序员都非常憎恶一而再、再而三地编写相同的代码。例如用三个不同的类 CFactory1、CFactory2 以及 CFactory3 来完成组件 CA、CB 和 CC 的创建实际是就是一个浪费。因为在此种情况下，这三个不同类厂的实现中，唯一可能会出现差别的地方将是：

```
CA* pA = new CA;
pA->QueryInterface(...);
```

在设计类厂和组件时，如果设计得好的话，可以做到只用一个类厂的实现来完成所有组件的创建。作者一般喜欢为所编写的每一个组件建立一个简单的函数。这一个函数将使用 new 操作符来创建此组件并返回一个 IUnknown 指针。然后可以用指向这些函数指针的指针构成一个表格，而对此表格的访问将用各组件的 CLSID 作为索引。DllGetClassObject 将查找此表格以获取相应的创建函数指针，并在创建好相应的类厂之后将此指针传给此类厂。之后类厂将通过此指针来调用相应的组件创建函数，而不再直接通过 new 操作符来完成这一操作(参见图 7-4)。

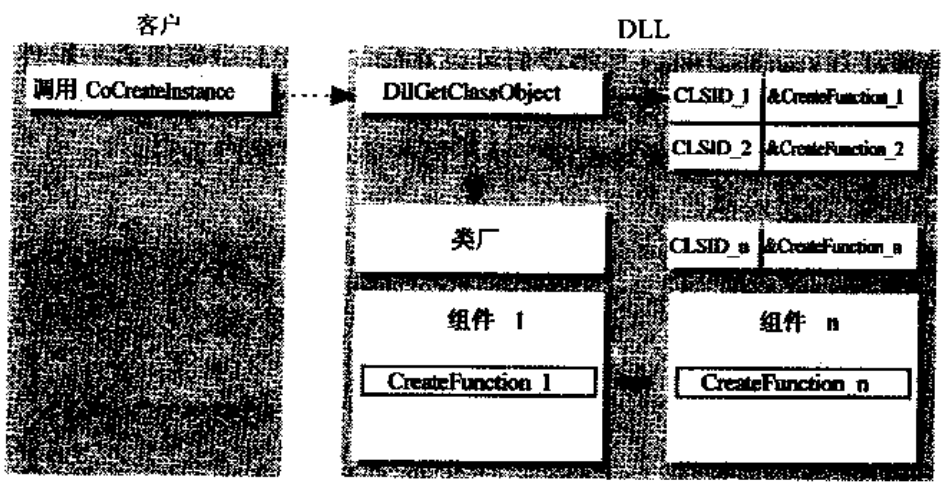


图 7-4 可以支持多个组件的类厂的简单实现

在第 9 章中我们将按照上面的想法实现一个可复用的类厂。

在接着介绍下面的内容之前，作者还想强调一点。即使当我们在几个不同的组件之间共享类厂的代码时，类厂的一个实例也仅能创建同某个 CLSID 相应的组件。类厂实例同 CLSID 之间的对应关系将永远是一对一的。虽然 CFactory 可以实现我们所需的所有类厂，但 CFactory 的任一实例却仍然只能创建同某个 CLSID 相应的组件。这一点主要是由于 IClassFactory::CreateInstance 不接收 CLSID 作为参数的结果。

7.5 DLL 的卸载

到目前为止,我们还一直没有讨论过 LockServer 和 DllCanUnloadNow 函数。下面我们将具体讨论它。

在第 5 章中我们已经看到,当客户同组件动态地进行链接时,它必须将组件所在的 DLL 装载到内存中。在 Win32 环境下,这可以由 LoadLibrary[①] 函数完成。当使用完某个 DLL 之后,需要将其从内存中释放。将那些不再需要的组件放到内存中将是极为低效的。为此 COM 库实现了一个名为 CoFreeUnusedLibraries 函数,以释放那些不再需要的库所占用的内存空间。在程序的空闲期间,客户应周期性地调用 CoFreeUnusedLibraries 函数。

7.5.1 DllCanUnloadNow 的使用

那些 DllFreeUnusedLibraries 何以知道客户不再需要某个 DLL 了并可以将其从内存中释放呢? 为此 CoFreeUnusedLibraries 将调用 DllCanUnloadNow 以询问 DLL 它是否可以被卸载掉。DllCanUnloadNow 将告诉 COM 它是否仍在提供对任意对象的支持。若 DLL 不再提供任何组件了,那些 CoFreeUnusedLibraries 就可以将此 DLL 卸载掉。DLL 为决定它是否仍在提供对组件的支持,它将维护一个关于组件的计数值。为此需要在 CMPNT.CPP 中加入如下的代码:

```
static long g_cComponents = 0;
```

然后 IClassFactory::CreateInstance 或组件的构造函数可以将 g_cComponents 值增大,而组件的析构函数则可将此值减小。若 g_cComponents 的值为零,那些 DllCanUnloadNow 将给出肯定的回复。

7.5.2 LockServer

注意,使用上面的方法,统计的将只是 DLL 提供的组件,而不包括 DLL 提供的那些类厂。但这些类厂可能一直是被用着的。因此在统计组件时,统计类厂将更为合理。对于进程中的组件服务器,对类厂进行统计是可以的。但在第 10 章中我们将引入在 EXE 而非 DLL 中实现的本地服务器。从内部实现上讲,进程外服务器的启动与关闭与进程中服务器的启动与关闭是不同的。(当然对客户而言,这两种服务器并没有任何差别。)正是由于进程外服务器的启动方式,在不使进程外服务器永远不能释放其自身的情况下,我们将无法对类厂进行计数。关于这一点我们将在第 10 章中进行更为详细的讨论。要说明的是一个正在运行的类厂并不足以保证将某个服务器装载在内存中。

---

① COM 库实际上使用的是 CoLoadLibrary。此函数将调用 LoadLibrary。

这样一来就产生了一个问题。将某个拥有正在运行的类厂的 DLL 从内存中卸出将会给客户带来一些麻烦。假定客户中有一个指向某个正在运行的类厂的指针，在相应的 DLL 被卸载之后，若客户试图使用 IClassFactory 指针，那么显然将会出现错误。客户需要一种手段，以防止当它想在某个函数的作用域范围之外使用一个 IClassFactory 指针时 DLL 被从内存中卸载掉。IClassFactory::LockServer 的作用正在于此。它给客户提供了一种将服务器保存在内存中、直至使用完毕的方法。此时，客户只需调用 LockServer(TRUE)以锁住相应的服务器，并在使用完毕之后调用 LockServer(FALSE)将其解锁。

LockServer 的实现将是非常简单的，只需相应地增大或减小 g_cComponents 计数值即可。当然许多人，包括作者在内，喜欢对组件和加锁使用不同的计数值。此时，DllCanUnloadNow 应同时检测这两个计数值。

7.6 本章小结

大多数情况下，可以使用 CoCreateInstance 来创建所需的组件。但在某些情况下，CoCreateInstance 的灵活性是不够的。此时可以使用 CoGetClassObject 来获取对类厂及用于创建组件的接口的直接控制。创建组件的标准接口是 IClassFactory。CoCreateInstance 也将使用这一接口。

客户具体使用 CoCreateInstance 还是 CoGetClassObject 来创建组件都是无关紧要的。不论何种情况，组件都将具有一个相应的类厂。类厂实际上是创建组件的组件。某个组件的类厂一般而言都将实现 IClassFactory 接口。但某些在创建时具有一些特殊要求的组件将实现其他的接口，或者在实现了 IClassFactory 之外还实现其他的接口。

若不能将 Lego 积木块搭在一块，即使我们拥有一个室内塑料积木块工厂，这些积木块玩起来也不会有什么意思。积木块的制造只是本书所介绍的全过程的一部分。实际上更让人兴奋的事情是将这些积木块搭在一块。与此类似，也只有在将若干个组件组装在一起构成一个新的结构时才是真正令人感兴趣的。下一章我们将介绍如何根据已有的组件来建造新的组件。

第 8 章　组件复用:包容与聚合

有些杂志经常喜欢将 COM 同其他一些事物如富人、豆类植物以及其他公司的组件架构进行比较。这些杂志经常会包含一个如表 8-1 所示的表格。

表 8-1　COM 同其他事物的比较

特性	富人	豆类植物	C++	COM
可食用性	√	√	×	×
支持继承	√	√	√	×
可竞选总统	√	×	×	×

在上述表格中,读者可能不会注意到富人是可食用的这一情况,但却一定会注意到:在这些文章看来,COM 不支持继承。这些杂志似乎并没有意识到 COM 支持多态这一面向对象编程的最为重要的特性,或 COM 非常短小精悍、优美,或者 COM 可以透明地分布到网络上等特点。他们似乎也没有意识到 COM 是与语言无关的,并且目前已经开发出来的 COM 组件比其他任何类型的组件都要多得多这些事实。所有这些杂志关心的问题似乎只有一个,那就是继承性。

那么 COM 是否支持继承呢? 对这个问题的回答是:可以又不可以。上述那些杂志所讲的继承实际上指的是"实现继承",这种继承指的是类继承其基类的代码或实现。COM 并不支持此种类型的继承。但是 COM 却支持"接口继承",这种继承指的是类继承其基类的类型或接口。

COM 不支持实现继承导致了人们的许多指责,如有人说它是一种很差的技术,甚至有人因此而声称 COM 并不是面向对象的。这些争论使作者想起了 Internet 上那些关于 OS/2 和 Windows、vi 和 Emacs、Java 和 Python 等等问题的争论。但作者认为在这些问题上进行争论无异于是对时间的浪费。

COM 不支持实现继承的原因在于这种继承方式将使得一个对象的实现同另外一个对象的实现紧紧地关联起来。在这种情况下,当基类的实现被修改之后,派生类将无法正常运行而必须被修改。对于一个中等规模的 C++ 程序而言,这或许并不成为一个问题,因在这种情况下,我们一般能够获取所有的源代码,因此也将能够对派生类进行修改。但是对于较大的 C++ 程序而言,对受影响的派生类进行修改所花费的时间将是相当长的。更糟的是,我们可能根本无法得到所有的源代码。这就是为什么一些用 C++ 编写大型程序的专家们强烈建议人们基于抽象基类来构建应用程序。

抽象基类是一种最纯粹的接口继承,并且正好也被用来实现 COM 接口。由于任何人可以用任何语言在任何地方编写 COM 组件,因此必须有一种非常严格的方法来保证组件的客户不会因组件的变化而受到影响。显然实现继承无法提供客户所需的保护。

因此,为了保证对组件的修改不会影响应用程序的正常运行,COM 并不支持实现继承。但这并不会使 COM 的功能有任何损失,因我们可以用组件包容来完全模拟实现继承。当然实现继承比包容要方便一些。但作者认为大多数开发人员喜欢的可能是一个更为强壮的系统。

但这并不是实现继承在 COM 中就没有任何用处。在下一章中,我们将使用实现继承以避免为每一个组件实现 IUnknown。但在这一章中我们将讨论组件包容。

8.1 包容和聚合

美国人似乎很少有对他们所拥有的一切感到满意的。他们总是在想着如何将事情变得更好,所以他们总是在追逐着各种各样的事情。这或许是美国人的一个优点。对于组件人们也差不多是这样。当拿到一个组件之后,我们可能会想办法对之加以扩展或改造以使之符合自己的使用需要,并可能会希望用此改造后的组件来代替原有的组件。在 C++ 中,对类的改造是用包容和继承来实现的。在 COM 中,则可使用包容和聚合来对组件进行改造。

包容和聚合实际上是使一个组件使用另外一个组件的一种技术。对于这两个组件,作者分别将其称为是外部组件和内部组件。在包容的情况下,外部组件将包含内部组件;而在聚合的情况下,则称外部组件聚合内部组件。

8.1.1 包容简介

COM 包容同 C++ 包容是相似的。但在 COM 中,同其他内容类似,包容也是在接口级

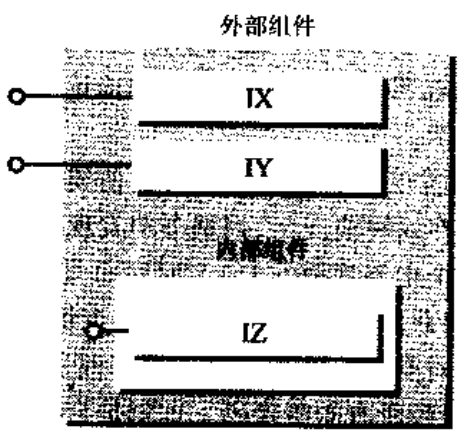


图 8-1 一个包含有一个内部组件并使用了其 IZ 接口的外部组件的内部结构

完成的。外部组件包含指向内部组件接口的指针。此时外部组件只是内部组件的一个客户，它将使用内部组件的接口来实现它自己的接口，参见图 8-1。

外部组件也可以通过将调用转发给内部组件的方法重新实现内部组件所支持的某个接口。并且外部组件还可以在内部组件代码的前后加上一些代码以对接口进行改造。关于这一点请参阅图 8-2。

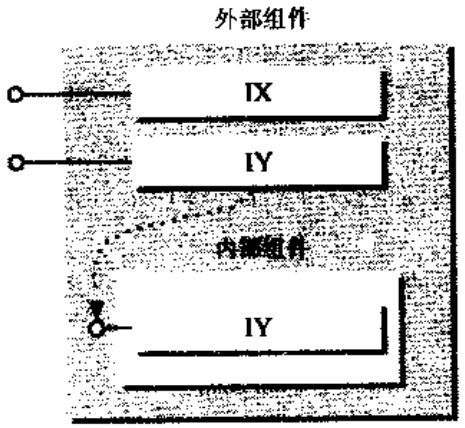


图 8-2 一个包含有一个内部组件并复用了其 IY 接口实现的外部组件的内部结构

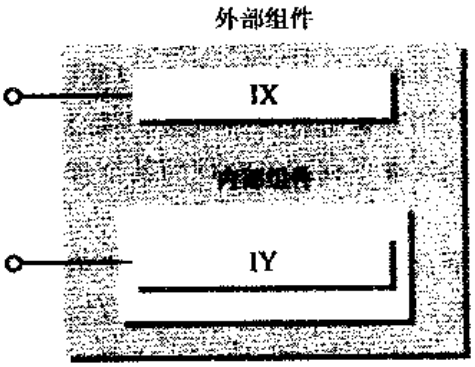


图 8-3 当外部组件聚合某个接口时，它将把接口指针直接传给客户，而并不像包容那样重新实现此接口并将函数调用转发给内部组件

8.1.2 聚合简介

聚合是包容的一个特例。当一个外部组件聚合了某个内部组件的一个接口时，它并没有像包容那样重新实现此接口并明确地将调用请求转发给内部组件。相反，外部组件将直接把内部组件的接口指针返回给客户。使用此种方法，外部组件将无需重新实现并转发接口中的所有函数。关于这一点请参阅图 8-3。但使用此种方式，外部组件将无法对接口中的函数进行任何改造。当外部组件将内部组件的接口指针返回给客户之后，客户就可以直接同内部组件打交道了。但此时客户不应该知道它是在同两个不同的组件交互，否则将无法满足封装的要求。这种使外部组件和内部组件显得像是同一组件的作法是成功地进行聚合的关键所在。下面我们将会看到，之所有会这样，正是使用 QueryInterface 的结果。

8.1.3 包容与聚合的比较

为清楚地说明包容和聚合之间的差别，我们可以考虑如下一种情况。假定一个金属制品商店的老板有两个雇员 Pemela 和 Angela。其中 Pemela 在这个商店已经工作了许多年了，她对整个商店的里里外外了如指掌。当有某项工作需要 Pemela 完成时，只需让客

户直接同 Pemela 打交道就可以了。但 Angela 则是一个新手,在金属制品行业没有什么太多的经验。当给 Angela 分派一项任务时,老板可能需要先启动此工作并将那些最为困难的部分完成。同时同客户打交道的工作也可能需要老板亲自完成。待 Angela 完工之后,老板可能需要同她一起检查一下她所完成的那一部分,并对某些不尽人意的地方加以修改。上面将工作交给 Pamela 的过程同聚合是类似的。当把工作分派给她时,老板就可以撒手不管了。但是对 Angela 而言,仍然需要老板完成同客户的交互。在 Angela 开始工作之前和完成之后,老板需要对此进行一些必要的检查。这一点同包容是类似的。

对于聚合的每一个接口,可能会包容成百上千的其他接口。这是因为当组件作为客户去使用其他组件的接口时,实际就可以看成是一种包容。聚合则更为特殊。当某个组件已经实现了我们待实现的某个接口时,就可以使用聚合,即将此组件的接口指针传给客户。

聚合和包容可以混合起来使用。一个组件可以扩展或改造许多不同组件所实现的多个接口。它可以在聚合某些接口的同时包容另外一些接口。

由于聚合是包容的一个特例,下面我们将先讨论包容然后再讨论聚合。

8.2 包容的实现

COM 组件的包容同 COM 组件的使用同样简单。在下面给出的示例代码中,组件 1(Component1)是一个实现了两个接口 IX 和 IY 的外部组件。但它复用了组件 2——组件 1 所包容的一个内部组件——对 IY 的实现,这一点同图 8-2 所示是完全相同的。

此例中客户和内部组件同前一章中给出的客户和组件基本上是相同的。当一个组件包容另外一个组件时,客户和组件都不需要完成任何其他工作,甚至它们都不必知道他们是在使用包容。

程序清单 8-1 显示了组件 1 的声明及大部分实现。清单中那些同包容相关的代码被突出出来了。其中的新成员变量 m_pIY 中保存的是指向被包容的组件(Component 2)中 IY 接口的指针。

程序清单 8-1 Component 1 创建包容组件的一个实例并保存指向类 IY 接口的指针。

```
Containment Code in CONTAIN \ CMPNT1
////////////////////////////////////////////////////////
//
// Component A
//
class CA : public IX,
           public IY
{
```

```
public:
    // IUnknown
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
    virtual ULONG   __stdcall AddRef();
    virtual ULONG   __stdcall Release();

    // Interface IX
    virtual void __stdcall Fx() { cout << "Fx" << endl; }

    // Interface IY
    virtual void __stdcall Fy() { m_pIY->Fy(); }

    // Constructor
    CA();

    //Destructor
    ~CA();

    // Initialization function called by the class factory.
    // and used to create the contained component
    HRESULT Init();

private:
    // Reference count
    long m_cRef;

    // Pointer to the contained component's IY interface.
    IY* m_pIY;
};
//
//Constructor
//
CA::CA()
: m_cRef(1),
  m_pIY(NULL)
{
  InterlockedIncrement(&g_cComponents);
}
//
//Destructor
//
CA::~CA()
{
  InterlockedDecrement(&g_cComponents);
  trace("Destroy self.");
  // Release contained component.
```

```
    if (m_pIY! = NULL)
    {
        m_pIY->Release();
    }
}
// Initialize the component by creating the contained component.
HRESULT CA::Init()
{
    trace("Create Contained component.");
    HRESULT hr = ::CoCreateInstance( CLSID_Component2,
                        NULL,
                        CLSCTX_INPROC_SERVER,
                        IID_IY,
                        (void**)&m_pIY);
    if (FAILED(hr))
    {
        trace("Could not create contained component.");
        return E_FAIL;
    }
    else
    {
      return S_OK;
    }
}
```

下面我们来看一下外部组件(Component1)是如何工作的。在 Component1 的实现中加入了一个新的成员 Init。此成员将按通常的方法调用 CoCreateInstance 来创建内部组件(Component2)。在此函数调用中,外部组件请求了内部组件 IY 接口的指针。若此调用成功此接口将被保存在 m_pIY 中。

在下面的代码中并没有给出 QueryInterface 及其他 IUnknown 函数的实现。这些函数实际上同没有使用包容的情况是一样的。当客户向 Component 1 请求 IY 接口时, Component 1 将返回其 IY 接口的一个指针。当客户调用 Component 1 中的 IY 接口时,Component 1 将把这些调用请求转发给 Component2。这一点是通过如下的代码实现的。

```
virtual void Fy() { m_pIY->Fy(); }
```

当 Component 1 将其自己销毁时,它将调用 m_pIY 上的 Release,以使 Component 2 将自己从内存中释放。

Component 1 的类厂同上一章中给出的类厂略有不同。所需的修改是在 CreateInstance 的实现中,在创建好 Component 1 之后加上对 Component 1 的 Init()的调用。关于此请参阅程序清单 8-2。

程序清单 8-2 外部组件的类厂调用新创建组件上的 Init()函数。

```
CreateInstance Function Code in CONTAIN \ CMPNT1
HRESULT_stdcallCFactory::CreateInstance(IUnknown* pUnknownOuter,
                                        const IID& iid,
                                        void** ppv)
{
  // Cannot aggregate
  if (pUnknownOuter != NULL)
  {
     return CLASS_E_NOAGGREGATION;
  }
  // Create component.
  CA* pA = new CA;
  if (pA == NULL)
  {
    return E_OUTOFMEMORY;
  }
  // Initialize the component.
  HRESULT hr = pA->Init();
  if (FAILED(hr))
  {
     // Initialization failed. Delete component.
     pA->Release();
     return hr;
  }
  // Get the requested interface.
  hr = pA->QueryInterface(iid, ppv);
  pA->Release();
  return hr;
}
```

上面就是实现包容所需的所有代码。下面我们来看一下如何使用包容。

8.2.1 接口扩展

包容的一个主要用途是通过给已有接口加上一些代码以扩展此接口。下面我们来看一个例子。假定想把一个 IAirplane 接口扩展成一个 IFloatPlane 接口。下面是这些接口的定义：

```
interface IAirplane : IUnknown
{
  void TakeOff();
```

```
    void Fly() ;
    void Land() ;
};
interface IFloatPlane : IAirplane
{
    void LandingSurface(UINT iSurfaceType) ;
    void Float() ;
    void Sink() ;
    void Rust() ;
    void DrainBankAccount() ;
};
```

假定在一个名为 MyAirplane 的组件中已经实现了 IAirplane 接口。此时外部组件可以方便地包容 MyAirplane 并使用 IAirplane 接口来实现 IFloatPlane 所继承的 IAirplane 的成员函数:

```
void CMyFloatPlane::Fly()
{
    m_pIAirplane->Fly() ;
};
```

在某些情况下可能需要修改 IAirplane 中的其他成员函数以支持从水上降落和起飞:

```
void CMyFloatPlane::Land()
{
    if (m_iLandingSurface == WATER)
    {
        WaterLanding() ;
    }
    else
    {
        m_pIAirplane->Land() ;
    }
}
```

可以看到对这些类使用包容是非常简单的。但是若 IAirplane 接口定义了大量的成员函数,此时编写那些转发调用请求的代码可能是一项极为繁重的工作。但幸运的是这并不是一项维护性的工作,因接口在发布之后将不会发生变化。在这种情况下,使用聚合将把那些不想编写把调用请求转发给内部组件的程序员解脱出来。但使用聚合的一个不便之处在于无法在接口中加上自己的代码。关于聚合我们将在下面详细讨论。

8.3 聚合的实现

首先让我们简要描述一下聚合是如何实现的。假定客户向外部组件请求接口 IY。

此时外部组件可以不实现 IY 接口，而只需将内部组件请求查询此 IY 接口并将此接口指针返回给客户。客户可以直接使用此指针来调用内部组件所实现的那些 IY 成员函数。此时就 IY 接口而言，外部组件相当于是被架空了：它放弃了对 IY 接口的控制而将此控制交给了内部组件。

虽然聚合看起来是非常简单的。但正确地实现内部组件的 IUnknown 接口却有一些困难。下面我们将会看到这一点。解决这一问题的关键在于 QueryInterface 函数。下面我们将给出外部组件的 QueryInterface 实现以使之返回内部组件的接口指针。

C++和聚合

在 C++ 中并没有同聚合等价的特性。聚合实际上是继承性的一个动态形式，而 C++ 的继承则总是静态的。在 C++ 中模拟聚合的最好的方法是重载 -> 操作符。下一章我们在实现智能指针时将详细介绍这一技术。但重载 -> 操作符同 COM 聚合相比仍有许多限制。例如它只能将调用请求转发给某一个类，而在 COM 中则可以聚合任意数目的接口。

8.3.1 QueryInterface 的实现

下面是实现了接口 IX 并通过聚合提供 IY 接口的一个外部组件的声明。

```
class CA : public IX
{
public:
  // IUnknown
  virtual HRESULT _stdcall QueryInterface(const IID& iid,
                                          void** ppv);
  virtual ULONG _stdcall AddRef();
  virtual ULONG _stdcall Release();

  // Interface IX
  virtual void _stdcall Fx() { cout << "Fx" << endl; }

  // Constructor
  CA();
  //Destruclor
  ~CA();

  // Initialization function called by the class factory
  // to create contained component.
  HRESULT Init();

private:
  // Reference Count
  long m_cRef;
```

```
    // Pointer to inner component's IUnknown
    IUnknown* m_pInnerUnknown;
};
```

从上面的声明中似乎看得出来此组件支持接口 IY:它既没有继承 IY,也没有为接口 IY 实现任何成员。在这里外部组件实际上使用的是内部组件对接口 IY 的实现。这一点是在其 QueryInterface 函数中完成的。此函数可以返回指向内部组件的一个接口的指针。在下面的代码中,成员变量将包含内部组件 IUnknown 接口的地址。

```
HRESULT _stdcall CA::QueryInterface(const IID& iid,
                                    void** ppv)
{
    if (iid == IID_IUnknown)
    {
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IX)
    {
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IY)
    {
        return m_pUnknownInner->QueryInterface(iid,ppv);
    }
    else
    {
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}
```

在上例中外部组件的 QueryInterface 只是调用了内部组件的 QueryInterface。这一点看起来是非常漂亮的,也是非常简单的,但不幸的是它却无法正常工作。

问题并不在于上面给出的代码,而在于内部组件的 IUnknown 接口。当某个组件被聚合时,它必须按照一种特殊的方式来处理 IUnknown 的成员调用。我们将看到它实际上需要实现两个 IUnknown 实现。

下面我们先讨论一下为什么按通常的方式来实现内部组件的 IUnknown 接口会引起问题。然后我们再介绍如何实现两个 IUnknown 来解决这个问题,以及如何修改内部组件的创建以使两个组件都能获取它们所需的指针。最后我们将介绍如何给外部组件提供指向内部组件接口的指针。

8.3.2 不正确的 IUnknown

聚合的目标就是使客户确信内部组件所实现的某个接口是由外部组件实现的。此时需要从内部组件中将一个指针直接传给客户并使客户相信此接口指针是属于外部组件的。若将内部组件按通常方式实现的接口指针传给客户，那么客户得到的是对于组件的一个分裂的视图。也就是说内部组件的接口将调用内部组件所实现的 QueryInterface，而外部组件则有其自己的 QueryInterface。当客户查询属于内部组件的接口时，它所获得的关于组件的功能视图与它查询外部组件的接口时是不同的。关于这一点请参阅图 8-4。

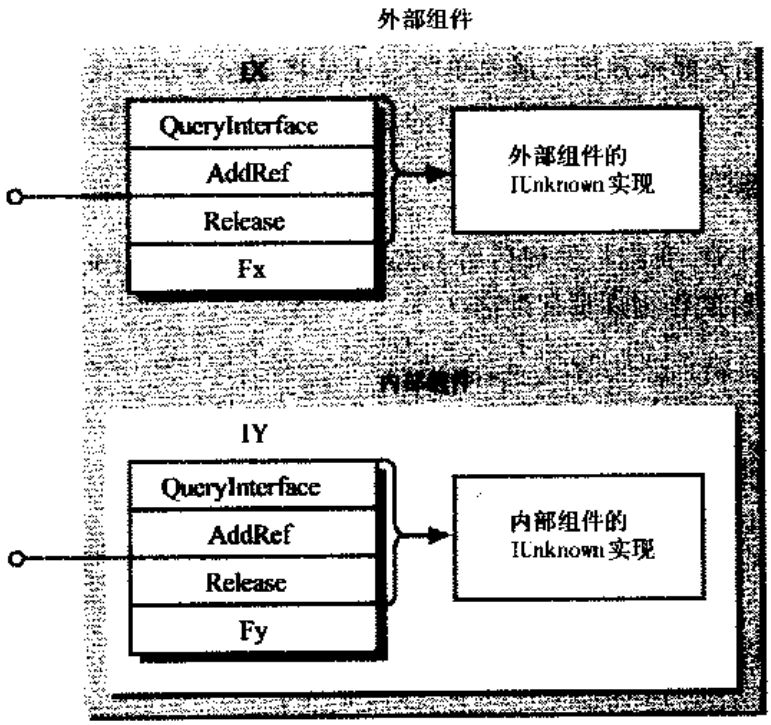


图 8-4 不同的组件具有不同的 IUnknown 实现

用一个例子来说明这个问题可能会更加清楚。假定有某个聚合组件。其中外部组件支持接口 IX 和 IY。它自己实现了 IX 接口，而 IY 接口是聚合得到的。内部组件实现了 IY 和 IZ 接口。在创建了外部组件之后，我们将可以看到其 IUnknown 接口指针。用此接口指针可以成功地查询 IX 或 IY，但查询 IZ 时将返回 E_NOINTERFACE。当查询 IY 接口时，得到的实际上将是内部组件的 IY 接口指针。若使用此 IY 指针查询 IZ，此查询将会成功。这是由于 IY 接口的 IUnknown 函数是由内部组件实现的。与此类似，通过 IY 接口查询 IX 将会失败，这是由于内部组件并不支持 IX。这样一来，关于 QueryInterface 实现的一条基本规则，即“若能够从某接口获取某特定接口，则从任意接口都将能够获取此接口”。

问题的根源就在于内部组件的 IUnknown 接口。客户实际上可以得到两个不同的 IUnknown接口，即内部组件的和外部组件的。这将给客户带来一些混乱，因为每一个 IUn-

known 均将实现一个 QueryInterface,而每一个 QueryInterface 均将分别支持一个不同的接口集。但客户应完全独立于聚合组件的实现,它不应该知道外部组件聚合了某个内部组件,并且永远不应看到内部组件的 IUnknown。在第 3 章中我们介绍过,两个不同的接口当且仅当它们返回相同的 IUnknown 指针时才会被认为是由同一组件实现的。因此,应将内部组件 IUnknown 接口向客户隐藏起来,而只给它提供一个 IUnknown 接口。内部组件上的接口必须使用外部组件所实现的 IUnknown 接口。此时外部组件的 IUnknown 接口将被称作是外部未知接口或控制未知接口。

8.3.3 聚合的未知接口

内部组件使用外部未知接口最简单的方法是将调用请求转发给外部未知接口。为此,内部组件需要一个指向外部未知接口的指针。并且它还需要知道它是被聚合的。

一、外部未知接口

回想一下第 7 章,我们将发现传给 CoCreateInstance 和 IClassFactory::CreateInstance 的一个 IUnknown 指针的作用还没有讨论:

```
HRESULT _stdcall CoCreateInstance(
    const CLSID& clsid,
    IUnknown * pUnknownOuter, // Outer Component
    DWORD dwClsContext,  // Server Context
    const IID& iid ,
    void * * ppv
  );

HRESULT _stdcall CreateInstance(
    IUnknown * pUnknownOuter,
    const IID& iid,
    void** ppv
  );
```

外部接口可以使用 pUnknownOuter 参数给内部组件传递其 IUnknown 接口的指针。若此外部未知接口指针非空,那么此时表示我们想进行组件的聚合。使用传给 CreateInstance 的 IUnknown 接口指针,被创建的组件将知道它是被聚合的,并且可以知道是谁在进行聚合。对于未被聚合的组件,它可以使用其自己的 IUnknown 实现。对于被聚合的组件,它应将相应的接口调用转发给外部未知接口。

二、代理和非代理未知接口

为支持聚合,内部组件实际上将实现两个 IUnknown 接口。其中的非代理未知接口将按通常的方式实现内部组件的 IUnknown 接口。而代理未知接口将把 IUnknown 成员函数

调用转发给外部未知接口或非代理未知接口。若内部组件没有被聚合，那么代理未知接口将这些调用转发给非代理未知接口。若内部组件被聚合了，那么代理未知接口将把调用请求转发给由外部组件所实现的外部未知接口。聚合组件的客户将调用代理未知接口，而外部组件将通过非代理未知接口来操作内部组件。对此我们可以用图示来加以说明。如图 8-5 所示为非聚合的情形，而图 8-6 则显示了聚合的情形。

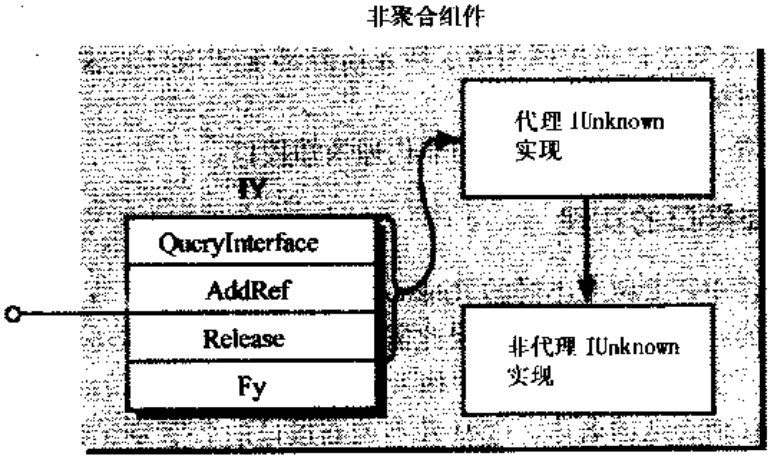


图 8-5 当组件未被聚合时，它将把 IUnknown 调用请求转发给其非代理未知接口

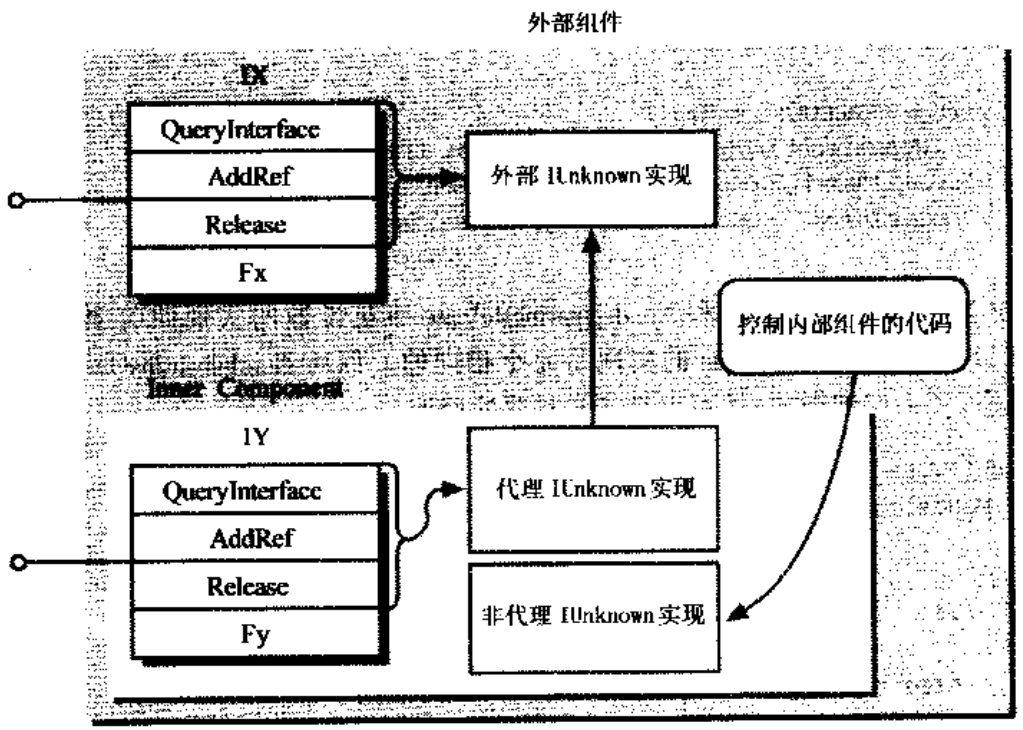


图 8-6 当组件被聚合时，它将把 IUnknown 调用请求转发给外部未知接口

在图8-5中,可以看到IY的IUnknown部分将调用代理未知接口的实现。而代理未知接口实现将调用非代理未知接口的实现。非代理未知接口的实现可以按常规的方式完成。

图8-6显示的是一个聚合了IY接口的组件。在被聚合的情况下,代理未知接口将调用外部组件的IUnknown实现。外部组件可以调用内部组件的非代理未知接口来控制内部组件的生命期。因此当某个组件调用IY接口的IUnknown成员函数时,它将调用代理未知接口,此代理未知接口将把调用请求转发给外部未知接口。这样做的结果是内部组件使用了外部组件的IUnknown实现。

下面我们来看一下如何实现代理和非代理未知接口。

三、非代理未知接口的实现

在我们的组件中需要两个不同的IUnknown接口实现。但C++不允许在同一个类中完成同一个接口的两个不同的实现。因此需要修改某一个IUnknown接口的名字以便这两个接口的名字不会出现冲突。本书中作者选择的名字为INondelegatingUnknown,当然读者可以自行选择任何合适的名称。COM并不关心接口的名字是什么,而只关心vtbl的结构。INondelegatingUnknown的声明同IUnknown是相同的,只是在所有成员函数的名称前都加上了前缀Nondelegating。

```
struct INondelegatingUnknown
{
    virtual HRESULT _stdcall
        NondelegatingQueryInterface(const IID&, void**) = 0;
    virtual ULONG _stdcall NondelegatingAddRef() = 0;
    virtual ULONG _stdcall NondelegatingRelease() = 0;
};
```

其中NondelegatingAddRef和NondelegatingRelease成员函数的实现同前面给出的IUnknown接口中AddRef和Release的实现是完全相同的。但对NondelegatingQueryInterface的实现却进行了一些细微的、然而也是非常重要的修改。

```
HRESULT _stdcall
CB::NondelegatingQueryInterface(const IID& iid,
                                void** ppv)
{
  if (iid == IID_IUnknown)
  {
     *ppv = static_cast<INondelegatingUnknown*>(this);
  }
  else if (iid == IID_IY)
  {
     *ppv = static_cast<IY*>(this);
```

```
    }
    else
    {
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    reiterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}
```

注意,在上面的代码中我们将内部组件的 this 指针转换成了一个 INondelegatingUnknown 指针。这种转换是非常重要的。通过这一转换,我们可以保证返回的是一个非代理的未知接口指针。当向非代理接口指针查询 IID_IUnknown 接口时,它返回的将总是一个指向其自身的指针。如果不进行此种转换,则返回指针将是指向代理未知接口的。当此组件被聚合时,代理未知接口将把所有的 QueryInterface、AddRef 以及 Release 调用转发给外部组件。

被聚合组件的客户将永远也不会获得内部组件的非代理接口指针。当客户请求 IUnknown 接口指针时,它得到的将是外部组件的 IUnknown 指针。内部组件的非代理未知接口指针只能由其外部组件获取。下面我们看一下如何实现代理未知接口。

四、代理未知接口的实现

代理未知接口的实现实际上是非常简单的:它只需将相应的调用请求转发给外部未知接口或非代理未知接口即可。下面我们将给出一个支持聚合的组件的声明。此组件中包含一个名为 m_pUnknownOuter 的指针。当此组件被聚合时,此指针将指向外部未知接口。当此组件未被聚合时,此指针将指向非代理未知接口。但代理未知接口中的成员函数被调用时,此调用将被转发到 m_pUnknownOuter 指向的接口。代理未知接口是以内联方式实现的。

```
class CB : public IY,
           public INondelegatingUnknown
{
public:
    // Delegating IUnknown
    virtual HRESULT _stdcall
        QueryInterface(const IID& iid, void** ppv)
    {
        //Delegate QueryInterface.
        return m_pUnknownOuter->QueryInterface(iid, ppv);
    }
    virtual ULONG _stdcall AddRef()
    {
```

```
    //Delegate AddRef.
    return m_pUnknowOuter->AddRef();
  }
  virtual ULONG __stdcall Release()
  {
    //Delegate Release.
    return m_pUnknowOuter->Release();
  }
  // Nondelegating IUnknown
  virtual HRESULT __stdcall
    NondelegatingQueryInterface(const IID& iid, void** ppv);
  virtual ULONG __stdcall NondelegatingAddRef();
  virtual ULONG __stdcall NondelegatingRelease();
  // Interface IY
  virtual void __stdcall Fy() { cout << "Fy" << endl; }
  // Constructor
  CB(IUnknown* m_pUnknowOuter);
  //Destructor
  ~CB();
private:
  long m_cRef;
  IUnknown* m_pUnknowOuter;
};
```

8.3.4 内部组件的创建

现在我们对如何实现内部组件已经非常清楚了。下面我们来看一下外部组件如何创建内部组件。下面我们将通过对如下三个函数的讨论来看一下内部组件创建的全过程,即外部组件的 Init 函数(创建过程实际上就是由这个函数开始的)。内部组件类厂的 CreateInstance 函数以及内部组件的构造函数。

一、外部组件的 Init 函数

在聚合一个组件时,外部组件要做的第一步是创建内部组件。我们知道组件聚合和包容的主要区别是对组件的聚合而言的,外部组件将把外部未知接口的指针传给内部组件。下面给出的代码显示了外部组件是如何创建内部组件的。注意,传给 CoCreateInstance 的第二个参数是指向外部组件 IUnknown 接口的指针。

注意第四个参数请求的是内部组件的 IUnknown 接口指针。类厂返回的将是指向内部组件的非代理 IUnknown 的指针。前面我们已经看到,外部组件需要此指针将 QueryInterface 调用转发给内部组件。因此,外部组件必须在此时请求此 IUnknown 指针,否则它将

永远也无法得到此指针。并且外部组件还需要将此指针暂存起来以备后用。

另外注意在本例中,我们并不需要明确地将 this 指针转换成一个 IUnknown 指针,因为 CA 只是继承了 IX,因此隐式的转换并不会有任何歧义。

```
HRESULT_stdcall
CA::Init()
{
  IUnknown* pUnknowOuter = this;
  HRESULT hr = CoCreateInstance(CLSID_Component2,
                                pUnknownOuter,
                                CLSCTX_INPROC_SERVER,
                                IID_IUnknown,
                                (void**)&m_pUnknownInner);
  if (FAILED(hr)
  {
     return E_FAIL;
  }
  return S_OK;
}
```

外部组件的 IClassFactory::CreateInstance 将调用 CA::Init。其 IClassFactory 实现将仍然保持不变。但内部组件的类厂却需要进行一些修改。关于这一点我们将在下面讨论。

二、内部组件的 IClassFactory::CreateInstance 函数

在被聚合的情况下,内部组件的 IClassFactory 组件必须使用 INondelegatingUnknown 接口而不能再使用 IUnknown。下面的代码给出了内部组件 IClassFactory::CreateInstance 的实现,其中与前面的 CreateInstance 实现中有差别的地方用粗体字标示出来了。注意当 pUnknownOuter 非空时(此表示外部组件想进行聚合),IClassFactory::CreateInstance 并不会失败返回。但当 iid 不是 IID_IUnknown 时,CreateInstance 必须失败。当一个组件被聚合时,此内部组件将只能返回一个 IUnknown 接口,这是由于外部组件在其他时候无法获取非代理未知接口的指针(因 QueryInterface 调用将被转发到外部未知接口)。

```
HRESULT_Stdcall
CFactory::CreateInstance(IUnknown* pUnknowOuter,
                         const IID& iid,
                         void** ppv)
{
  // To aggregate,iid must be IID_IUnknow.
  if ((pUnknowOuter != NULL) && (iid != IID_IUnknown))
  {
     return CLASS_E_NOAGGREGATION;
  }
```

```
    // Create component.
    CB* pB = new CB(pUnknownOuter);
    if (pB == NULL)
    {
        return E_OUTOFMEMORY;
    }

    // Get the requested interface.
    HRESULT hr = pB->NondelegatingQueryInterface(iid, ppv);
    pB->NondelegatingRelease();
    return hr;
}
```

上面代码中的 CreateInstance 函数将调用 NondelegatingQueryInterface 而非 QueryInterface 来获取新创建的内部组件中客户所请求的接口。当内部组件被聚合时,它将把 QueryInterface 调用转发给外部未知接口。类厂需要返回一个指向非代理未知接口的指针,因此,它调用的是 NondelegatingQueryInterface。

三、内部组件的构造函数

在上面的代码中,CreateInstance 将把外部未知接口的指针传给内部组件的构造函数。此构造函数将初始化 m_pUnknownOuter。此成员变量将被代理未知接口将一些函数调用转发给非代理接口或外部未知接口。若此组件未被聚合(pUnknownOuter 为 NULL),此构造函数将把 m_pUnknownOuter 设成为指向非代理未知接口。如下面的代码所示:

```
CB::CB(IUnknown* pUnknowOuter)
: m_cRef(1)
{
    ::InterlockedIncrement(&g_cComponents);

    if (pUnknowOuter == NULL)
    {
        // Not being aggregated; use nondelegating unknown.
        m_pUnknowOuter = reinterpret_cast<IUnknown*>
                         (static_cast<INondelegatingUnknown*>
                         (this));
    }
    else
    {
        // Being aggregated; use outer unknown.
        m_pUnknowOuter = pUnknowOuter;
    }
}
```

8.3.5 外部组件中指向内部组件接口的指针

在创建内部组件的第一步 CA::Init 的实现中,所请求的是 IUnknown 接口而非 IY 接口。但外部组件实际上想聚合的却是 IY。因此,在继承进行下面的处理之前,最好先检测一下内部组件是否真的实现了 IY 接口。前面我们已经讲过,当聚合一个组件时,外部组件只能请求被聚合组件的 IUnknown 接口。若传给 CFactory::CreateInstance 的接口标识符不是 IID_IUnknown,则此函数将返回 CLASS_E_NOAGGREGATION。因此需要在创建了内部组件之后再请求其 IID_IY 接口。

但此时别忘了前面我们提到过的一个问题。当通过 m_pUnknownInner 调用 QueryInterface以获取一个 IID_IY 接口指针时,它将调用此接口的 AddRef 函数。由于内部组件是被聚合的,因此它将把此 AddRef 调用转发给外部组件。这样实际的结果将是外部组件的引用计数被增大了,而内部组件的引用计数则没有变化。为突出起见,在此我们再将这一点重复一遍:当外部组件查询内部组件的非代理未知接口或其他任何接口时,被增大的将是外部组件的引用计数。这一点在客户请求内部组件的接口时正是我们所期望的结果。但是现在外部组件有一个 IY 接口指针,并且外部组件的引用计数值将因此指针而被增大。如果允许这样的话,外部组件的引用计数将永远也不会为 0,因此也将永远不会被从内存中释放掉。

由于外部组件中 IY 接口指针的生命期是嵌套在外部组件的生命期内的,因此我们并不需要此接口进行引用计数。但是也不要调用 IY 的 Release 函数来减小引用计数值,因为对 IY 接口的处理也必须像它是被单独进行引用计数那样。(对于我们例子中给出的内部组件的实现,到底释放哪个指针是无关紧要的。但对于其他的实现这一点可能是至关重要的。)释放 IY 接口可能会导致系统释放它所占用的资源。因此当调用 Release 时,应使用传给 CoCreateInstance 的那个指针。下面给出的是请求 IY 接口的 CA::Init 函数的实现。

```
HRESULT _stdcall CA::Init()
{
  // Get pointer to outer unknown
  IUnknown* pUnknowOuter = this;

  // Create inner component.
  HRESULT hr
     = ::CoCreateInstance(CLSID_Component2,
        pUnknowOuter, // Outer component's IUnknown
        CLSCTX_INPROC_SERVER,
        IID_IUnknown, // IUnknown when aggregating.
        (void**)&m_pUnknownInner);
  if (FAILED(hr))
  {
```

```
        // Could not create contained component.
        return E_FAIL ;
    }

    // QI increments the reference count on the outer component.
    // Get the IY interface from the inner component.
    hr = m_pUnknownInner->QueryInterface(IID_IY,(void**)&m_pIY) ;
    if (FAILED(hr))
    {
        //Inner component does not support interface IY.
        m_pUnknownInner->Release() ;
        return E_FAIL ;
    }

    // We need to release the reference count added
    // to the outer component in the above call.
    pUnknownOuter->Release() ;
    return S_OK ;
}
```

在实现 QueryInterface 时可以有两种不同的选择:可返回 m_pIY 或调用内部组件的 QueryInterface 函数。所以既可以使用前面我们已经看到过的代码:

```
else if (iid == IID_IY)
{
    return m_pUnknownInner->QueryInterface(iid, ppv);
}
```

也可以使用如下的代码:

```
else if (iid == IID_IY)
{
    *ppv = m_pIY ;
}
```

到此为止,我们介绍了如何创建内部组件、如何查询其某个接口、如何修改引用计数以及如何将接口指针返回给客户。剩下来还有一件未做的事情是在外部组件的析构函数中释放此接口。由于此接口并没有被进行引用计数,因此不能简单的调用 p_IY->Release。在外部组件的 Init 函数中,当获取了 IY 接口之后,我们将此引用计数去掉了。现在我们需要将这个过程倒过来,即恢复引用计数并调用 IY 接口上的 Release 函数。但此时应该小心,否则最后一个 Rleease 调用将使外部组件的引用计数变成 0,从而会导致此组件试图再次将自己释放。

因此,释放外部组件中指向内部组件的接口指针将是一个三步的过程。首先需要保证组件不会试图再次将其自己释放。其次,需要对外部组件调用 AddRef,这是由于对内部组件的 Release 调用将会导致对外部组件的 Release 调用。最后才可以将外部组件释

放。这个过程可以用下面的代码表示：

```
// 1. Increment the reference count to avoid
// recursive calls to Release.
m_cRef = 1;

// 2. AddRef the outer unknown.
IUnknown * pUnknownOuter = this;
pUnknownOuter->AddRef();

// 3. Release the interface.
m_pIY->Rleease();
```

下面我们来想象一下上面的代码是如何工作的。首先我们将把引用计数设为 1。其次是将此计数值增大为 2。然后我们将调用接口 IY 的 Release。此时内部组件将把此 Release 调用转发给外部组件。在此调用之后，外部组件的引用计数值将从 2 变成为 1。因此若没有将引用计数的初值设成为 1，在调用 Release 之后将会导致外部组件试图将其释放。

在我们的实现中，当内部组件被聚合时，对其 Release 函数的调用将被转发给外部未知接口。但是外部组件对内部组件的处理应象内部组件的每一个接口都被进行引用计数一样，因内部组件的其他实现可能不仅仅是将 Release 调用转发给外部组件，它还可以释放一些资源或进行其他一些操作。

上面的许多代码似乎显得有些多余。但当外部组件本身被聚合时，按照上述步骤进行就显得非常有必要了。在第 9 章中我们将给出一个组件聚合的例子，在该例中，被聚合的组件本身还聚合了其他的组件。上面我们给出的过程虽然繁琐了一些，但一旦完成了这个过程之后，几乎就可以“一劳永逸”了。

8.4 一个完整的例子

下面我们将实现一个聚合了一个接口的组件。在这个例子中，组件 1 支持两个接口 IX 和 IY。但它只实现了接口 IX，而没有实现接口 IY，并且也并没有将此接口调用转发给组件 2 的实现。相反，当客户向组件 1 查询接口 IY 时，组件 1 将返回内部组件即组件 2 所实现的 IY 接口的指针。程序清单 8-3 给出的是外部组件的实现，程序清单 8-4 则是内部组件的实现。客户基本上没有什么变化，因它并不关心我们使用的是包容还是聚合。

程序清单 8-3 外部组件(聚合组件)的实现。

```
//
// Cmpnt1.cpp - Component 1
//
//
#include <iostream.h>
```

```
#include <objbase.h>
#include "Iface.h"
#include "Registry.h"
void trace(const char* msg) { cout << "Component 1: \t" << msg << endl ;}
//////////////////////////////////////////////////////////
//
// Global variables
//
// Static variables
static HMODULE g_hModule = NULL ;      // DLL module handle
static long g_cComponents = 0 ;        // Count of active components
static long g_cServerLocks = 0 ;       // Count of locks
// Friendly name of component
const char g_szFriendlyName[]
      = "Inside COM, Chapter 8 Example 2, Component 1" ;
// Version-independent ProgID
const char g_szVerIndProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt1" ;
// ProgID
const char g_szProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt1.1" ;
//////////////////////////////////////////////////////////
//
// Component A
//
class CA : public IX
      // public IY
{
public:
      // IUnknown
      virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv) ;
      virtual ULONG   __stdcall AddRef() ;
      virtual ULONG   __stdcall Release() ;

      // Interface IX
      virtual void __stdcall Fx() { cout << "Fx" << endl ;}

      /* Component1 aggregates instead of implementing interface IY.
      // Interface IY
      virtual void __stdcall Fy() { m_pIY->Fy() ;}
      */
      // Constructor
```

```
    CA() ;
    // Destructor
    ~CA() ;
    // Initialization function called by the class factory
    // to create contained component
    HRESULT __stdcall Init() ;

private:
    // Reference count
    long m_cRef ;
    // Pointer to the aggregated component's IY interface
    // (We do not have to retain an IY pointer. However, we
    // can use it in QueryInterface.)
    IY * m_pIY ;
    // Pointer to inner component's IUnknown
    IUnknown * m_pUnknownInner ;
} ;

//
// Constructor
//
CA::CA()
: m_cRef(1),
  m_pUnknownInner(NULL)
{
    InterlockedIncrement(&g_cComponents) ;
}

//
// Destructor
//
CA::~CA()
{
    InterlockedDecrement(&g_cComponents) ;
    trace("Destroy self.") ;
    // Prevent recursive destruction on next AddRef/Release pair.
    m_cRef = 1 ;
    // Counter the pUnknownOuter->Release in the Init method.
    IUnknown * pUnknownOuter = this ;
    pUnknownOuter->AddRef() ;
    // Properly release the pointer; there may be per-interface
    // reference counts.
    m_pIY->Release() ;
    // Release contained component.
```

```
    if (m_pUnknownInner != NULL)
    {
        m_pUnknownInner->Release();
    }
}

// Initialize the component by creating the contained component.
HRESULT __stdcall CA::Init()
{
    // Get the pointer to the outer unknown.
    // Since this component is not aggregated, the outer unknown
    // is the same as the this pointer.
    IUnknown* pUnknownOuter = this;
    trace("Create inner component.");
    HRESULT hr =
        ::CoCreateInstance(CLSID_Component2,
                           pUnknownOuter, // Outer component's IUnknown
                           CLSCTX_INPROC_SERVER,
                           IID_IUnknown,  // IUnknown when aggregating
                           (void**)&m_pUnknownInner);
    if (FAILED(hr))
    {
        trace("Could not create contained component.");
        return E_FAIL;
    }

    // This call will increment the reference count on the outer component.
    trace("Get the IY interface from the inner component.");
    hr = m_pUnknownInner->QueryInterface(IID_IY, (void**)&m_pIY);
    if (FAILED(hr))
    {
        trace("Inner component does not support interface IY.");
        m_pUnknownInner->Release();
        return E_FAIL;
    }

    // We need to release the reference count added to the
    // outer component in the above call.  So call Release
    // on the pointer you passed to CoCreateInstance.
    pUnknownOuter->Release();
    return S_OK;
}


//
// IUnknown implementation
//
HRESULT __stdcall CA::QueryInterface(const IID& iid, void** ppv)
```

```
{
    if (iid == IID_IUnknown)
    {
        *ppv = static_cast<IUnknown*>(this);
    }
    else if (iid == IID_IX)
    {
        *ppv = static_cast<IX*>(this);
    }
    else if (iid == IID_IY)
    {
        trace("Return inner component's IY interface.");
#if 1
        // You can query for the interface.
        return m_pUnknownInner->QueryInterface(iid,ppv);
#else
        // Or you can return a cached pointer.
        *ppv = m_pIY;
        // Fall through so it will get AddRef'ed
#endif
    }
    else
    {
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}
ULONG __stdcall CA::AddRef()
{
    return InterlockedIncrement(&m_cRef);
}
ULONG __stdcall CA::Release()
{
    if (InterlockedDecrement(&m_cRef) == 0)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}
```

```
//////////////////////////////////////////////////////////////
// Class factory
//
class CFactory : public IClassFactory
{
public:
    // IUnknown
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
    virtual ULONG   __stdcall AddRef();
    virtual ULONG   __stdcall Release();
    // Interface IClassFactory
    virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
                                             const IID& iid,
                                             void** ppv);
    virtual HRESULT __stdcall LockServer(BOOL bLock);

    // Constructor
    CFactory() : m_cRef(1) {}

    // Destructor
    ~CFactory() {}
private:
    long m_cRef;
};
//
// Class factory IUnknown implementation
//
HRESULT __stdcall CFactory::QueryInterface(REFIID iid, void** ppv)
{
    IUnknown* pI;
    if ((iid == IID_IUnknown) || (iid == IID_IClassFactory))
    {
        pI = static_cast<IClassFactory*>(this);
    }
    else
    {
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    pI->AddRef();
    *ppv = pI;
    return S_OK;
}

ULONG CFactory::AddRef()
{
    return InterlockedIncrement(&m_cRef);
}
```

```
ULONG CFactory::Release()
{
    if (InterlockedDecrement(&m_cRef) == 0)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}

//
// IClassFactory implementation
//
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
                                           const IID& iid,
                                           void** ppv)
{
    HRESULT hr = E_FALL;
    // Cannot aggregate
    if (pUnknownOuter != NULL)
    {
        return CLASS_E_NOAGGREGATION;
    }

    // Create component.
    CA* pA = new CA;
    if (pA == NULL)
    {
        return E_OUTOFMEMORY;
    }

    // Initialize the component.
    hr = pA->Init();
    if (FAILED(hr))
    {
        // Initialization failed. Delete component.
        pA->Release();
        return hr;
    }

    // Get the requested interface.
    hr = pA->QueryInterface(iid, ppv);
    pA->Release();
    return hr;
}

// LockServer
HRESULT __stdcall CFactory::LockServer(BOOL bLock)
{
    if (bLock)
```

```
        {
            InterlockedIncrement(&g_cServerLocks);
        }
        else
        {
            InterlockedDecrement(&g_cServerLocks);
        }
        return S_OK;
    }

    ////////////////////////////////////////////////////////////////
    //
    // Exported functions
    //
    STDAPI DllCanUnloadNow()
    {
        if ((g_cComponents == 0) && (g_cServerLocks == 0))
        {
            return S_OK;
        }
        else
        {
            return S_FALSE;
        }
    }


    //
    // Get class factory.
    //
    STDAPI DllGetClassObject(const CLSID& clsid,
                             const IID& iid,
                             void** ppv)
    {
        // Can we create this component?
        if (clsid != CLSID_Component1)
        {
            return CLASS_E_CLASSNOTAVAILABLE;
        }

        // Create class factory.
        CFactory* pFactory = new CFactory; // No Addref in constructor
        if (pFactory == NULL)
```

```
    {
        return E_OUTOFMEMORY ;
    }

    // Get requested interface.
    HRESULT hr = pFactory->QueryInterface(iid, ppv) ;
    pFactory->Release() ;

    return hr ;
}

//
// Server registration
//
STDAPI DllRegisterServer()
{
    return RegisterServer(g_hModule,
                          CLSID_Component1,
                          g_szFriendlyName,
                          g_szVerIndProgID,
                          g_szProgID) ;
}

STDAPI DllUnregisterServer()
{
    return UnregisterServer(CLSID_Component1,
                            g_szVerIndProgID,
                            g_szProgID) ;
}

////////////////////////////////////////////////////////////
//
// DLL module information
//
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD dwReason,
                      void* lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        g_hModule = hModule ;
    }
    return TRUE ;
}
```

程序清单 8-4 内部组件(被聚合组件)的实现。

CMPNT2.CPP

```
//
// Cmpnt2.cpp - Component 2
//   Note the changes in the class factory marked with
//
#include <iostream.h>
#include <objbase.h>

#include "Iface.h"
#include "Registry.h"

void trace(const char * msg) { cout << "Component 2: \t" << msg << endl ;}

////////////////////////////////////////////////////////
//
// Global variables
//

// Static variables
static HMODULE g_hModule = NULL ;      // DLL module handle
static long g_cComponents = 0 ;        // Count of active components
static long g_cServerLocks = 0 ;       // Count of locks

// Friendly name of component
const char g_szFriendlyName[]
      = "Inside COM, Chapter 8 Example 2, Component 2" ;

// Version-independent ProgID
const char g_szVerIndProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt2" ;

// ProgID
const char g_szProgID[] = "InsideCOM.Chap08.Ex2.Cmpnt2.1" ;

////////////////////////////////////////////////////////
//
// Nondelegating IUnknown interface
//
struct INondelegatingUnknown
{
      virtual HRESULT __stdcall
            NondelegatingQueryInterface(const IID&, void**) = 0;
      virtual ULONG __stdcall NondelegatingAddRef() = 0;
      virtual ULONG __stdcall NondelegatingRelease() = 0;
};

////////////////////////////////////////////////////////
//
// Component
//
class CB : public IV,
```

```
        public INondelegatingUnknown
{
public:
    // Delegating IUnknown
    virtual HRESULT __stdcall
        QueryInterface(const IID& iid, void** ppv)
    {
        trace("Delegate QueryInterface.");
        return m_pUnknownOuter->QueryInterface(iid, ppv);
    }
    virtual ULONG __stdcall AddRef()
    {
        trace("Delegate AddRef.");
        return m_pUnknownOuter->AddRef();
    }
    virtual ULONG __stdcall Release()
    {
        trace("Delegate Release.");
        return m_pUnknownOuter->Release();
    }
    // Nondelegating IUnknown
    virtual HRESULT __stdcall
        NondelegatingQueryInterface(const IID& iid, void** ppv);
    virtual ULONG   __stdcall NondelegatingAddRef();
    virtual ULONG   __stdcall NondelegatingRelease();
    // Interface IY
    virtual void __stdcall Fy() { cout << "Fy" << endl;}
    // Constructor
    CB(IUnknown* m_pUnknownOuter);
    // Destructor
    ~CB();
private:
    long m_cRef;

    IUnknown* m_pUnknownOuter;
};
//
// IUnknown implementation
//
HRESULT __stdcall CB::NondelegatingQueryInterface(const IID& iid,void** ppv)
{
    if (iid == IID_IUnknown)
    {
```

```
        // !!! CAST IS VERY IMPORTANT !!!
        *ppv = static_cast<INondelegatingUnknown*>(this);
    }
    else if (iid == IID_IY)
    {
        *ppv = static_cast<IY*>(this);
    }
    else
    {
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}

ULONG __stdcall CB::NondelegatingAddRef()
{
    return InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CB::NondelegatingRelease()
{
    if (InterlockedDecrement(&m_cRef) == 0)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}

//
// Constructor
//
CB::CB(IUnknown* pUnknownOuter)
: m_cRef(1)
{
    ::InterlockedIncrement(&g_cComponents);
    if (pUnknownOuter == NULL)
    {
        trace("Not aggregating; delegate to nondelegating IUnknown.");
        m_pUnknownOuter = reinterpret_cast<IUnknown*>
                          (static_cast<INondelegatingUnknown*>
                          (this));
    }
    else
    {
        trace("Aggregating; delegate to outer IUnknown.");
```

```
{

        m_pUnknownOuter = pUnknownOuter;
    }
//
// Destructor
//
CB::~CB()
{
    InterlockedDecrement(&g_cComponents);
    trace("Destroy self.");
}

////////////////////////////////////////////////////////
//
// Class factory
//
class CFactory : public IClassFactory
{
public:
    // IUnknown
    virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
    virtual ULONG   __stdcall AddRef();
    virtual ULONG   __stdcall Release();

    // Interface IClassFactory
    virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,
                                             const IID& iid,
                                             void** ppv);
    virtual HRESULT __stdcall LockServer(BOOL bLock);

    // Constructor
    CFactory() : m_cRef(1) {}

    // Destructor
    ~CFactory() {}

private:
    long m_cRef;
};

//
// Class factory IUnknown implementation
//
HRESULT CFactory::QueryInterface(const IID& iid, void** ppv)
{
    if ((iid == IID_IUnknown) || (iid == IID_IClassFactory))
    {
        *ppv = static_cast<IClassFactory*>(this);
    }
```

```
    else
    {
        *ppv = NULL;
        return E_NOINTERFACE;
    }
    reinterpret_cast<IUnknown*>(*ppv)->AddRef();
    return S_OK;
}

ULONG __stdcall CFactory::AddRef()
{
    return InterlockedIncrement(&m_cRef);
}

ULONG __stdcall CFactory::Release()
{
    if (InterlockedDecrement(&m_cRef) == 0)
    {
        delete this;
        return 0;
    }
    return m_cRef;
}

//
// IClassFactory implementation
//
HRESULT __stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
                                           const IID& iid,
                                           void** ppv)
{
    // Aggregate only if the requested iid is IID_IUnknown.
    if ((pUnknownOuter != NULL) && (iid != IID_IUnknown))
    {
        return CLASS_E_NOAGGREGATION;
    }

    // Create component.
    CB* pB = new CB(pUnknownOuter);
    if (pB == NULL)
    {
        return E_OUTOFMEMORY;
    }

    // Get the requested interface.
    HRESULT hr = pB->NondelegatingQueryInterface(iid, ppv);
    pB->NondelegatingRelease();
    return hr;
}
```

```
// LockServer
HRESULT CFactory::LockServer(BOOL bLock)
{
      if (bLock)
      {
            InterlockedIncrement(&g_cServerLocks) ;
      }
      else
      {
            InterlockedDecrement(&g_cServerLocks) ;
      }
      return S_OK ;
}

////////////////////////////////////////////////////////
//
// Exported functions
//
STDAPI DllCanUnloadNow()
{
      if ((g_cComponents == 0) && (g_cServerLocks == 0))
      {
            return S_OK ;
      }
      else
      {
            return S_FALSE ;
      }
}

//
// Get class factory.
//
STDAPI DllGetClassObject (const CLSID& clsid,
                          const IID& iid,
                          void** ppv)
{
      // Can we create this component?
      if (clsid != CLSID_Component2)
      {
            return CLASS_E_CLASSNOTAVAILABLE ;
      }

      // Create class factory.
      CFactory* pFactory = new CFactory ; // No Addref in constructor
      if (pFactory == NULL)
```

```
    {
        return E_OUTOFMEMORY ;
    }

    // Get requested interface.
    HRESULT hr = pFactory->QueryInterface(iid, ppv) ;
    pFactory->Release() ;

    return hr ;
}

//
// Server registration
//
STDAPI DllRegisterServer()
{
    return RegisterServer(g_hModule,
                          CLSID_Component2,
                          g_szFriendlyName,
                          g_szVerIndProgID,
                          g_szProgID) ;
}

STDAPI DllUnregisterServer()
{
    return UnregisterServer(CLSID_Component2,
                            g_szVerIndProgID,
                            g_szProgID) ;
}
////////////////////////////////////////////////////////////
//
// DLL module information
//
BOOL APIENTRY DllMain(HANDLE hModule,
                      DWORD dwReason,
                      void* lpReserved)
{
    if (dwReason == DLL_PROCESS_ATTACH)
    {
        g_hModule = hModule ;
    }
    return TRUE ;
}
```

8.4.1 盲目聚合

在前面的例子中,外部组件只是聚合了内部组件实现的一个接口。客户能够访问的内部组件接口将只有 IY。若内部组件实现了接口 IA,客户也无法得到指向此接口的指针,此时外部组件将返回 E_NOINTERFACE。

那么当外部组件需要聚合内部组件实现的多个接口时怎么办呢? 为此,按照下面的方法对外部组件进行修改以使之支持内部组件实现的另外一个接口:

```
else if ((iid == IID_IY) || (iid == IID_IZ))
{
    return m_pInner->QueryInterface(iid, ppv);
}
```

当然这样做需要能够得到外部组件的源代码并需要一个编译器来编译修改后的代码。这样一来,当我们希望客户能够访问内部组件支持的所有接口时,包括那些在编写好、编译好或发行了外部组件之后新加上的接口时怎么办呢? 当然解决这个问题的方法也是很简单的,那就是将条件判断去掉,即不对接口标识符进行判断,而只是简单地将请求的接口 IID 直接传递给内部组件:

```
⋮
else if (iid == IID_IX)
{
    *ppv = static_cast<IX*>(this);
}
else // No conditional
{
    return m_pInner->QueryInterface(iid, ppv);
}
⋮
```

这种类型的聚合被称作是盲目聚合,因外部组件只是盲目地将接口 ID 传给内部组件。在使用盲目聚合时,外部组件实际上将放弃它暴露给客户的内部组件接口的控制权。大多数情况下不应使用盲目聚合。原因之一是内部组件支持的接口可能同外部组件所支持的接口是不兼容的。例如,外部组件可能会支持一个保存文件的接口 ISlowFile,而内部组件则支持另外一个保存文件的接口 IFastFile。假定客户总是先于 ISlowFile 请求 IFastFile。若外部组件盲目地聚合了内部组件,那么客户将得到内部组件 IFastFile 的接口指针。由于内部组件并不知道关于外部组件的任何信息,因此它将不能正确地保存同外部组件有关的任何信息。避免出现这些问题的方法是避免使用盲目聚合。

当然完全禁止使用也是不太现实的。但为避免冲突,可以使用如下两种方法。第一种方法是在实现外部组件时,不要在其中包含可能同内部组件的接口在功能上可能会出

现冲突的接口。第二种方法是将外部组件和客户或外部组件和内部组件作为匹配对来实现。

一、元接口

可能会引起外部组件和内部组件出现冲突的那些接口通常是那些在功能上有重叠的接口。若能够保证外部组件接口同内部组件的接口在功能上不会出现重叠,则接口出现冲突的可能性也将大大降低。当然要做到这一点是比较困难的,因内部组件可能会被升级以支持那些在实现外部组件时并没有考虑到的接口。那些最不可能同组件中已有接口出现冲突的接口被称作是元接口,或类接口。元接口将用于操作组件本身而非组件实现的抽象。

例如,假定有一个位图变换程序。用户可以使用若干变换算法对一个位图进行修改。这些变换算法都是以内部组件的形式实现的,用户可以在系统运行时加入新的变换算法。每一个内部组件都可以接收一副位图,在根据其变换算法对之进行变换之后输出变换后的位图。外部组件有一个名为 ISetColor 的接口,以便让用户设置内部组件进行处理时所用的颜色。另外外部组件还有一个 IToolInfo 接口,其作用是在工具条上显示与每一个变换算法相应的按钮并在用户选择某个图标时创建一个被包容的组件。如图 8-7 所示。

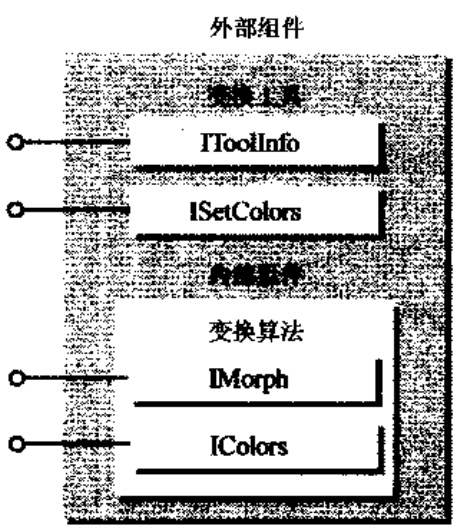


图 8-7 接口 IToolInfo 是一个元接口,它不太可能会同内部组件的接口出现冲突。而接口 ISetColors 则不是一个元接口,它将只在内部组件的作用域内起作用,并与内部组件的 IColors 方法是冲突的

ISetColors 就是一个扩展变换算法抽象的常规接口例子。变换组件可能会有一个名为 IColor 的接口,以操作其颜色集。第二个接口 IToolInfo 则是元接口的一个例子。它的所有操作只是给应用程序提供了一种将变换算法当成是工具条按钮的方法,而与位图的变换没有什么关系。此接口没有扩展变换组件的抽象,只是给其客户提供了一些关于此组件

本身的一些信息。

当在一个组件系统中的一个由不同的组件构成的集合上实现元接口时，它将是最有用的。此时元组件将给系统提供一种操作其组件的一致的方法，从而提高代码的可复用性。绝大多数情况下，在给已有组件加上元接口是由客户的开发人员完成的。通过元接口，客户可以获取具有不同来源的组件，并可按相同的方法对之进行处理。

二、匹配对

另外一种避免接口冲突的方法是给内部组件或客户提供关于外部组件的知识。若客户同外部组件是一块建立的，那么它将知道外部组件所实现的那些接口，并可使用这些接口，而不使用内部组件可能会实现的接口。若内部组件和外部组件是一块建立的，那么使它们之间没有冲突的接口将是一件非常容易的事情。当然，这种匹配对的方法要求开发人员除了能够控制外部组件之外，还要能够控制内部组件或客户。

8.5 现实世界中的聚合和包容

前面我们介绍了如何使用包容和聚合实现组件的复用。但在复用组件时，这些却并不是那么美好的。由于外部组件只是内部组件的一个客户，如果它想对内部组件的某一个成员函数进行定制，那么只能在调用此函数之前或之后调用其他函数，而没法在此函数的中间加上一些新的功能。另外由于某个特定的成员函数可以设置或修改组件的那些客户无法访问的内部状态，因此也无法替换接口中某个特定函数的整个实现。

例如若使用接口来打开一个文件，那么必须调用同一接口来关闭此文件，因为客户并不知道关于此接口或其任一函数的实现细节。同样地，外部组件也并没有超越这些常规客户的特殊访问权限或能力，它也必须象其他任何客户那样一致地、正确地使用内部组件。

为克服这些缺点，组件的设计必须能够为组件的复用提供方便。这一点对于 C++ 类也是一样的：如果没有一种合适的设计，想对其进行扩展或特殊化将是极为困难的，甚至是不可能的。这满足这种定制的要求，在 C++ 类中提供了保护型的成员函数，以便派生类能够利用这些函数获取关于基类的内部状态信息。与此类似，COM 组件也能够利用接口以使之能够更简便地被包容或聚合。

8.5.1 组件的内部状态信息

COM 中的所有功能都是通过接口实现的。因此，为提供内部状态信息，内部组件可以加上一个新的接口。此接口可以给外部组件提供一些状态信息，以帮助开发人员实现对组件的定制。如图 8-8 所示。关于这些内部状态接口，并没有什么特定奇怪的地方：它们只不过为简化组件的定制或使此种定制成为可能而提供了对内部组件状态的一些受控

制的访问。常规的客户也可以在需要的时候使用这些接口,但大多数情况下,这些接口可能是没有什么用处的。

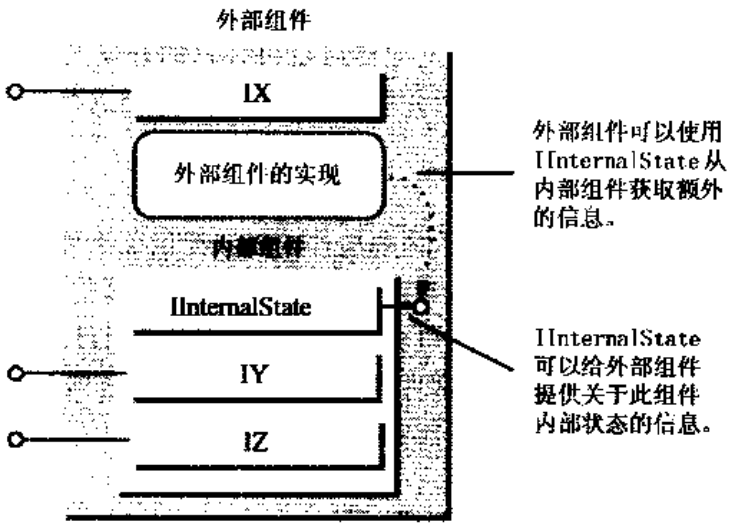


图 8-8 内部组件可以通过一个能够让外部组件查询其内部状态信息的接口以简化定制

能够给外部组件提供此类信息的接口在 COM 中是非常重要的。一般而言,COM 组件都是由许多小接口构成的。从内部实现上讲,这些接口都是互相依赖的,例如它们将共享相同的成员变量及其他内部状态条件。这意味着并不能从组件中选择某个接口并单独使用它,因为此接口可能依赖于其他某个接口的状态或信息来控制组件内部状态的某些方面。参见下面的图 8-9。此处我们再一次看到,外部组件只是内部组件的一个客户,它并没有什么特殊的能力。

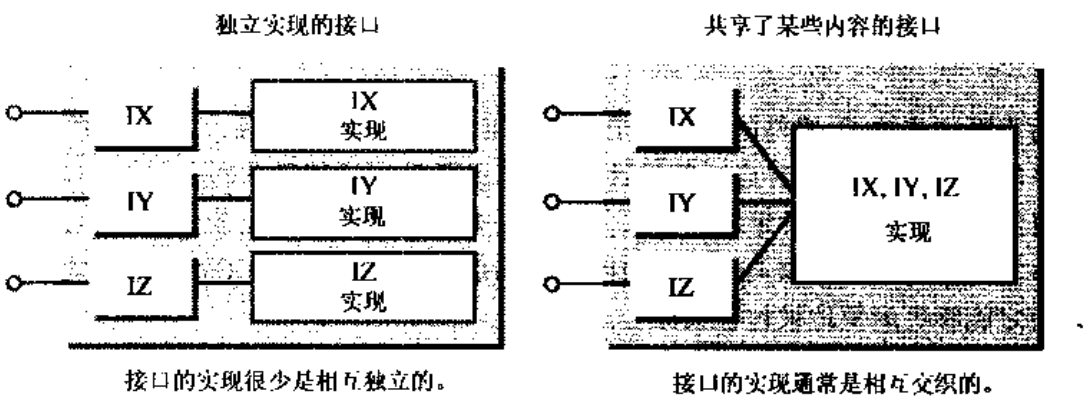


图 8-9 客户通常是将接口当作是独立的实体对待。但是一般而言,接口一般都要共享一些实现细节。这使得对于组件的扩展或特殊化将更为困难。在组件中加上新的接口以提供关于组件的内部视图可以使对组件的定制得以简化

8.5.2 虚拟函数的模拟

增加新的接口不但相当于是给 COM 组件提供了一个与 C++ 中受保护的成员函数相等价的东西，而且使得可以用接口来替代虚拟函数。许多情况下，虚拟函数是用作回调函数的。基类可以在完成某个操作之前、之间或之后调用某个虚拟函数以给派生类提供一个定制此操作的机会。同样地，COM 组件也可以通过定义一个定制接口以实现相同的功能。组件并不需要实现此接口，而只需调用定制接口。那些希望定制组件的客户可以实现此定制接口并将此指针传给此组件。客户可以使用这一技术而不使用组件的包容或聚合(参阅图 8-10)。

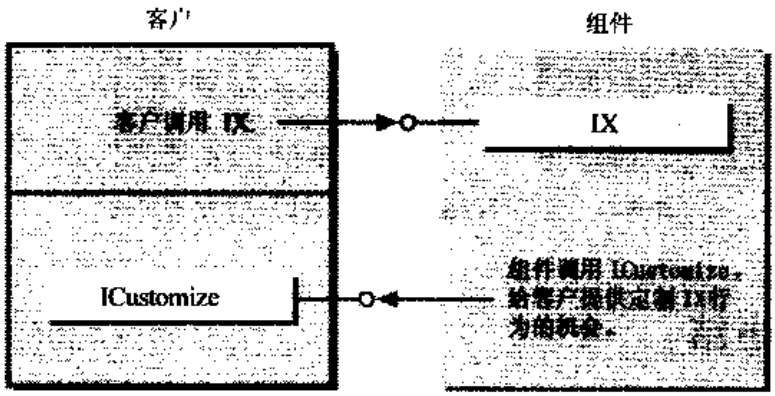


图 8-10 组件可以定义一个向外的接口并调用之以实现对它的定制

继承同使用回调函数或像类似于 ICustomize 的外部接口的真正区别在于：在后一种情况下，必须将客户连接到组件上。关于这一点我们将在第 13 章中详细讨论。但在使用此技术时，必须要注意循环引用计数的问题。一般而言，需要将 ICustomize 作为其所在组件的一部分实现，并对之单独进行引用计数。

设计良好的接口通常会给其潜在的客户提供关于其内部状态的信息及定制接口。真正良好的接口还应该提供此定制接口的缺省实现，以使客户在不需要定制此组件的情况下无需去实现这些接口。

从概念上讲，实现继承所能够提供的都可以通过接口、包容和聚合来达到。但显然 C++ 中的实现继承比图 8-11 所示的 COM 解决方案要更方便一些。但是为使用 C++ 的实现继承，需要能够得到基类的源代码，这一点意味着当基类发生变化时，整个程序需要重新编译。在组件架构下，组件经常在发生变化，而源代码一般也是不可获得的，因此重新编译在许多情况下是不可能的。正是通过组件复用，COM 可以显著地降低组件和它所定制的组件之间的依赖关系。

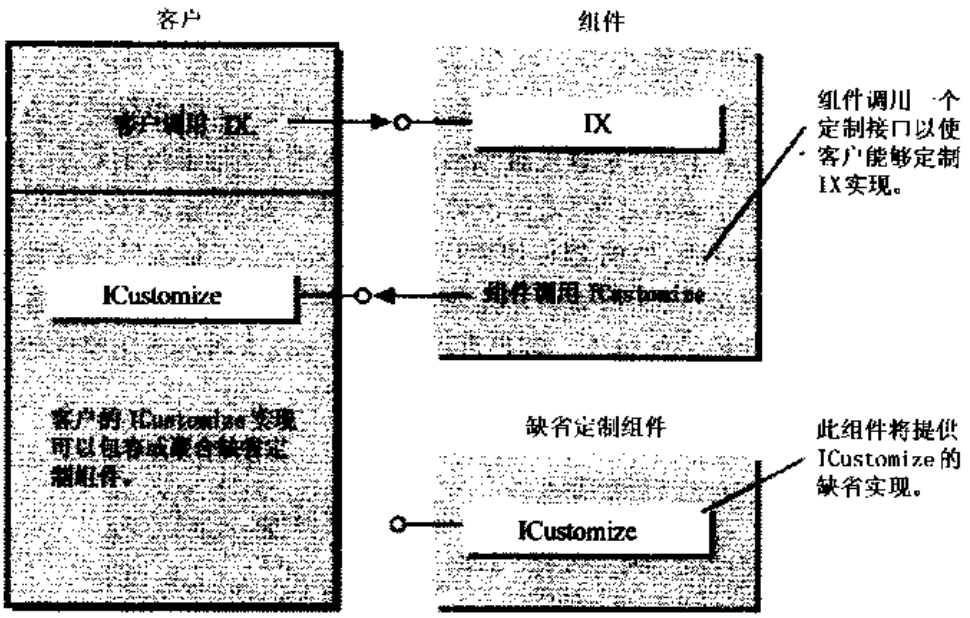


图 8-11 组件可以提供定制组件的一个缺省实现

8.6 本章小结

在这一章中我们介绍了如何通过组件的包容和聚合实现组件的复用、扩展和定制。组件的复用同将一个组件包容在另外一个组件中是一样简单的。当需要对组件进行定制时,可以在调用其成员函数的前后加上自己的代码。

如果只是希望给组件增加新的接口,那么可以使用聚合的方法。在作者看来,聚合实际上是包容的一个特例。当外部组件聚合内部组件的某个接口时,外部组件可以将接口指针直接传给客户,而不需要重新实现此接口或转发相应的函数调用。

但并不是所有组件都是可以被聚合的。为此需要对组件的实现稍稍加以修改。此时内部组件需要两个不同的未知接口。其中一个未知接口要实现 IUnknown 接口的成员函数,而另外一个未知接口则可以在组件未被聚合的情况下将有关的函数调用转发给非代理未知接口;而在其被聚合的情况下则将其转发给外部组件的 IUnknown 接口。

包容和聚合为实现组件的复用和定制提供了一种极具鲁棒性的机制。这些技术使得在 COM 架构下不再需要实现继承,因为在组件架构下,客户必须同组件的实现完全隔离开。否则当客户所用的组件发生变化时,客户必须被重新编写、重新编译或重新链接。

在本章所讨论的组件复用的基础上,下一章我们将讨论另外一种类型的复用:此时被复用的不再是组件,而是 C++ 源代码。在那里我们将实现 IUnknown 和 IClassFactory 的基类,以便派生出我们所需的组件。

第 9 章　编程工作的简化

不管人们做什么事情，他们总会尽量想办法让它简化——当然在进行体育锻炼时，这样做也许是与锻炼的宗旨相违背的。时下电视上充斥着大量非常成功的击球运动器械及其他类似设备的广告。这些运动器械使已有的最简单的运动思想实现了“机械化”，并且使之执行起来更为容易。（对于这些机器，最好的评论是它可以帮助运动者将注意力放在击球上而不是脖子或其他什么地方。天才知道它有没有这种作用！）

既然每个人都想使事情得以简化，这一章我们就来看一下如何使 COM 组件的实现和使用更为容易。当然不论怎样，编写 COM 组件总比击球要难一些，如果按困难的方式来做这件事，那么肯定会吃力不讨好。

这一章内容的安排是：首先我们将使用 C++ 类以使程序员可以像使用 C++ 类那样使用 COM 组件。其次我们将建立一些 C++ 类以完成组件的实现中那些令人生厌的工作。

> **编译器对 COM 的支持**
>
> Microsoft Visual C++ 5.0 为简化 COM 组件的开发和使用而对 C++ 语言进行了一些扩展。这些扩展目前正处在开发过程中，有关的详细细节可参阅 Visual C++ 5.0 的联机帮助。

9.1　客户端的简化

前面我们已经看到，COM 组件的使用不像典型的 C++ 类的使用那样简单。首先一点是需要进行引用计数。若忘记了对某指针调用 AddRef，为了找出程序中的错误，可能需要将整个周末的时间赔进去。并且可能会访问指向某个已被释放的组件上接口的指针，从而导致应用程序的崩溃。要找出这些忘记了的 AddRef 及 Release 调用是相当难的。更糟的是，在每次运行程序时，组件可能不是在同一地方被释放的。

即使我们在该调用 Release 的时候调用了它，但在程序中却不一定会真正地调用它，因为 C++ 的异常处理程序不知道关于 COM 组件的任何信息。因此在发生了异常之后，异常处理程序不会调用 Release 以释放此 COM 组件。

正确地处理 AddRef 和 Release 只是万里长征的第一步，下面还需要做许多工作以使调用 QueryInterface 的代码合理化。读者一定会注意到，一个 QueryInterface 调用将占用几

行代码。一对 QueryInterface 调用极易使函数中实际有用的代码黯然失色。为避免此种问题,作者的做法通常是缓存接口指针,而不是在每次需要它的时候都去查询这些接口。这种方法可以提高程序的效率及代码的可读性,付出的代价则是增大了内存开销。但 QueryInterface 还有另外一个更为严重的问题,即它不是类型安全的。如果将某个不正确的指针传给了 QueryInterface 函数,编译器不会帮助用户找出这一潜在的错误。例如虽然下面的代码将一个 IY 接口指针赋给了一个 IZ 接口指针,但编译时仍然会通过:

```
IZ * pIA ;
PIX->Queryinterface(IID_IY, (void* *) &Piz) ;
```

之所以会出现上述这种情况,关键在于那个可恶的 void 指针向编译器隐藏了类型信息。

对于上述这些问题,可以通过封装加以解决。为进行封装,有两种不同的方法,其一是使用智能指针类来封装接口指针;另外一种方法则是通过包装类对接口本身进行封装。下面我们分别介绍这两种方法,首先介绍的是智能指针。

9.1.1 智能接口指针

简化客户的第一种方法是通过智能接口指针而不是常规的接口指针来访问组件。智能接口指针的使用同常规的 C++ 指针的使用是相同的。它将把引用计数这类细节隐藏起来,并且当程序的执行离开了智能接口指针的作用域之后,相应的接口将被释放掉。这将使得对 COM 组件的使用同 C++ 对象的使用是类似的。

一、什么是智能指针

一个智能指针实际上就是一个重载了操作符 -> 的类。智能接口指针类包含指向另外一个对象的指针。当用户调用智能指针上的 -> 操作符时,智能指针把此调用转发给它所包含的指针所指的对象。智能接口指针中所包含的指针将是指向一个接口的。

下面我们来看一个简单的例子。CFooPointer 可以说得上是一个最简单的智能指针类,其中包含一个指针,并重载了 -> 操作符。

程序清单 9-1 智能接口指针类 IPtr 的实现。

```
3class CFoo
{
public:
    virtual void Bar() ;
};

class CFooPointer
{
```

```
public:
    CFooPointer(CFoo* p) { m_p = p; }
    CFoo* operator->() { return m_p; }
private:
    CFoo* m_p;
};
:
void Funky(CFoo* pFoo)
{
// Create and initialize the smart pointer.
CFooPointer spFoo(pFoo);

// The following is the same as pFoo->Bar();
spFoo->Bar();
}
```

在下面的例子中，Funky 函数创建了一个名为 spFoo 的 CFooPointer 对象并用 pFoo 对之进行初始化。spFoo 指针将把相应的函数调用转发给 m_p，也就是 pFoo。这样通过 spFoo 将可以调用 CFoo 所实现的任何函数。这样做最妙的地方可能在于无需明确地在 CFooPointer 中列出 CFoo 的所有成员函数。对于 CFoo 来说，函数 operator-> 表示的是“反引用我”；而对 CFooPointer 来说，函数 operator-> 表示的则是“不要反引用我，反引用 m_p 好了”。关于这一点请参阅图 9-1。

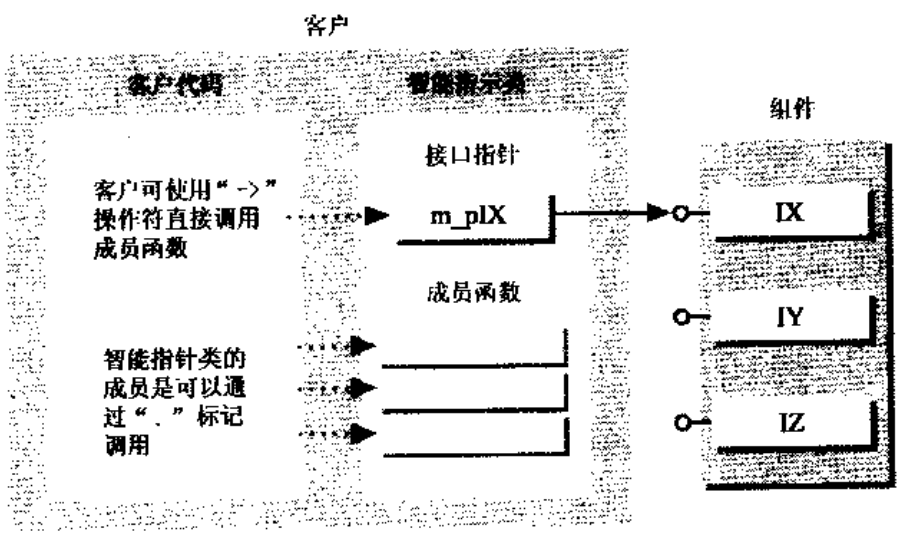


图 9-1 智能接口指针可以将函数调用转发给相应类中所包含的接口指针

CFooPointer 作为一个智能指针是不够的，它实际上并没有做任何事情。一个具有良好优化能力的编译器可能会将 CFooPointer 优化得消声匿迹。同时它的行为也不像一个指针，例如将 pFoo 值赋给 spFoo 是不可能的。这实际上将没法工作，因操作符 = 并没有被重

载以给 m_p 赋值。为使客户相信一个 CFooPointer 同一个指向 CFoo 的指针是相同的，CFooPointer 需要重载一系列其他的操作符，如 * 和& 等。因为这些操作符需要处理 m_p 指针而不是 CFooPointer 对象本身。

二、接口指针类的实现

用于处理 COM 接口的智能接口指针类没有字符串类那样丰富。在 AcitveX 模板库(ATL)中有两个名为 CComPrt 及 CComQIPtr 的 COM 接口指针类。在 Microsoft 基本类库(MFC)中现在也有了一个名为 CIP 的类，只不过它是内部使用的。(参见文件 AFXCOM_.H。)CIP 是一个全功能的智能接口指针，它几乎可以做任何事情。下面给出的将是作者的智能接口指针类实现，这主要是考虑到这些代码阅读起来容易一些。这里给出的类同 ATL 和 MFC 中给出的类是相似的，只不过功能没有那么全而已。

我们给出的接口指针类的名称为 IPtr，程序清单 9-1 给出了其实现。读者不要被这段代码的长度所吓倒，实际上它是相当简单的，只不过为了阅读起来更为方便而在其中加上了一些空白。

IPtr Template in PTR.H

```
//
// IPtr - Smart Interface Pointer
//   Use:  IPtr<IX, &IID_IX> spIX;
//   Do not use with IUnknown; IPtr<IUnknown, &IID_IUnknown>
//     will not compile.  Instead, use IUnknownPtr.
//
template <class T, const IID* piid> class IPtr
{
public:
  // Constructors
  IPtr()
  {
    m_pI = NULL;
  }
  IPtr(T* lp)
  {
    m_pI = lp;
    if (m_pI != NULL)
    {
      m_pI->AddRef();
    }
  }
  IPtr(IUnknown* pI)
```

```
{
    m_pI = NULL;
    if (pI != NULL)
    {
        pI->QueryInterface(*piid, (void **)&m_pI);
    }
}

// Destructor
~IPtr()
{
    Release();
}

// Reset
void Release()
{
    if (m_pI != NULL)
    {
        T* pOld = m_pI;
        m_pI = NULL;
        pOld->Release();
    }
}

// Conversion
operator T*() { return m_pI; }

// Pointer operations
T& operator*()  { assert(m_pI != NULL); return *m_pI; }
T** operator&() { assert(m_pI == NULL); return &m_pI; }
T* operator->() { assert(m_pI != NULL); return m_pI; }

// Assignment from the same interface
{
    if (m_pI != pI)
    {
        IUnknown* pOld = m_pI;    // Save current value.
        m_pI = pI;                // Assign new value.
        if (m_pI != NULL)
        {
            m_pI->AddRef();
        }
        if (pOld != NULL)
        {
            pOld->Release();      // Release the old interface.
        }
```

```
    }
  T* operator=(T* pI)
    return m_pI;
  }

  // Assignment from another interface
  T* operator=(IUnknown* pI)
  {
    IUnknown* pOld = m_pI;        // Save current value.
    m_pI == NULL;

    // Query for correct interface.
    if (pI != NULL)
    {
      HRESULT hr = pI->QueryInterface(*piid, (void**)&m_pI);
      assert(SUCCEEDED(hr) && (m_pI != NULL));
    }
    if (pOld != NULL)
    {
      pOld->Release();          // Release old pointer.
    }
    return m_pI;
  }

  // Boolean functions
  BOOL operator! () { return (m_pI == NULL) ? TRUE : FALSE; }

  // Requires a compiler that supports BOOL
  operator BOOL() const
  {
    return (m_pI != NULL) ? TRUE : FALSE;
  }

  // GUID
  const IID& iid() { return *piid; }
private:
  // Pointer variable
  T* m_pI;
};
```

三、接口指针的使用

IPtr 实现的使用是相当简单的，特别是对于模板类。首先可以通过一个接口类型及指向其 IID 的指针来创建一个接口指针对象。（更好的方法是使用一个引用来作为模板参数，但遗憾的是大多数编译器不支持这一点。）然后就可以调用 CoCreateInstance 来创建

所需的组件并获取其指针了。从下面的代码中我们可以看到 IPtr 是如何有效地模拟真正的指针的。例如我们可以对某个 IPtr 对象使用 & 操作符，如同它是一个真正的指针那样。

```
void main()
{
   IPtr < IX, &IID_IX > spIX ;
   HRESULT hr = ::CoCreateInstance(CLSID_COMPONENT1,
                                   NULL,
                                   CLSCTX_ALL,
                                   spIX.iid()
                                   (void** )&spIX) ;
   if (SUCCEEDED(hr))
   {
      spIX-> Fx() ;
   }
}
```

上面对于 CoCreateInstance 的调用并不是类型安全的，但可以通过在 IPtr 模板中定义另外一个函数以使之成为类型安全的。

```
HRESULT CreateInstance(const CLSID& clsid,IUnknown * pI, DWORD clsctx)
{
      Release() ;
      return CoCreateInstnace(clsid, pI, clsctx, * piid, void** )&m_pI) ;
}
```

对于 IPtr 可按如下方式使用之：

```
IPtr < IX, &IID_IX > spIX ;
HRESULT hr = spIX.CreateInstance(CLSID_Component1,
                                 NULL,
                                 CLSCTX_INPROC_SERVER) ;
```

另外可以看到在智能接口指针变量的前面加上了一个前缀 sp，以表示相应的指针为智能指针。

四、引用计数

在本例中，关于智能指针最妙之处可能在于我们无需记住去调用 Release。当程序的执行脱离开智能指针的作用域范围之外时，它将在其析构函数中自动调用 Release。这种作法还有另外一个好处就是即使在产生了异常的情况下，相应的接口也将能够被释放掉，因智能指针实际上是一个 C++ 对象。

若想将某个智能指针指向的接口释放掉，此时不能通过智能指针调用 Release。因智能指针不知道客户调用的函数是什么，而总是盲目地转发这些调用。如果这样做的话，接

口将被释放掉,但在智能指针的指针变量中却仍然保存着一个非空的接口指针值。此时如果接口再使用此智能指针,将可能引起访问错误。

当希望释放接口时,不同的智能指针具有不同的方法。例如大多数智能指针(包括IPtr)将实现一个 Release 函数。此 Release 函数需要通过点记法来调用,而不能用 -> 操作符:

```
spIX.Release();
```

释放 IPtr 对应接口的另外一种方法是给它指定一个 NULL 值:

```
spIX = NULL;
```

至于此语句为什么能够完成接口的释放,我们将在下面介绍。

五、赋值

IPtr 类重载了 = 操作符,以便可以将一个接口指针赋值给 IPtr 包含的指针:

```
T* operator=(T* pI)
{
  if (m_pI != pI)
  {
    IUnknown* pOld = m_pI;
    m_pI = pI;
    if (m_pI != NULL)
    {
      m_pI->AddRef();
    }
    if (pOld != NULL)
    {
      pOld->Release();
    }
  }
  return m_pI;
}
```

注意 = 操作符实现中的两个令人感兴趣的方面。首先它将相应的调用 AddRef 和 Release,这样在程序中就无需再进行这些调用了。其次,智能接口指针将在给 m_pI 设置了新指针值之后释放当前指针。这可以防止在赋值之前相应的组件被从内存中删除掉。

下面的代码将把 pIX1 指针值赋给 spIX 的 m_pI 成员变量。赋值操作符将在此过程中调用相应接口上的 AddRef。在使用完 spIX 后,pIX2 将被赋给 spIX。此时 = 操作符将释放当前所包含的接口指针(指向 pIX1 的指针),然后保存 pIX2,并调用 pIX2 上的 AddRef。

```
void Fuzzy(IX* pIX1, IX* pIX2)
```

```
{
    IPtr<IX, IID_IX> spIX;
    spIX = pIX;
    spIX->Fx();

    spIX = pIX2;
    spIX->Fx();
}
```

在 IPtr 的实现中提供了一个转换操作符,以便能够将一个 IPtr 对象赋给相同类型的另外一个 IPtr 对象。关于这一点可参见下面的例子:

```
typedef IPtr<IX, IID_IX> SPIX;
SPIX g_spIX;
void Wuzzy(SPIX spIX)
{
  g_spIX = spIX;
}
```

此赋值操作符只是在两个指针的类型相同时才能够正常工作。为提高代码的可读性,我们使用了 typedef 语句。

六、未知接口赋值

我们的目标是要使对 QueryInterface 调用得以简化。这可以通过对赋值操作符的另外一个重载来完成。

```
T* operator=(IUnknown* pIUnknown);
```

若将另外一个不同类型的接口指针赋值给一个智能指针,赋值操作符将自动调用 QueryInterface。例如在下面的代码中我们将一个 IY 接口指针赋给了一个 IX 智能接口指针。但应注意的是,应对赋值后的智能指针值进行检查,以决定此赋值是否成功。某些智能接口指针类在 QueryInterface 不成功时将会引起一个异常。

```
void WasABear(IY* pIY)
{
  IPtr<IX, IID_IX> spIX;
  spIX = pIY;
  if(spIX)
  {
     spIX->Fx();
  }
}
```

就作者而言,并不赞成赋值操作符调用 QueryInterface。因为我们知道关于重载有一

条规则就是要保证重载后的操作符的行为同相应的内置操作符是相同的。这一点对于调用了 QueryInterface 的赋值操作符显然是不能成立的。因为 C++ 赋值不论如何都不会失败,但 QueryInterface 却可能会失败,所以调用了 QueryInterface 的赋值操作符也可能会失败。

不幸的是,工业界却并不遵循这一点。例如 Microsoft Visual Basic 中就实现了一个调用 QueryInterface 的赋值操作符。ATL 和 MFC 中的智能指针也重载了赋值操作符并在其中调用了 QueryInterface。

interface_cast

上述重载赋值运算符的作法实际是极不可取的。Visual Basic 在赋值运算中调用 QueryInterface 的作法并不意味着 C++ 这样做也是正确的或者是合适的。C++ 也许是一个有章可依的糟糕的梦,但 Visual Basic 则是一个噩梦。

作者更喜欢用一个名为 interface_cast 的函数来封装 QueryInterface。此函数是一个模板函数,程序员可以像使用 dynamic_cast 那样使用它。下面是 interface_cast 的实现。

```
template <class I, const GUID* pGUID>
I* interface_cast(IUnknown* pIUnknown)
{
  I* pI = NULL;
  HRESULT hr = pIUnknown->QueryInterface(*pGUID,
                                         (void**)&pI);
  assert(SUCCEEDED(hr));
  return pI;
}
```

为使用此函数可以按以下类似的方法:

```
IY* pIY = interface_cast<IY, &IID_IY>(this);
```

函数 interface_cast 的优点是在使用它时甚至不需要智能指针类。而其不利之处则在于它需要一个能够明确将模板函数实例化的编译器。Visual C++ 5.0 就是这样的一个编译器。

七、IUnknownPointer

在 PTR.H 中还包含另外一个指针类 IUnknownPtr,它实际上是 IPtr 的专用于 IUnknown 的一个特殊版本。IUnknownPtr 类并不是一个模板类,并且没有实现那个调用了 QueryInterface 的赋值运算符。之所有要定义这么一个 IUnknownPtr 类,是因为 IPtr 不能用 IUnknown 来实例化,因为这样做将会导致两个具有相同原型的赋值运算符。所以对于智能 IUnknown 指针,不要使用 IPtr。

```
IPtr<IUnknown & IID_IUnknown> spIUnknown ;   //Error
```

而可以使用 IUnknownPtr:

```
IUnknownPtr spIUnknown ;
```

八、带有智能指针客户的实现

本章的示例代码中包含两个客户:Client1 和 Client2。其中 Client1 的实现方法与前面的章节中所介绍的是相同的。Client2 则是使用智能指针实现的。程序清单 9-2 所列为 Client2 的代码。可以看到,由于在此代码中所有的 QueryInterface 调用都被隐藏起来了,因此程序的可读性也大大提高了。同时 typedef 也将使模板类更为友好。所有处理组件的代码均被放在一个名为 Think 的函数中。

程序清单 9-2 客户 2 (Client 2)使用了 IPtr 智能接口指针类。

CLIENT2.CPP

```
//
// Client2.cpp - Client implementation with smart pointers
//
#include <objbase.h>

#include "Iface.h"
#include "Util.h"     // Traces with labels for our output
#include "Ptr.h"      // Smart pointer classes

static inline void trace(const char * msg)
      { Util::Trace("Client 2", msg, S_OK) ;}
static inline void trace(const char * msg, HRESULT hr)
      { Util::Trace("Client 2", msg, hr) ;}

void Think()
{
      trace("Create Component 1.") ;
      IPtr<IX, &IID_IX> spIX ;
      HRESULT hr = CoCreateInstance(CLSID_Component1,
                                    NULL,
                                    CLSCTX_INPROC_SERVER,
                                    spIX.iid(),
                                    (void**)&spIX) ;
      if (SUCCEEDED(hr))
      {
            trace("Succeeded creating component.") ;
            spIX->Fx() ;
            trace("Get interface IY.") ;
            IPtr<IY, &IID_IY> spIY ;
            spIY = spIX ;          // Use Assignment.
```

```
        if (spIY)
        {
            spIY->Fy();

            trace("Get interface IX from IY.");
            IPtr<IX, &IID_IX> spIX2(spIY); // Use Constructor.
            if (! spIX2)
            {
                trace("Could not get interface IX from IY.");
            }
            else
            {
                spIX2->Fx();
            }
        }

        trace("Get interface IZ.");
        IPtr<IZ, &IID_IZ> spIZ;
        spIZ = spIX;
        if (spIZ)
        {
            spIZ->Fz();

            trace("Get interface IX from IZ.");
            IPtr<IX, &IID_IX> spIX2(spIZ);
            if (! spIX2)
            {
                trace("Could not get interface IX from IZ.");
            }
            else
            {
                spIX2->Fx();
            }
        }
    }
    else
    {
        trace("Could not create component.", hr);
    }
}

int main()
{
    // Initialize COM Library.
    CoInitialize(NULL);
```

```
    // Exercise the smart pointers.
    Think();

    // Uninitialize COM Library.
    CoUninitialize();

    return 0;
}
```

值得注意的是，上面的代码中处理嵌入的 QueryInterface 调用失败的方法。为检测此调用是否成功，我们使用了一个 BOOL 转换操作符，当 IPtr::m_p 非空时它将返回 TRUE。为检测此调用是否失败，使用的则是!操作符。

九、智能指针可能会引起的问题

使用智能指针可能会引起的问题大多数都是无关紧要的。在使用智能指针时，应注意避免调用被包容的接口上的 Release 函数。智能指针中的成员函数都是通过“.”而不是通过“->”来访问的。作为对这些问题的补偿，使用智能指针可以获得令人难以想象的灵活性：我们可以编写一个可以用于所有接口（除 IUnknown 外）的智能指针。

但智能指针的此种特性同时也是其最大的缺点。正是它们太一般化了，它们封装的不是使用的接口而是智能指针的使用。对于许多简单的接口，这一点正是我们希望的。但对于某些接口，更好的方法可能是封装接口本身。

9.1.2 C++ 包装类

若希望封装某个接口或某个接口集，智能指针可能就不太合适了。为对接口进行封装，可以使用 C++ 包装类。一个包装类是一个或多个 COM 接口的客户，它所提供的是对这些接口使用的抽象。应用程序可以调用包装类中那些 COM 接口中成员函数的函数，以使 COM 接口的调用得以简化。这一点同 MFC 包装类简化了 Win32 API 调用是一样的。关于这一点，请参阅图 9-2。

包装类最佳的一些特性在于它们可以利用 C++ 的一些特性，如函数名的重载、操作符的重载以及缺省值等，这些使得 C++ 程序员可以极自然地使用包装类。在 Visual C++ 中提供了可以为 AcitveX 控件及其他许多 COM 组件自动生成包装类的工具。

一、包装类实际上就是组件的包容

同智能指针不同的是，不论包装类是否要改变接口的行为。它都必须重新实现它所包装的接口中的所有成员。另外一个重要的差别是包装类可以在调用接口成员函数的前后加上新的代码。将此种方法同前一章中讨论过的组件复用相比较，将可以发现包装类实际上同组件的包容相同，而智能指针则同组件的聚合类似。

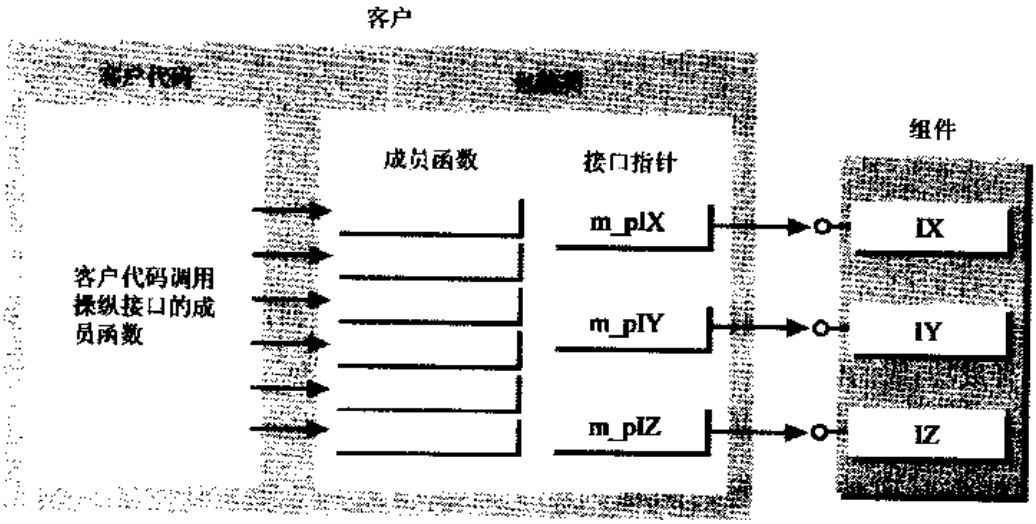


图 9-2 C++ 包装类可以封装组件上的接口以使组件的使用更为容易

二、多个接口的包装

另外一个使用包装类的原因是它可以将几个单独的接口组合成一个逻辑单位。我们知道 COM 组件可能会具有许多小的接口。接口越多,能够一致地使用组件的可能性也将越大,这意味着复用整个应用架构的可能性也将越大。小接口复用虽然说起来很容易,但使用起来却可能非常困难。例如在使用小接口时,我们可能需要用一个接口折叠一个对象,用另外一个接口来拉长一个对象,而用另外一个完全不同的接口来切断一个对象。这意味着客户需要不停地查询接口,从而导致开销的增大。一个具有许多接口的组件就如同是一个臃肿的官僚机构:每个人都只负责他的那一部分工作,为了完成某件事情,可能需要跑好多好多的地方。而将某些接口组合成一个 C++ 包装类则可以使通过这些接口来使用对象的工作得以简化。MFC OLE 支持基本上就是一种功能强大的包装类。

当然不论是使用智能指针、包装类、还是包装类中的智能指针,都可以提高客户代码的可读性,并降低调试的难度。

9.2 服务器端的简化

下面我们将讨论如何才能使 COM 组件的实现得以简化。在前一节中我们介绍了如何简化 COM 组件的使用。在这一节中,我们将讨论怎样简化 COM 组件的实现。为实现这一点,我们将采取两面夹击的策略:首先我们将定义一个实现 IUnknown 的基类 CUnknown。对于从 CUnknown 继承的类,将不再需要考虑其 AddRef 或 Release 的实现问题,并且 QueryInterface 的实现也将会得以简化。

其次我们将给出 IClassFactory 的一个实现 CFactory,它可以同任何从 CUnknown 派生的

COM 组件一块工作。借助于 CLSID 和其他信息，CFactory 将能够创建相应的组件。CFactory 和 CUnknown 都是非常简单的，读者理解起来也应该不会有什么困难。因此在此我们将把重点放在如何用 CFactory 和 CUnknown 来实现一个 COM 组件上，而不再详细讨论 CFactory 和 CUnknown 的具体实现。但首先我们还是要简要介绍一些 CFactory 和 CUnknown。在本书后面的例子中我们将用到 CFactory 和 CUnknown，这也是作者实现这两个类的真正原因。

AcitveX 模板库

使用 CUnknown 和 CFactory 并不是简化 COM 组件实现的唯一方法。另外一种方法是使用 AcitveX 模板库(ATL)。此库给 COM 组件的建立提供了一个可复用的模板类集合。ATL 中的类所支持的功能是此处我们给出的 CFactory 和 CUnknown 的一个超集。它们之间的最大差别在于 ATL 使用了模板。

在明白了 CFactory 和 CUnknown 的工作原理之后，理解 ATL 类的工作原理应该是非常简单的。但 CFactory 和 CUnknown 由于没有使用模板，因此理解起来将会容易一些。关于 ATL 的使用，可以写出一本厚厚的书来，但这一点并不是本书的重点。本书的重点在于介绍如何创建和使用 COM 组件。

9.2.1 未知接口基类

在第 8 章中我们看到被聚合的组件需要实现两个未知接口：一个代理未知接口和一个非代理未知接口。当此组件被聚合时，代理未知接口将把 IUnknown 调用转发给外部组件；而当它未被聚合时，IUnknown 调用则将被转发给非代理未知接口。由于我们希望支持能够被聚合的组件，因此我们希望 CUnknown 实现 INondelegatingUnknown 而不是 IUnknown。程序清单 9-3 列出了 CUnknown 的头文件。

程序清单 9-3 基类 CUnknown 将实现一个非代理未知接口。宏 DECLEAR_IUNKNOWN 将实现一个代理未知接口。

CUNKNOWN.H

```
#ifndef _CUnknown_h_
#define _CUnknown_h_

#include <objbase.h>

//////////////////////////////////////////////////////////////////
//
// Nondelegating IUnknown interface
//   - Nondelegating version of IUnknown
//
interface INondelegatingUnknown
```

```
|
    virtual HRESULT _stdcall
        NondelegatingQueryInterface(const IID& iid, void** ppv) = 0;
    virtual ULONG   _stdcall NondelegatingAddRef() = 0;
    virtual ULONG   _stdcall NondelegatingRelease() = 0;
|;

//////////////////////////////////////////////////////////
//
// Declaration of CUnknown
//   -Base class for implementing IUnknown
//

class CUnknown : public INondelegatingUnknown
|
public:
    // Nondelegating IUnknown implementation
    virtual HRESULT _stdcall NondelegatingQueryInterface(const IID&, void*
* );
    virtual ULONG   _stdcall NondelegatingAddRef();
    virtual ULONG   _stdcall NondelegatingRelease();

    // Constructor
    CUnknown(IUnknown* pUnknownOuter);

    // Destructor
    virtual ~CUnknown();

    // Initialization (especially for aggregates)
    virtual HRESULT Init() { return S_OK;|

    // Notification to derived classes that we are releasing
    virtual void FinalRelease();

    // Count of currently active components
    static long ActiveComponents()
        { return s_cActiveComponents;|

    // Helper function
    HRESULT FinishQI(IUnknown* pI, void** ppv);

protected:
    // Support for delegation
    IUnknown* GetOuterUnknown() const
        { return m_pUnknownOuter;|

private:
    // Reference count for this object
    long m_cRef;

    // Pointer to (external) outer IUnknown
```

```
    IUnknown * m_pUnknownOuter ;
    // Count of all active instances
    static long s_cActiveComponents ;
} ;

//////////////////////////////////////////////////////////////
//
// Delegating IUnknown
//   - Delegates to the nondelegating IUnknown, or to the
//     outer IUnknown if the component is aggregated.
//
#define DECLARE_IUNKNOWN                                    \
    virtual HRESULT __stdcall                               \
        QueryInterface(const IID& iid, void** ppv)          \
    {                                                       \
        return GetOuterUnknown()->QueryInterface(iid,ppv) ; \
    } ;                                                     \
    virtual ULONG __stdcall AddRef()                        \
    {                                                       \
        return GetOuterUnknown()->AddRef() ;                \
    } ;                                                     \
    virtual ULONG __stdcall Release()                       \
    {                                                       \
        return GetOuterUnknown()->Release() ;               \
    } ;
//////////////////////////////////////////////////////////////
#endif
```

CUnknown 实现 INondelegatingUnknown 接口的方法同第 8 章中被聚合组件实现此接口的方法是相同的。当然 CUnknown 不能知道继承它的组件将要实现的接口。后面我们将会看到，组件必须重载 NondelegatingQueryInterface 函数，以在其中加上它们提供的接口的代码。

在此我们不准备对 CUnknown 的实现进行详细的讨论，而只讨论一下其中的几个关键性问题。

一、DECLASE_IUNKNOWN 宏

我们一般不希望在每次实现一个组件时都去实现代理 IUnknown 接口。相反，这一点正是我们定义 CUnknown 的目的所在。在程序清单 9-3 中包含一个名为 DECLARE_IUNKNOWN 的宏，它实现的就是一个代理 IUnknown 接口。虽然作者对于宏可以说是深恶痛绝，但在此处使用宏还是比较合适的。AcitveX 模板库没有使用常规的宏，而是大量地使

用了编译器所认可的宏,也即模板。

实现 INondelegatingUnknown 的另外一个原因

前面我们提到:CUnknown 实现 INondelegatingUnknown 的原因在于为了支持组件的聚合。这也许足够了。但还有另外一个重要的原因在于:派生类必须实现它所继承的任何抽象基类。假定有一个抽象基类 IUnknown,组件 CA 继承了 IUnknown,这样一来它必须实现 IUnknown 中的纯虚成员函数。但此时我们可能希望复用 IUnknown 的一个已有实现 CUnknown。若 CA 同时继承了 CUnknown 和 IUnknown,那么它还是需要实现 IUnknown 中的纯虚函数。

但如果 CA 是通过 CUnknown 和 IX 继承 IUnknown 的,由于此时 IX 并没有实现 IUnknown 的成员函数,因此它们仍然是抽象的,仍然需要 CA 来实现它。这样一来,CA 的实现将使 CUnknown 中的实现无效。为避免这个问题,我们可以用 CUnknown 来实现 INondelegatingUnknown。这种正确的情况如图 9-3 所示。

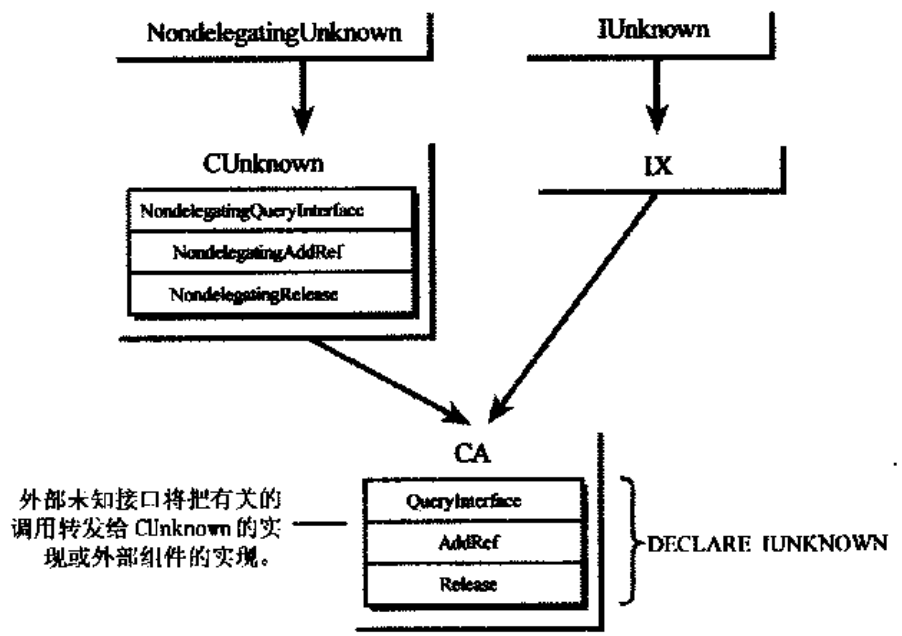


图 9-3 继承 CUnknown 不会实现从 IX 继承得到的 IUnknown 接口。与此相反,CUnknown 实现的是 INondelegatingUnknown。IUnknown 将由组件实现

二、GetOuterUnknown 函数

程序清单 9-3 中的 DECLEARE_ IUNKNOWN 宏的实现使用了函数 CUnknown::GetOuterUnknown。此函数的返回值为一个 IUnknown 指针。对于从 CUnknown 继承的组件,当它没有被聚合时,GetOuterUnknown 返回的是组件的 INondelegatingUnknown 接口指针。当

此组件被聚合时，它返回的则是外部组件的 IUnknown 接口指针。只要组件需要使用其 IUnknown指针，它就可以使用 GetOuterUnknown。

注意，GetOuterUnknown 不会调用它返回的接口指针上的 AddRef，因为在大多数情况下我们不需要对此指针进行引用计数。

三、CUnknown 的构造函数

CUnknown 的构造函数将接收指向外部未知接口的指针作为一个参数，并将其保存起来供 GetOuterUnknown 函数使用。对于从 CUnknown 继承的类的构造函数，都应接收一个指向外部未知接口的指针并将其传给 CUnknown 的构造函数。

四、Init 和 FinalRelease 函数

CUnknown 提供了两个虚拟函数以帮助派生类对内部组件进行管理。派生类可以重载 Init 以创建被包容或聚合的组件。在 CFactory::CreateInstance 创建了组件之后，它将调用相应的 CUnknown::Init。而在 CUnknown::NondelegatinRelease 删除某个组件之前，它将调用 CUnknown::FinalRelease。这可以给被删除的组件提供一个释放它所保持的指向内部组件接口指针的机会。CUnknown::FinalRelease 将增大引用计数值，以避免当相应组件释放被包容组件上的接口时引起 Release 的递归调用。

后面我们将会看到 CUnknown 是如何使组件的实现变得异常简单的。在介绍这方面的情况之前，我们先来看一下可以简化组件的登记与创建的 CFactory。

9.2.2 类厂基类

在实现了一个组件之后，我们需要为此类实现一个类厂。在前面的有关章节中，我们的作法是为每一个类都从开始建立一个单独的类厂。在本章中，类厂实际上已由 C++类 CFactory 实现了。CFactory 不仅实现了 IClassFactory 接口，而且它也提供了可用于实现 DLL 入口点如 DllGetClassObject 的代码。另外 CFactory 还可以便利在同一个 DLL 中包含多个不同的组件成为可能。这些不同的组件将共享同一个 IClassFactory 实现。（在第 7 章中我们已经简要地介绍了这一点。）

CFactory 可用于满足下述条件的任意组件：

- 此组件必须实现一个具有如下原型的创建函数：

```
HRESULT CreateFunction(IUnknown * pUnknownOuter,
                       CUnknown ** ppNewComponent)
```

  这主要是由于 IClassFactory::CreateInstance 的一个一般性实现必须具有创建组件的一般性方法。

- 此组件必须是从 CUnknown 继承得到的。

  IClassFactory::CreateInstance 的 CFactory 实现在创建了组件之后将调用 CUnknown

::Init。组件可以 Init 成员函数以完成一些构造后的初始化处理，如创建另外一个用于包容或聚合的组件。

- 此组件必须填充一个 CFactoryData 结构并将其加入到全局数组 g_FactoryDataArray 中。

当然这些要求同 COM 没有任何关系，只不过是作者实现 CFactory 所用的方法的结果。下面我们来详细讨论一下第一个和最后一个需求。

一、创建函数原型

CFactory 需要一种创建组件的方法。为此它可以使用具有如下原型的任何函数：

```
HRESULT CreateFunction(IUnknown * pUnknownOuter,
                       CUnknown ** ppNewComponent)
```

实例创建函数一般是调用被创建的组件的构造函数，并将相应的指针通过输出参数 ppNewComponent 返回给调用者。若第一个参数 pUnknownOuter 非空，则组件将是被聚合的。

在实现 CreateInstance 的时候，作者的作法是将其作为实现组件的类的一个静态成员函数。这样，此函数将具有与相应组件相同的名字空间。此函数的实现方法可以与前一章中用到的 CreateInstance 函数的实现方法相同。

二、组件数据

CFactory 需要知道它能够创建哪些组件。为此我们使用了一个名为 g_FactoryDataArray 的全局数组，其中包含能够被 CFactory 创建的组件的信息。g_FactoryDataArray 中的元素为 CFactoryData 类。此类的定义如下：

```
typedefHRESULT ( * FPCREATEINSTANCE)(IUnknown * , CUnknown ** ) ;
class CFactoryData
{
public:
  // The class ID for the component
  const CLSID * m_pCLSID ;
  // Pointer to the function that creates it
  FPCREATEINSTANCE CreateInstance ;
  // Name of the component to register in the registry
  const char * m_RegistryName ;
  // ProgID
  const char * m_szProgID ;
  // Version - independent ProgID
```

```
    const char* m_szVerIndProgID;

    // Helper function for finding the class ID
    BOOL IsClassID(const CLSID& clsid) const
        {return (*m_pCLSID == clsid);}
};
```

可以看到 CFactoryData 包含五个成员变量，即组件的类标识符、指向组件创建函数的指针、将保存在 Windows 注册表中的一个易记名称、ProgID 以及一个与版本无关的 ProgID。另外它还有一个用于查找 CLSID 的帮助函数。CFactoryData 结构的填充并不是什么困难的事情。如程序清单 9-4 所列的 SERVER.CPP 所示即为一例。

程序清单 9-4　为使用类厂，必须定义一个 g_FactoryDataArray，并用每个组件的 CFactoryData 结构填充它。

SERVER.CPP

```
#include "CFactory.h"
#include "Iface.h"
#include "Cmpnt1.h"
#include "Cmpnt2.h"
#include "Cmpnt3.h"

////////////////////////////////////////////////////////////////
//
// Server.cpp
//
// This file contains the component server code.
// The FactoryDataArray contains the components that
// can be served.
//

// Each component derived from CUnknown defines a static function
// for creating the component with the following prototype.
// HRESULT CreateInstance(IUnknown* pUnknownOuter,
//                        CUnknown** ppNewComponent);
// This function is used to create the component.
//

//
// The following array contains the data used by CFactory
// to create components. Each element in the array contains
// the CLSID, the pointer to the creation function, and the name
// of the component to place in the Registry.
//
CFactoryData g_FactoryDataArray[] =
```

```
{
    {&CLSID_Component1, CA::CreateInstance,
        "Inside COM, Chapter 9 Example, Component 1", // Friendly name
        "InsideCOM.Chap09.Cmpnt1.1",          // ProgID
        "InsideCOM.Chap09.Cmpnt1"},           // Version-independent
                                              //   ProgID
    {&CLSID_Component2, CB::CreateInstance,
        "Inside COM, Chapter 9 Example, Component 2",
        "InsideCOM.Chap09.Cmpnt2.1",
        "InsideCOM.Chap09.Cmpnt2"},
    {&CLSID_Component3, CC::CreateInstance,
        "Inside COM, Chapter 9 Example, Component 3",
        "InsideCOM.Chap09.Cmpnt3.1",
        "InsideCOM.Chap09.Cmpnt3"}
};
int g_cFactoryDataEntries
    = sizeof(g_FactoryDataArray) / sizeof(CFactoryData);
```

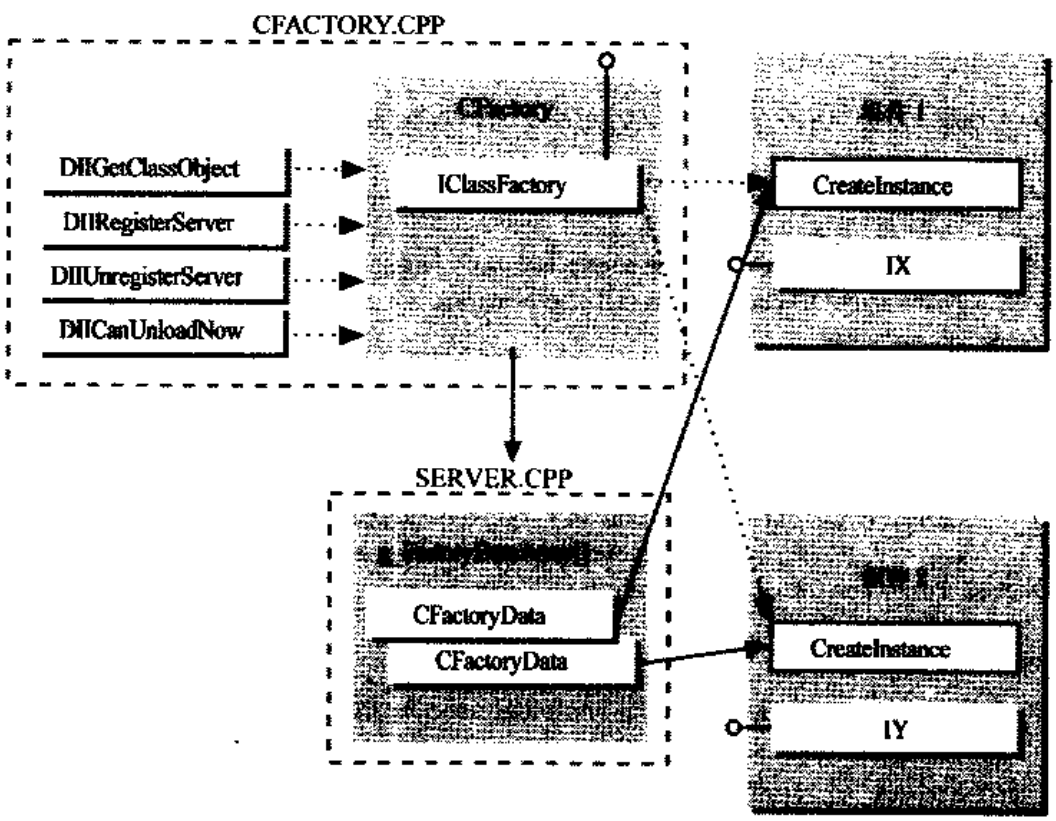


图 9-4 CFactoryData 的结构

可以看到,在程序清单 9-4 中,我们用 DLL 中包含的三个组件的信息来初始化 g_FactoryDataArray 数组的 CFactoryData 元素。

CFactory 将利用 g_FactoryDataArray 来决定它能创建哪些组件。对于在此数组中的组件,CFactory 都能创建。此时它将利用此数组以获取一个指针组件创建函数的指针。同时它还将利用 CFactoryData 数组中的信息来登记组件。图 9-4 所示为一个用 CFactory 实现的进程中组件服务器的结构。

在文件 CFACTORY.H 中,通过调用 CFactory 中的函数来实现各 DLL 入口点。DllGetClassObject 调用静态函数 CFactory::GetClassObject,而后者将在 g_FactoryDataArray 中查找与客户待创建的组件相应的 CFactoryData 结构。g_FactoryDataArray 是在文件 SERVER.CPP 中定义的,其中包含 DLL 支持的所有组件的信息。CFactory::GetClassObject 将建立组件的类厂,并将与它所创建的组件的 CFactoryData 结构传递给此类厂。

在创建了 CFactory 组件之后,它的 ICFactory::CreateInstance 将被调用。ICFactory::CreateInstance 将使用保存在 CFactoryData 数组中的创建函数实例指针来创建相应的组件。CFactory::GetClassObject 和 CFactory::CreateInstance 的实现代码如程序清单 9-5 与程序清单 9-6 所列。

程序清单 9-5 CFactory::GetClassObject 的实现。

CFACTORY.CPP 中的 GetClassObject 实现

```
//
// GetClassObject
//   - Create a class factory based on a CLSID.
//
HRESULT CFactory::GetClassObject(const CLSID& clsid,
                                 const IID& iid,
                                 void** ppv)
{
    if ((iid != IID_IUnknown) && (iid != IID_IClassFactory))
    {
        return E_NOINTERFACE;
    }

    // Traverse the array of data looking for this class ID.
    for (int i = 0; i < g_cFactoryDataEntries; i++)
    {
        const CFactoryData* pData = &g_FactoryDataArray[i];
        if (pData->IsClassID(clsid))
        {
            // Found the ClassID in the array of components we can
            // create.  So create a class factory for this component.
```

```
            // Pass the CFactoryData structure to the class factory
            // so that it knows what kind of components to create.
            *ppv = (IUnknown*) new CFactory(pData);
            if (*ppv == NULL)
            {
                return E_OUTOFMEMORY;
            }
            return NOERROR;
        }
    }
    return CLASS_E_CLASSNOTAVAILABLE;
}
```

程序清单 9-6 CFactory::CreateInstance 的实现。

CFACTORY.CPP 中的 CreateInstance 的实现

```
HRESULT _stdcall CFactory::CreateInstance(IUnknown* pUnknownOuter,
                                          const IID& iid,
                                          void** ppv)
{
    // Aggregate only if the requested IID is IID_IUnknown.
    if ((pUnknownOuter != NULL) && (iid != IID_IUnknown))
    {
        return CLASS_E_NOAGGREGATION;
    }
    // Create the component.
    CUnknown* pNewComponent;
    HRESULT hr = m_pFactoryData->CreateInstance(pUnknownOuter,
                                                &pNewComponent);
    if (FAILED(hr))
    {
        return hr;
    }
    // Initialize the component.
    hr = pNewComponent->Init();
    if (FAILED(hr))
    {
        // Initialization failed.  Release the component.
        pNewComponent->NondelegatingRelease();
        return hr;
    }
    // Get the requested interface.
```

```
    hr = pNewComponent->NondelegatingQueryInterface(iid, ppv);
    // Release the reference held by the class factory.
    pNewComponent->NondelegatingRelease();
    return hr;
}
```

上面所述就是使用 CFactory 创建组件的全部内容。开发人员要做的只不过是在实现了组件之后将其有关的数据放到一个数组中,然后就可以高枕无忧了。

9.2.3 CUnknown 和 CFactory 的使用

到此为止我们就可以复用未知接口及类厂的实现了。相信读者一定不愿意一而再、再而三地重复为 QueryInterface、AddRef 和 Release 编写相同的代码,至少作者自己是这样。从现在开始,我们的组件将不再重复实现 AddRef 和 Release,只需要将它们所需的新接口加入到其 QueryInterface 实现中。并且它们只需要实现一个实现创建函数,无需实现一个完整的类厂。新的客户代码将同程序清单 9-7 所示的代码类似。

程序清单 9-7 一个使用 CUnknown 实现 IUnknown 接口的组件。

CMPNT2.H

```
//
// Cmpnt2.h - Component 2
//
#include "Iface.h"
#include "CUnknown.h" // Base class for IUnknown

//////////////////////////////////////////////////////////////////////
//
// Component B
//
class CB : public CUnknown,
           public IY
{
public:
    // Creation
    static HRESULT CreateInstance(IUnknown* pUnknownOuter,
                                  CUnknown** ppNewComponent);

private:
    // Declare the delegating IUnknown.
    DECLARE_IUNKNOWN

    // Nondelegating IUnknown
    virtual HRESULT __stdcall
        NondelegatingQueryInterface(const IID& iid, void** ppv);
```

```
    // Interface IY
    virtual void __stdcall Fy() ;

    // Initialization
    virtual HRESULT Init() ;

    // Cleanup
    virtual void FinalRelease() ;

    // Constructor
    CB(IUnknown * pUnknownOuter) ;

    // Destructor
    ~CB() ;

    // Pointer to inner object being aggregated
    IUnknown * m_pUnknownInner ;

    // Pointer to IZ interface supported by inner component
    IZ * m_pIZ ;
} ;
```

上述程序清单为本章例子中组件 Component 2 的头文件。稍后我们将讨论相应的实现代码。在此例中，Component 1 实现的是接口 IX 本身，组件 Component 1 聚合了组件 Component 2 以提供接口 IY 和 IZ。组件 Component 2 实现了 IY 并聚合了组件 Component 3，而后者实现了接口 IZ。这样一来，组件 Component 2 既是聚合组件，同时也是被聚合组件。

下面我们从头至尾来看一下程序清单 9-7 中的代码。可以看到此组件继承了 CUnknown，CUnknown 提供了 IUnknown 的实现。然后我们定义了一个供 CFactory 用来创建组件的静态函数。CFactory 并没有对此函数的名称作什么特殊要求，因此可以为此函数取任意一个名称。

之后我们用宏 DECLARE_IUNKNOWN 实现了代理未知接口。非代理未知接口是由 CUnknown 实现的。

虽然 CUnknown 完全实现了 AddRef 和 Release，但它不能完全实现 QueryInterface，这是因为它不知道我们的组件支持的接口。因此，在组件中实现了一个 NondelegatingQueryInterface 来处理它支持的接口。

派生类可以重载 Init 函数以建立其他用于聚合或包容的组件。在 CFactory::NondelegatingRelease 删除相应的对象之前，它将调用 FinalRelease。对于那些需要释放指向内部组件指针的组件可以重载此函数。CUnknown::FinalRelease 将相应地增大引用计数以防止对组件的递归析构。

下面我们来具体看一下对于组件 Component 2，上述各个具体方面是如何实现的。此实现代码如程序清单 9-8 所列。

程序清单 9-8 一个使用了 CUnknown 和 CFactory 的组件的实现。

CMPNT2.CPP

```
//
// Cmpnt2.cpp - Component 2
//
#include <objbase.h>

#include "Iface.h"
#include "Util.h"
#include "CUnknown.h"  // Base class for IUnknown
#include "Cmpnt2.h"

static inline void trace(char * msg)
      {Util::Trace("Component 2", msg, S_OK) ;}
static inline void trace(char * msg, HRESULT hr)
      {Util::Trace("Component 2", msg, hr) ;}

//////////////////////////////////////////////////////////////
//
// Interface IY implementation
//
void __stdcall CB::Fy()
{
      trace("Fy") ;
}

//
// Constructor
//
CB::CB(IUnknown * pUnknownOuter)
: CUnknown(pUnknownOuter),
  m_pUnknownInner(NULL),
  m_pIZ(NULL)
{
      // Empty
}

//
// Destructor
//
CB::~CB()
{
      trace("Destroy self.") ;
}
```

```
//
// NondelegatingQueryInterface implementation
//
HRESULT _stdcall
      CB::NondelegatingQueryInterface(const IID& iid, void** ppv)
{
      if (iid == IID_IY)
      {
            return FinishQI(static_cast<IY*>(this), ppv) ;
      }
      else if (iid == IID_IZ)
      {
            return m_pUnknownInner->QueryInterface(iid, ppv) ;
      }
      else
      {
            return CUnknown::NondelegatingQueryInterface(iid, ppv) ;
      }
}

//
// Initialize the component and create the contained component.
//
HRESULT CB::Init()
{
      trace("Create Component 3, which is aggregated.") ;
      HRESULT hr =
            CoCreateInstance(CLSID_Component3,
                             GetOuterUnknown(),
                             CLSCTX_INPROC_SERVER,
                             IID_IUnknown,
                             (void**)&m_pUnknownInner) ;
      if (FAILED(hr))
      {
            trace("Could not create inner component.", hr) ;
            return E_FAIL ;
      }

      trace("Get pointer to interface IZ to cache.") ;
      hr = m_pUnknownInner->QueryInterface(IID_IZ, (void**)&m_pIZ) ;
      if (FAILED(hr))
      {
            trace("Inner component does not support IZ.", hr) ;
            m_pUnknownInner->Release() ;
            m_pUnknownInner = NULL ;
```

```
        return E_FAIL ;
    }

    // Decrement the reference count caused by the QI call.
    trace("Got IZ interface pointer. Release reference.") ;
    GetOuterUnknown() -> Release() ;

    return S_OK ;
}

//
// FinalRelease-Called by Release before it deletes the component
//
void CB::FinalRelease()
{
    // Call base class to incremement m_cRef and prevent recursion.
    CUnknown::FinalRelease() ;

    // Counter the GetOuterUnknown() -> Release in the Init method.
    GetOuterUnknown() -> AddRef() ;

    // Properly release the pointer, as there might be
    // per-interface reference counts.
    m_pIZ -> Release();

    // Release the contained component.
    // (We can do this now since we've released the interfaces.)
    if (m_pUnknownInner ! = NULL)
    {
        m_pUnknownInner -> Release() ;
    }
}

////////////////////////////////////////////////////////////
//
// Creation function used by CFactory
//
HRESULT CB::CreateInstance(IUnknown * pUnknownOuter,
                           CUnknown ** ppNewComponent)
{
    * ppNewComponent = new CB(pUnknownOuter) ;
    return S_OK ;
}
```

一、NondelegatingQueryInterface

组件的最有趣的部分也许就是 NondelegatingQueryInterface 了。它的实现方法同前面章节中实现 QueryInterface 的方法是类似的。但需要注意的是两个不同之处。首先一点是我们使用了函数 FinishQI,其目的纯粹是为了方便,它的作用是使在派生类中实现 NondelegatingQueryInterface 稍微容易一些。FinishQI 的实现如下所示:

```
HRESULT CUnknown::FinishQI(IUnknown * pI, void ** ppv)
{
      * ppv = pI ;
      pI -> AddRef() ;
      return S_OK ;
}
```

另外一个差别是,我们不一定要处理 IUnknown。IUnknown 和其他并不知道的接口将由基类来处理:

```
HRESULT _stdcall
  CUnknown::NondelegatingQueryInterface(const IID& iid, void ** ppv)
{
      // CUnknown supports only IUnknown.
      if (iid == IID_IUnknown)
      {
            return FinishQI(reinterpret_cast < IUnknown * >

      (static_cast < INondelegatingUnknown * > (this)),ppv) ;
      }
      else
      {
            * ppv = NULL ;
            return E_NOINTERFACE ;
      }
}
```

9.2.4 集成步骤

从前面的代码中我们可以看到,使用 CUnknown 和 CFactory 来编写组件是相当简单的。下面我们将对此过程作一总结,这就是下面建立组件、类厂及包含它的 DLL 的一些步骤:

1 编写实现组件的类。

- 可以从 CUnknown 或其他从 CUnknown 派生的类派生出待实现的组件。
- 使用 DECLARE_IUNKNOWN 宏来实现代理未知接口。

- 在组件的构造函数中初始化 CUnknown。
- 实现 NondelegatingQueryInterface，在其中加入此组件支持而其基类不支持的接口。对于那些组件所不支持接口，可调用相应的基类。
- 若需要在构造了组件之后进行其他一些初始化处理，可重载 Init 函数，如此时可建立被包容及被聚合的组件。
- 若需要在组件被删除之前进行其他一些清理工作，可重载 FinalRelease 函数，如此时可以释放那些指向被包容或聚合的组件的指针。
- 给组件实现一个静态的 CreateInstance 函数。
- 实现组件支持的那些接口。

2 对于待放到同一 DLL 中的其他组件，重复上面的步骤 1。

3 编写类厂。

- 建立一个文件，以包含全局 CFactoryData 数组 g_FactoryDataArray。
- 定义 g_FactoryDataArray 数组并用 DLL 中提供的所有组件的信息填充此组件。
- 定义变量 g_cFactoryDataEntries，其中包含有数组 g_FactoryDataArray 中组件的个数。

4 编写一个定义 DLL 入口点的 DEF 文件。

5 将上面所编写的代码同 CUNKNOWN.CPP 和 CFACTORY.CPP 一块编译链接。

可以看到整个过程是相当简单的。按照上述的步骤，我们可以在几分钟之内编写好一个新的组件。

9.3 本章小结

在有了智能接口指针类之后，COM 组件的使用就同 C++ 类的使用差不多了，这主要是由于智能接口指针将引用计数隐藏起来了，并且智能接口指针还使得我们可以按一种类型安全的方法来获取接口指针。在智能接口指针类的实现中，一般的作法是重载运算符 =，以在将一种类型的接口指针赋给另外一种类型的接口指针时调用 QueryInterface。

同智能指针类能使 COM 对象的使用更为容易类似，另外两个简单的 C++ 类 CUnknown 和 CFactory 可以使 COM 组件的实现尽可能地简化。这一点是通过提供可复用的 IUnknown和 ICFactory 的实现而达到的。

第 10 章　EXE 中的服务器

作者上一次去柏林时，柏林墙还没有被推倒。当作者离开 Checkpoint Charlie 的 American 防区前往东柏林时，可以明显地感觉到跨越了一个边界。那些带刺的铁丝网、雷区以及身背机关枪的士兵可以使人能够明确地感觉到这条边界的存在。即使在远离防御线的地方，仍然可以看到一些能够告诉我们这条边界的存在的景象：在墙的东侧，一些二冲程的小汽车吐着油烟在街上奔跑着，而商店外面则满是排着长队的人群。

在跨越边界时，不论边界两边的差别是多么微小，都将可以感受到事情将会发生一些变化。本章我们讨论的就是当跨越边界时——主要是进程间的边界——会出现一些什么问题以及如何解决这些问题。另外我们还将讨论在跨越机器边界时的处理方法。

那么为什么需要跨越进程边界呢？这是因为在某些情况下，需要用 EXE 而不是 DLL 来实现组件。使用 EXE 实现组件的原因之一是应用程序可能已经就是一个 EXE 文件了。在这种情况下，稍加努力就可以使应用程序的服务“暴露”出来，以便客户能够使用它。

不同 EXE 中的组件和客户将在不同的进程中运行，这是因为每一个 EXE 都有其自己的进程空间。这样一来客户和组件之间的交互就会跨越进程边界了。所幸的是，这并不需要对组件的代码进行修改，但却确实需要对上一章中介绍的 CFactory 的代码进行一些修改。在讨论此实现之前，我们先来看一下使用 COM 接口进行跨越进程边界的交互时会产生哪些问题以及如何解决这些问题。

10.1　不同的进程

每一个 EXE 文件都将在不同的进程中运行，而每一个进程都有其自己的进程空间。一个进程空间中的逻辑地址 0x0000ABBA 所对应的物理地址将不同于另外一个进程中同一逻辑地址所对应的物理地址。因此若一个进程将地址 0x0000ABBA 传给另外一个进程，后者用此地址所访问到的内存单元将不是前一进程所希望的(如图 10-1 所示)。

同每一个 EXE 都有其自己的进程不同，DLL 将被映射到链接它们的 EXE 文件的进程空间中。由于这个原因，DLL 也被称作是进程中服务器，而 EXE 则被称作是进程外服务器。在某些情况下，EXE 也被称作是本地服务器以同另外一种类型的进程外服务器“远程服务器”相区别。远程服务器指的是运行于另外一个不同的机器上的进程外服务器。

在第 5 章中我们介绍了组件同其客户共享相同的地址空间的重要性。组件将把一个接口传给客户。一个接口实际上是一个函数指针的数组。为调用接口中的函数，客户必

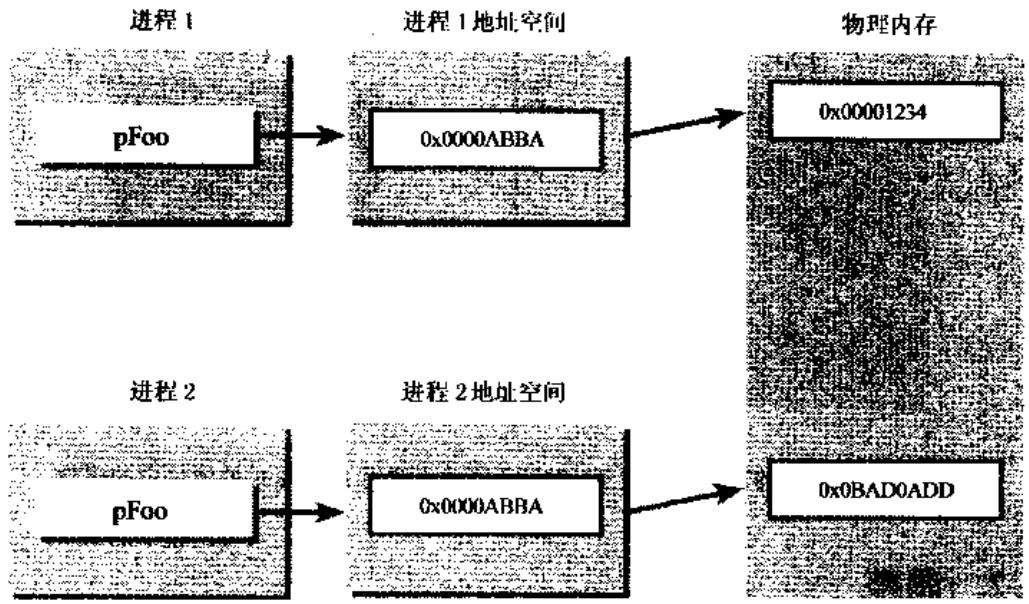


图 10-1 不同进程空间中的相同的内存地址将指向不同的物理内存单元

须能够访问同接口相关联的内存。若组件是一个 DLL,那么这一点是很容易实现的,因为组件和客户处于相同的地址空间中。但如果组件和客户在不同的地址空间中,客户将无法访问组件进程的地址空间。这样一来客户连访问同接口相关联的内存都无法做到,那么调用接口中的函数自然就更不可能了。如果真是这样的话,我们费了那么大力气构造的接口将没有任何用处了。

对于跨越进程边界的接口,我们需要考虑如下一些条件:

- 一个进程需要能够调用另外一个进程中的函数。
- 一个进程需要能够将数据传递给另外一个进程。
- 客户无需关心它所访问的服务器是进程内服务器还是进程外服务器。

10.1.1 本地过程调用

对于进程间的通信,有几种不同的方法,如动态数据交换(DDE)、命名管道以及共享内存等。COM 所用的方法则为本地过程调用(LPC)。LPC 是同一机器上不同进程间通信的一种方法,它是基于远程过程调用(RPC)的用于单机上进程间通信的专利技术。

RPC 标准是在开发软件基金会(OSF)分布式计算环境(DCE) RPC 规范中定义的,它使得不同机器上的进程可以使用各种网络传输技术进程通信。本章后面我们将要介绍的分布式 COM(DCOM)就将使用 RPC 进程跨网络的通信。

那么 LPC 是如何工作的呢? 它们是由操作系统实现的。这一点看似有些神奇,实际上并非如此。由于操作系统知道同每一个进程逻辑地址空间相对应的物理地址,因此操作系统可以调用任意进程中的任意函数。

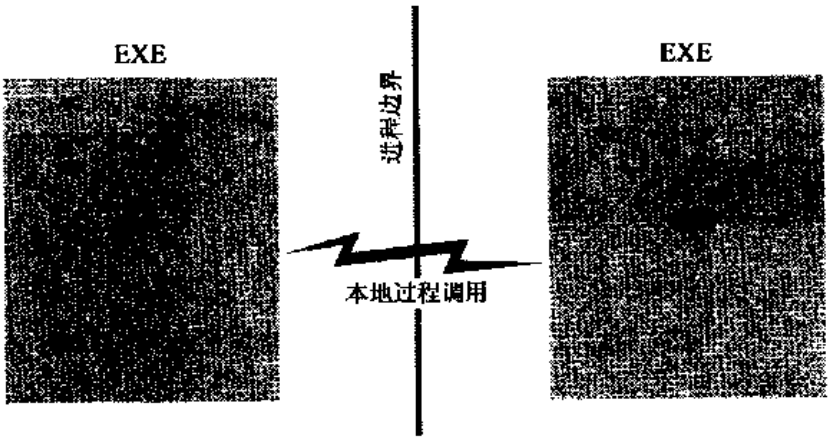


图 10-2 EXE中的客户将通过 Win32LPC 技术来调用 EXE 形式的组件中的函数

10.1.2 调整

调用 EXE 中的函数只是第一步。另外我们还需要一种方法将函数调用的参数从一个进程的地址空间中传到另外一个进程的地址空间中。这种方法被称作是"调整"。

若两个进程都在同一台机器上,则调整过程将是相当直接的:只需将参数数据从一个进程的地址空间复制到另外一个进程的地址空间就可以了。若参与参数传递的两个进程在不同的地址空间中,那么考虑到不同机器在数据表示方面的不同,如整数的字节顺序可能会不一样,必须将参数数据转换成标准的格式。

LPC 技术可以将数据从一个进程复制到另外一个进程中。但为了完成这一点,仅仅用一个 C++ 头文件是不够的,还需要其他一些信息以将参数打包并传送给其他进程。例如对指针的处理就将不同于对整数的处理。对指针的调整指的是要将指针引用的结构复制到另外一个进程中。但若此指针是一个接口指针,则将不能复制此指针所引用的内存。这里我们看到,调整并不是用一条简单的 memcpy 命令就能完成的。

为对组件进行调整,可以实现一个名为 IMarshal 的接口。在 COM 创建组件的过程中,它将查询组件的 IMarshal 接口。然后它将调用 IMarshal 的成员函数以在调用函数的前后调整或反调整有关的参数。COM 库中实现了一个可以供大多数接口使用的 IMarshal 的标准版本。在需要对性能进程优化时,可以对 IMarshal 进行定制。Kraig Brochschmidt 所著《Inside OLE》对这个问题有详细的讨论。

10.1.3 代理/残根 DLL

前面九章并没有详细讨论如何通过一个接口来调用 COM 组件,而只是稍微提及此种调用是通过 LPC 完成的。从最开始我们就有一个明确的目标,那就是客户应该可以按照相同的方式与进程中、本地及远程组件进行通信。显然,若客户必须自己处理 LPC 的问

题,这一目标将无法实现。为达到这一目标,COM 使用了一种非常简单的方法。

大多数 Windows 程序员可能并不知道,几乎在每次调用 Win32 函数时,他们都用到了 LPC。当调用 Win32 函数时,系统实现上将调用一个 DLL 中的函数,而此函数将通过 LPC 调用 Windows 中的实际代码。这种结构可以将用户进程同 Windows 代码隔离开。由于不同的进程具有不同的地址空间,因此用户程序不可能对操作系统造成破坏。

COM 使用的结构与此类似。客户将同一个模仿组件的 DLL 进行通信。这个 DLL 可以为客户完成参数的调整及 LPC 调用。在 COM 中,此 DLL(也是一个组件)被称作是一个代理。

用 COM 的术语来说,一个代理就是同另外一个组件行为相同的组件。代理必须是 DLL 形式的,因为它们需要访问客户进程的地址空间以便对传给接口函数的数据进行调整。对数据的调整只完成了任务的一半,组件还需要一个被称作是残根的 DLL,以对从客户传来的数据进行反调整。残根也将对传回给客户的数据进行调整(参见图 10-3)。

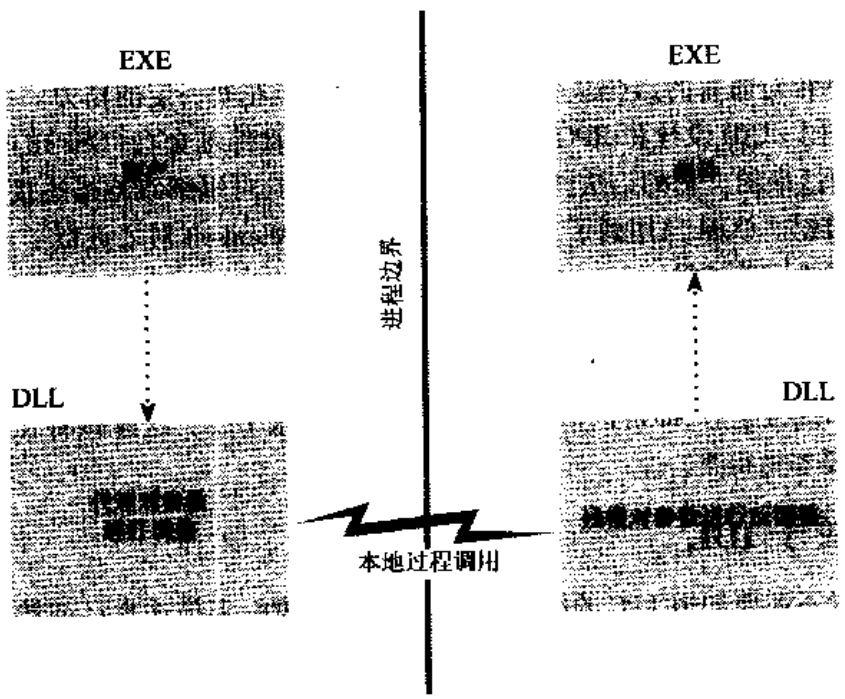


图 10-3 客户可以同一个代理 DLL 进行通信。代理将对函数参数进行调整并通过 LPC 来调用残根 DLL。残根 DLL 将对这些参数进行反调整,并将其传给组件中合适的函数以调用之

当然图 10-3 对整个过程进行了大大的简化。当然它能够充分说明为使这种机制能够正常运转,需要编写大量的代码,如代理及残根 DLL 等。同时我们还需要了解一下 LPC,以便能够进行跨越进程边界的函数调用。另外还需要实现一个调整组件和客户之

间数据的 IMarshal 接口。所幸的是,所有这些大量的工作都不需要我们自己来亲自完成,至少在大多数情况下是如此。

10.2 IDL/MIDL 简介

对于程序员而言,他们可能更喜欢编写具有某些特殊功能的组件,而不想将精力花费在如何使这些组件能够相互通信上。作者自己宁愿将时间花费在编写 OpenGL 程序上,而不想在将数据从一个程序移动到另外一个程序上花费什么功夫。所幸的是,程序员并不需要自己来编写这些代码。借助于一种名为 IDL(接口定义语言)的语言,我们可以编写接口的一个描述,然后用 MIDL 编译器即可生成代理和残根 DLL。

当然,如果程序员自己愿意做这些事情,那是再好不过的了。这正是 COM 的一个优点:它给大多数接口提供了一个缺省的实现,但同时又允许开发人员给出自己的实现。但大多数接口不需要对调整过程进行定制,并且大多数组件也不需要自己编写的代理和残根 DLL。因此我们仍然可以用比较简单的方法来完成有关的编程工作。当然这里所说的简单的方法是相对而言的,它实际上仍然需要开发人员花费一定的精力。

IDL 语言同 UUID 设计和 RPC 规范一样是从开放软件基金会(OSF)的分布式计算环境(DCE)借用过来的。它的语法同 C 和 C++ 是一样的,可以细致地描述接口和组件所共享的接口和数据。COM 只用到了 IDL 的一个子集,但 Microsoft 对之进行了一些扩展,以便使之能够更好地支持 COM。

在用 IDL 对接口和组件进行了描述之后,可以通过 MIDL(Microsoft 的 IDL 编译器)来编译它。MIDL 将接收接口的 IDL 描述,并生成相应代理和残根 DLL 的 C 代码。对这些代码进行编译和链接之后,即可以得到相应的代理和残根 DLL。这样做将比自己手工实现这两个 DLL 要容易得多。

10.2.1 关于 IDL

虽然读者不需要知道 LPC 的细节,但知道如何用 IDL 来描述自己的接口却是必不可少的。用 IDL 来描述接口可能并没有什么困难之处,但却容易使开发人员感受到挫折之苦,这主要是因为 IDL 并不是一致的,文档资料都编写得比较差,好的例子又极难找到,同时给出的错误信息也都比较模棱两可。作者的看法是:尽力找出一个例子。由于 IDL 会节省开发者极大的时间和精力,因此不要对此有任何抱怨。比较好的作法是花一点时间仔细阅读一下 MSDN CD 上的 IDL 文档。这些文档虽然比较干巴无味,但预先阅读一下总比由于不知道怎么用 IDL 来描述接口而耗费大量的时间要强得多。

此时读者可能还是想着自己手工来编写有关的代码,而不想用 IDL。但在下一章中我们将使用 IDL 编译器来建立类型库。虽然这些类型库也可以手工建立,但实际上这样做没有任何好处可言。一句话,还是花一点时间去学习一下 IDL,因为这样我们将能够同

时得到代理代码和类型库。

10.2.2 IDL 接口描述举例

下面我们来看一下用 IDL 描述的一个接口。如下面的从第 10 章的例子中摘出来的一段代码所示：

```
import "unknwn.idl";
// Interface IX
[
    object,
    uuid(32bb8323-b41b-11cf-a6bb-0080c7b2d682),
    helpstring("IX Interface"),
    pointer_default(unique)
]
interface IX : IUnknown
{
    HRESULT FxStringIn([in, string] wchar_t* szIn);
    HRESULT FxStringOut([out, string] wchar_t** szOut);
};
```

在 C++ 中与上述代码相应的函数将如下所示：

```
virtual HRESULT __stdcall FxStringIn(wchar_t* szIn);
virtual HRESUTL __stdcall FxStringOut(wchar_t** pszOut);
```

可以看到 IDL 的语法同 C++ 并无太大的差异。最明显的不同可能是 IDL 用方括号([])来作为信息分隔符。在每一个接口定义的前面，都有一个属性列表或称接口头。在前面给出的例子中，接口头包含四项内容。其中 object 表示所定义的接口为一个 COM 接口，关键字 object 是 Microsoft 对于 IDL 的一个扩展。第二个关键字 uuid 为相应接口的 IID。第三个关键字用于将一个帮助串放到一个类型库中。关于类型库，由于它同进程外服务器没有什么直接关系，我们将在下一章中讨论。第四个关键字 pointer_default 显得有些让人迷惑不解，我们将在下面详细讨论。

一、pointer_default 关键字

使用 IDL 的目的是为了提供足够的信息，以便函数参数可以被调整。为此，IDL 需要知道如何处理指针。pointer_default 关键字的作用就是告诉 MIDL 编译器在没有为指针指定其他属性时如何处理此指针。pointer_default 关键字具有三个不同的选项：

- ref——将指针当成是引用对待。此时表示此指针将总是指向一个合法的地址，并可被反引用。这种指针不能为空。在调用前后它们将指向同一内存地址。在函数内部，不能为它们指定别名。

- unique——此类指针可以为空，并且在函数内可以修改它们的值，但不能为之指定别名。
- ptr——此选项指定相应的指针就是一个 C 指针。此类指针可以是有一个别名、可以为 NULL，并且其值可以被修改。

MIDL 将使用这些值对它生成的代理及残根代码进行优化。

二、IDL 中的输入与输出参数

MIDL 还可以使用 in 和 out 参数属性对代理及残根代码进行进一步的优化。对于被标记为 in 的参数，MIDL 将知道仅仅需要将此参数值从客户传递给组件，残根代码不需要送回任何值。out 关键字告诉 MIDL 相应的参数仅被用来从组件向客户传回有关的数据。代理不需要对输出参数值进行调整，也不需要将此值传送给组件。

可以同时用此这两个关键字来标记某个参数，如下所示：

```
HRESULT foo([in] int x, [in, out] int * y, [out] int * z);
```

在上面的例子中，参数 y 既是一个输入参数，也是一个输出参数。对于输出参数，MIDL 要求它必须是一个指针。

三、IDL 中的字符串

在对某块数据进行调整时，必须知道此块数据的大小，这样才能够复制它。对于 C++ 串，决定其长度是很容易的，这只需找出末尾的那个空字符即可。通过在函数中加上一个 string 修饰符，MIDL 将知道相应的参数为一个字符串，并且它可以通过查找末尾的空字符而决定其长度。

COM 中对于字符串的标准约定是使用 Unicode 字符即 wchar_t。在那些内部实现上并不支持 Unicode 的平台上也将如此。这就是我们前面为什么将字符串定义为 wchar_t 的原因。但也可以不使用 wchar_t 而使用在 COM 头文件中定义的 OLECHAR 或 LPOLESTR。

四、IDL 中的 HRESULT 值

可以注意到前面给出的 IDL 例子中的两个函数都将返回一个 HRESULT 值。这是由于 MIDL 要求用 object 修饰符标记的接口返回 HRESULT 值。之所以要求这一点，主要原因在于远程服务器。当连接到一个远程服务器时，任何函数调用都可能由于网络传输的问题而失败。因此应该有一种让所有函数都能指示网络错误的方法。最简单的方法就是让所有的函数返回一个 HRESULT 值。

因于这个原因，大多数 COM 函数返回的均为一个 HRESULT 值。（有些程序员喜欢为 COM 组件编写一些包装类，以当某个成员函数返回失败代码时产生一个异常。实际上，

Microsoft Visual C++ 5.0编译器可以导入一个类型库,并自动为那些成员生成相应的包装类,以在它们返回一个失败的 HRESULT 值时产生一个异常。)若某个函数需要返回一个不是 HRESULT 类型的值,那么可以在相应的地方加上一个输出参数。在 FxStringOut 中,我们定义了一个输出参数以从组件得到一个返回串。函数 FxStringOut 将使用 CoTaskMemAlloc 来分配返回串所需的内存。客户必须使用 CoTaskMemFree 来释放这块内存。下面的代码显示了如何使用前面定义的 IX 接口。

```
wchar_t* szOut = NULL;
HRESULT hr = pIX->FxStringIn(L"This is the test.");
assert(SUCCEEDED(hr));
hr = pIX->FxStringOut(&szOut);
assert(SUCCEEDED(hr));

// Display returned string.
ostrstream sout;
sout << "FxStringOut returned a string:"
    << szOut // Use overloaded operator << for wchar_t.
    << ends;
trace(sout.str());

// Free the returned string.
::CoTaskMemFree(szOut);
```

可以看到我们使用了 CoTaskMemFree 来释放返回的字符串所占用的内存。

五、IDL 中的 import 关键字

import 用于将其他 IDL 文件中的定义包含到当前文件中。例如在前面给出的例子中,就包含了 IUnkown 接口的 IDL 描述 UNKNWN.IDL。import 同 C++ 的预处理指令 #include 命令是类似的,只不过 import 可以使用任意多次,而不会引起重复定义的问题。所有 COM 及 OLE(AcitveX)的标准接口都定义于相应的 IDL 文件中。这些文件一般位于 C++ 编译器的 \INCLUDE 文件夹中。仔细阅读这些标准的 OLE 接口的 IDL 文件将是提高用 IDL 编写接口描述的一种极好方法。

六、IDL 中的 size_is 修饰符

下面我们来看一下在客户和组件之间传递数组的接口的 IDL 描述:

```
// Interface IY
[
    object,
    uuid(32bb8324-b41b-11cf-a6bb-0080c7b2d682),
    helpstring("IY Interface"),
    pointer_default(unique)
]
```

```
interface IY : IUnknown
{
    HRESULT FyCount([out] long* sizeArray);
    HRESULT FyArrayIn([in] long sizeIn,
        [in, size_is(sizeIn)] long arrayIn[]);
    HRESULT FyArrayOut([out, in] long* psizeInOut,
        [out, size_is(*psizeInOut)] long arrayOut[]);
};
```

此接口的头文件同前一个接口的是相同的。这里我们感兴趣的是 size_is 修饰符。我们知道调用的一个主要作用是将数据从一个位置复制到另外一个位置。因此知道待复制的数据的大小是非常重要的。若此大小是固定的,将不会有任何问题。但若被复制的数据的大小在运行时才能被确定,情况就会变得复杂一些。例如当将一个数组传给一个函数时,代理将如何决定此数组的大小呢?

这就是为什么要引入 size_is 修饰符的目的。在前面代码中的 FyArrayIn 函数中,size_is 修饰符将告诉 MIDL 数组中元素的个数将被保存在 sizeIn 中。提供给 size_is 修饰符的参数只能是输入参数或输入-输出参数。在 IY 接口的第二个函数 FyArrayOut 中,我们就用了一个输入-输出参数作为 size_is 的参数。

客户在通过第二个参数传入一个分配的数组,而数组中元素的个数则通过第一个参数 psizeOut 传入。此函数将用其内部数组填充此数组,然后它将把 psizeOut 设置为实际填充到数组中元素的个数。

就作者自己而言,一般是不太喜欢用输入-输出参数的,并且也绝不会设计出类似上面给出的接口。相反,我们可以定义一个单独的输出参数,以返回填充的数组元素的个数,如下所示:

```
HRESULT FyArrayOut2([in] long sizeIn,
        [out, size_is(sizeIn)] long arrayOut[],
        [out] long* psizeOut);
```

下面的代码是从本章的例子中摘出来的,首先它将通过 IY 接口给组件传递一个数组,然后将组件返回的数组元素显示出来。

```
// Send an array to the component.
long arrayIn[] = { 22, 44, 206, 76, 300, 500 };
long sizeIn = sizeof(arrayIn) / sizeof(arrayIn[0]);
HRESULT hr = pIY->FyArrayIn(sizeIn, arrayIn);
assert(SUCCEEDED(hr));

// Get the array back from the component.

// Get the size of the array.
long sizeOut = 0;
```

```
hr = pIY->FyCount(&sizeOut) ;
assert(SUCCEEDED(hr)) ;

// Allocate the array.
long * arrayOut = new long[sizeOut] ;

// Get the array.
hr = pIY->FyArrayOut(&sizeOut, arrayOut) ;
assert(SUCCEEDED(hr)) ;

// Display the array returned from the function.
ostrstream sout ;
sout << "FyArray returned"
    << sizeOut
    << "elements:" ;
for (int i = 0 ; i < sizeOut ; i++)
{
    sout << " "<< arrayOut[i] ;
}
sout <<"." << ends ;
trace(sout.str()) ;

// Cleanup
delete [] arrayOut ;
```

根据 COM 规范,在为输出参数分配内存时,都需要使用 CoTaskMemAlloc 函数。但许多COM 接口并不使用这一函数。与 IY::FyArrayOut 最相似的是 IxxxxENUM::Next,它也没有使用 CoTaskMemAlloc。COM 最让人气馁的可能要算是某些 COM 库函数使用了 CoTaskMemAlloc,而另外一些库函数则不使用它。并且从文档中很难看出来,哪些使用了,而哪些没有使用。例如比较一下 StringFromCLSID 和 StringFromGUID2 的文档,很难看出来哪个需要用 CoTaskMemFree 来释放内存。在不能决定这一点时,可以参考第 6 章中的内容。

七、IDL 中的结构

在 C++ 程序中传递的一定不只是简单类型的数据,而且也传递一些复杂的结构。在 IDL 文件中也可以定义 C 和 C++ 风格的结构,并可用它们作为函数的参数。例如下面的接口将使用一个包含三个域的结构:

```
// Structure for interface IZ
typedef struct
{
    double x ;
    double y ;
    double z ;
} Point3d ;
```

```
// Interface IZ
[
    object,
    uuid(32bb8325-b41b-11cf-a6bb-0080c7b2d682),
    helpstring("IZ Interface"),
    pointer_default(unique)
]
interface IZ : IUnknown
{
    HRESULT FzStructIn([in] Point3d pt);
    HRESULT FzStructOut([out] Point3d* pt);
};
```

从这里我们再一次看到,IDL 同 C++是非常类似的。但在传递包含指针的复杂结构时,情况也会变得复杂得多。MIDL 需要精确地知道一个指针所指的到底是什么内容,这样它才能够对指针所指的数据进行调整。由于这个原因,绝不要使用 void* 作为参数。若需要传递一个一般性的接口指针,可以使用 一个 IUnkown* 接口。最灵活的方法是让客户传递一个 IID,这同 QueryInterface 是相似的:

```
HRESULT GetIface([in]constIID& iid, [out, iid_is(iid)]IUnknown** ppI);
```

此处的 IDL 属性 iid_is 的作用是告知 MIDL 相应接口的 IID 为何。当然我们也可以用如下的方法:

```
HRESULT GetMyInterface([out] IMyInterface** pIMy);
```

但这种方法有一个问题就是当我们换用另外一个接口,比如说是 IMy2 或 IMyNewAndVastlyImproved 时上述方法就难以处理了。

10.2.3 MIDL 编译器

在编写好一个 IDL 文件之后,即可用 MIDL 编译器来编译之以生成相应的代码。例如如果有一个描述我们接口的文件 FOO.IDL,则可以用如下的命令编译此文件:

```
midl foo.idl
```

执行此命令之后,将生成如表 10-1 所示的那些文件。

当 IDL 文件中有一个 library 语句时,MIDL 命令将生成一个类型库。(关于类型库我们将在下一章中讨论)。图 10-4 显示了 MIDL 编译器生成的文件的图示。此图也显示了如何将这些文件组合起来以生成代理 DLL。关于此过程我们将在后面讨论。

一、示例程序的建立

下面我们把前面的文件组合起来以生成一个示例程序。为此需要编写一个 makefile。

表 10-1 MIDL 编译器生成的文件

文件名	内容
FOO.H	一个同 C 和 C++ 兼容的、包含 IDL 中描述的所有接口声明的头文件。可以用两个等价的开关/header 或/h 改变此头文件的名称。
FOO_I.C	一个定义有 IDL 文件中所用的所有 GUID 的 C 文件。可以使用/iid 开关改变此文件的名称。
FOO_P.C	一个实现 IDL 文件中接口的代理及残根的 C 文件。此文件的名称可以用/proxy加以修改。
DLLDATA.C	一个实现包含代理及残根的 DLL 的 C 文件。此文件的名称可以用/dlldata 开关加以修改。

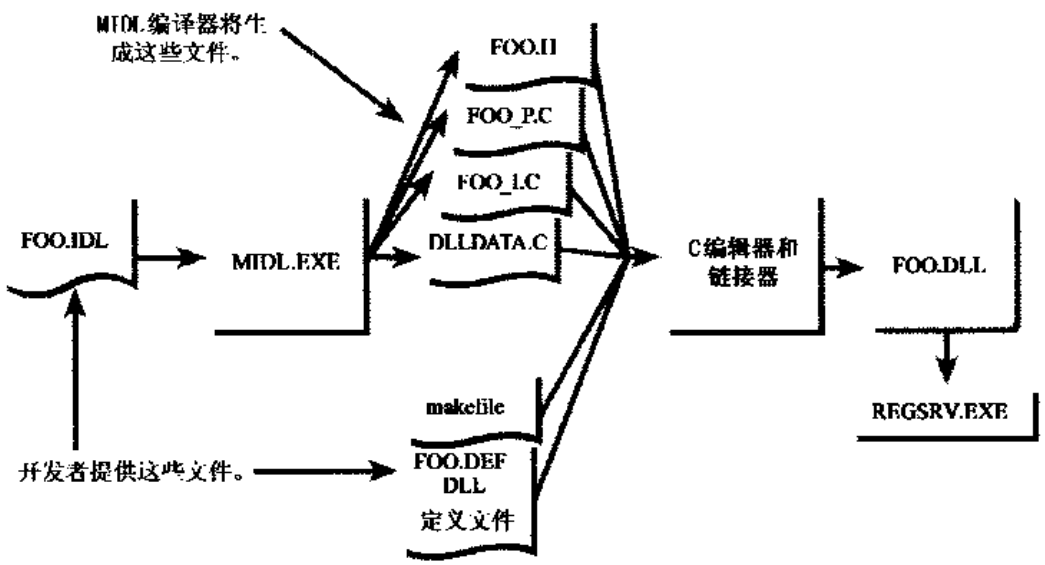


图 10-4 使用 MIDL 生成并使用相应文件的过程

此 makefile 最好能够生成两个不同版本的组件服务器：SERVER.DLL 和 SERVER.EXE。如下面的 makefile 的所示即为一例：

```
always:
    nmake -f make - one
    nmake -f make - one OUTPROC = 1
```

其中 make - one 将完成所有有关文件的编译和链接。为建立组件的两个不同版本，可以输入如下命令：

```
nmake - f makefile
```

在编写 make - one 时，可以将进程中服务器的中间文件放到 \ INPROC 文件夹中，而

将进程外服务器的中间文件放在 \ OUTPROC 文件夹中。

在 make - one 中运行 MIDL 的命令可以如下：

```
midl /h iface.h /iid guids.c /proxy proxy.c server.idl
```

这样一来，MIDL 生成的文件的名称将被修改成与我们前面使用的名称相匹配。为了避免在 IFACE.H 和 SERVER.IDL 中重复编写接口的定义，我们将接口定义放到了文件 SERVER.IDL 中，并用它来生成 IFACE.H。与此类似，那个包含 GUID 的文件 GUIDS.CPP 也不再需要了，而只需使用 GUIDS.C 即可。

MIDL 编译器生成的头文件既可用于 C 程序，也可用于 C++ 程序。这些头文件最大的不足在于它们几乎是没法阅读的。不信的话，读者可以看一下为 SERVER.IDL 生成的头文件 IFACE.H，就会发现要弄清楚这些文件将是一件多么困难的事情。但这总比在多个地方对接口进行维护要好得多。并且还有一点是，C 程序员可以用这些头文件，C++ 程序员也可以用。

二、代理 DLL 的建立

为得到一个代理/残根 DLL，需要编译和链接 MIDL 生成的头文件。MIDL 编译器将为组件生成相应的代理和残根的代码。但将这些代码编译成一个 DLL 仍然是程序员的工作。为此第一步需要为代理 DLL 编写一个 DEF 文件。当然这一步是非常简单的，如下所示：

```
LIBRARY     Proxy.dll
DESCRIPTION  'Proxy/Stub DLL'
EXPORTS
    DllGetClassObject       @1 PRIVATE
    DllCanUnloadNow         @2 PRIVATE
    GetProxyDllInfo         @3 PRIVATE
    DllRegisterServer       @4 PRIVATE
    DllUnregisterServer     @5 PRIVATE
```

接下来的工作就是编译和链接。为此可以在 make - one 使用如下的代码：

```
iface.h server.tlb proxy.c guids.c dlldata.c : server.idl
    midl /h iface.h /iid guids.c /proxy proxy.c server.idl

dlldata.obj : dlldata.c
    cl /c /DWIN32 /DREGISTER _PROXY _DLL dlldata.c

proxy.obj : proxy.c
    cl /c /DWIN32 /DREGISTER _PROXY _DLL proxy.c

PROXYSTUBOBJS = dlldata.obj \
        proxy.obj   \
```

```
        guids.obj

PROXYSTUBLIBS= kernel32.lib \
        rpcndr.lib   \
        rpcns4.lib   \
        rpcrt4.lib   \
        uuid.lib

proxy.dll : $(PROXYSTUBOBJS) proxy.def
    link /dll /out:proxy.dll /def:proxy.def \
        $(PROXYSTUBOBJS) $(PROXYSTUBLIBS)
    regsvr32 /s proxy.dll
```

三、代理/残根的登记

注意上面的 makefile 文件在编译文件 DLLDATA.C 和 PROXY.C 时定义了一个宏 REGISTER_PROXY_DLL。这个宏将生成完成代理/残根的登记所需的代码。从上面的代码可以看到,在生成代理 DLL 之后此 DLL 将被登记。这可以保证代理 DLL 的登记不会被遗漏掉。否则可能会花费许多时间和精力也弄不清楚为什么程序不能正常工作。

那么代理/残根 DLL 登记在注册表中的内容是什么呢? 下面来看一下我们的例子。在建立好示例程序之后,makefile将自动地登记代理和服务器,这样开发人员就可以不必

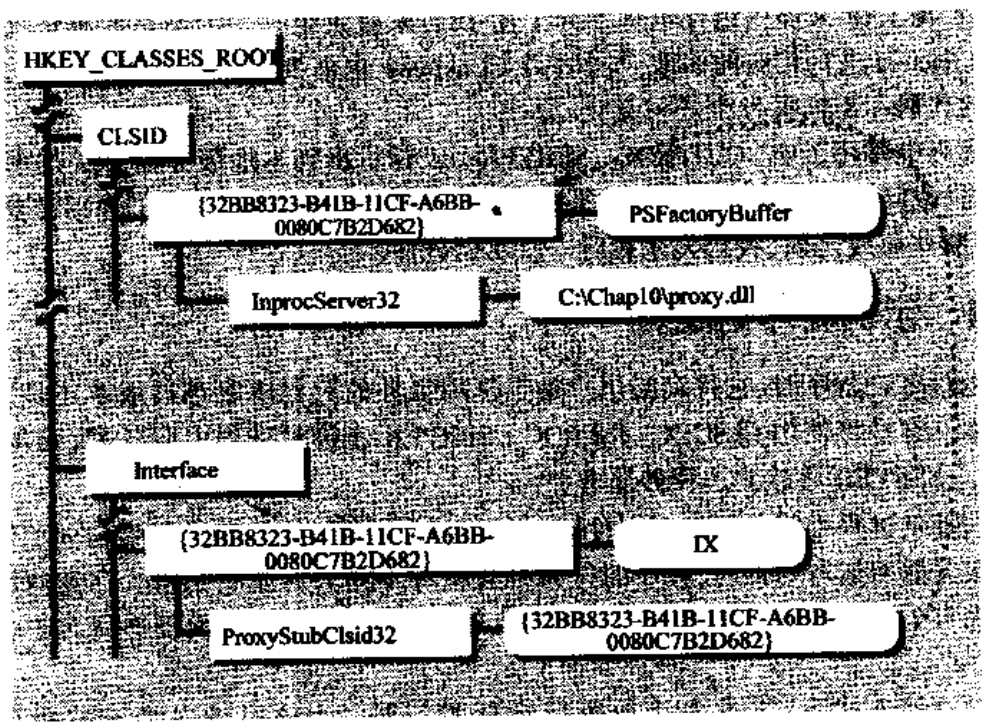


图 10-5 MIDL-生成的代理/残根代码所加入的注册表项的结构

完成这些处理。另外也可以用文件 REGISTER.BAT 来登记程序的预编译版本。

下面我们来运行 REGEDIT.EXE,并看一下下述的注册表关键字:

```
HKEY_CLASSES_ROOT\
    Interfaces\
        {32BB8323-B41B-11CF-A6BB-0080C7B2D682}
```

上面所列的 GUID 为接口 IX 的 IID。在此注册表关键字的下面我们将看到另外几个项。最有意思的一项是 ProxyStubClsid32。此关键字包含此接口的代理/残根 DLL 的 CLSID。此值对于接口 IX、IY 和 IZ 都是相同的。当在 HKEY_CLASSES_ROOT\CLSID 下查找此 CLSID 时,将可以找到一个值为 PROXY.DLL 的 InprocServer32 关键字。这里我们看到,接口同实现它们的组件是单独登记的(参见图 10-5)。

至此可以看到,有了 MIDL 之后,就可以像调用进程中组件那样进行跨进程边界的函数调用并对参数进行调整。

10.3 本地服务器的实现

下面我们需要讨论一下为支持进程外组件需要对 CFactory 进行哪些修改。常言道:入乡随俗。在跨越边界时,都需要相应地改变自己的行为与习惯以与新的环境相适应。与此类似,用 EXE 来提供组件也将不同于用 DLL 来提供组件。为满足此种情况的变化,必须对 CFactory 进行相应的修改。同时对 CUnknown 也需要进行一些细微的修改。但组件本身的代码却不需要发生任何变化。

在代码中使用了宏_OUTPROC_SERVER_以标示出同本地服务器相关的部分(当定义此宏时)或同进程中服务器相关的部分(当没有定义此宏时)。在我们介绍需要对 CFactory 进行哪些修改之前,我们先来运行一下示例程序。

10.3.1 示例程序的运行

当运行客户程序时,它将询问用户是想使用组件的进程中服务器版本还是进程外服务器版本。客户将使用 CLSCTX_INPROC_SERVER 连接到进程中服务器,而用 CLSCTX_LOCAL_SERVER 连接到进程外服务器。

当选择从进程中服务器运行组件时,程序的行为将同前一章介绍的相同。但是当以进程外服务器来运行组件时,程序的行为将会稍有不同。首先我们将会注意到程序的输出将仅来自于客户。这主要是由于组件在另外一个进程中运行,而没有同客户共享控制台窗口。

在后一种方式下,需首先从命令行运行服务器。为此可输入如下的命令如双击 SERVER.EXE 的图标。

```
c:\>start server
```

此时服务器将开始运行并显示出一个窗口。下面即可以运行客户程序并让其连接到本地服务器。此时客户程序将把输出送到控制台窗口，而本地服务器的输出则将被显示在它自己的窗口中。

10.3.2 去掉入口点函数

下面我们来看一下本例中会出现一些什么令人奇怪的现象。EXE 无法输出函数。现在我们的进程中服务器依赖如下的一些输出函数：

```
DllCanUnloadNow
DllRegisterServer
DllUnregisterServer
DllGetClassObject
```

因此需要找出这些函数的替换物。替换 DllCanUnloadNow 是很简单的。不同于 DLL 的是，EXE 不是被动的，它可以对自己的生命期进行控制，如它可以监控加锁计数值，并在此值为零时将自己从内存中卸载掉。因此 EXE 不需要实现 DllCanUnloadNow。这样一来我们可以将此函数去掉。

下面的两个函数 DllRegisterServer 和 DllUnregisterServer 替换起来也是很容易的。通过给 EXE 文件提供相应的命令行参数，如 RegServer 和 UnRegServer，它们将可以完成自登记。本地服务器所要做的，就是在接收到相应的命令行参数时调用 CFactory::RegisterAll 或 CFactory::UnregisterAll。（本地服务器将把相应的 EXE 文件的位置登记在关键字 LOcalServer32 下而不是 InprocServer32 下。）

这样一来就剩下 DllGetClassObject 了，替换此函数将比 DLL 输出的其他函数稍微麻烦一些。

10.3.3 类厂的启动

回想一下第 7 章中的内容，我们知道 CoCreateInstance 将调用 CoGetClassObject，而后者将调用 DllGetClassObject。DllGetClassObject 将返回一个 IClassFactory 指针，此指针将被用于创建相应的组件。由于 EXE 不输出 DllGetClassObject，因此需要给 CoGetClassObject 提供另外一种方法以获取一个 IClassFactory 指针。

COM 解决这个问题的方法是维护一个关于被登记的类厂的内部表格。当客户用合适的参数调用 CoGetClassObject 时，COM 将首先检查此关于类厂的私有表格，以得到与客户请求的 CLSID 相应的类厂。若相应的类厂不在此表格中，COM 将在注册表中进行查找并启动相应的 EXE。此 EXE 将完成相应类厂的登记，以便 COM 能够找到它们。EXE 可以调用 COM 函数 CoRegisterClassObject 来完成类厂的登记。并且此时 EXE 必须登记它所支持的所有类厂。对于 CFactory 我们可以给它加上一个新的静态成员函数 StartFactories，它将为 CFactoryData 结构中所列的每一个组件调用 CoRegisterClassObject。此函数的代码如

下所列。

```
BOOL CFactory::StartFactories()
{
    CFactoryData * pStart = &g_FactoryDataArray[0];
    const CFactoryData * pEnd =
        g_FactoryDataArray[g_cFactoryDataEntries - 1];

    for(CFactoryData * pData = pStart; pData <= pEnd; pData++)
    {
        // Initialize the class factory pointer and cookie.
        pData->m_pIClassFactory = NULL;
        pData->m_dwRegister = NULL;

        // Create the class factory for this component.
        IClassFactory * pIFactory = new CFactory(pData);

        // Register the class factory.
        DWORD dwRegister;
        HRESULT hr = ::CoRegisterClassObject(
                *(pData->m_pCLSID),
                static_cast<IUnknown *>(pIFactory),
                CLSCTX_LOCAL_SERVER,
                REGCLS_MULTIPLEUSE,
                &dwRegister);
        if (FAILED(hr))
        {
            pIFactory->Release();
            return FALSE;
        }

        // Set the data.
        pData->m_pIClassFactory = pIFactory;
        pData->m_dwRegister = dwRegister;
    }
    return TRUE;
}
```

上面的代码中,用到了给 CFactoryData 新增加的两个成员变量。其中变量 m_pIClassFactory 保存的是与 m_pCLSID 相应的类 ID 的运行时类厂。而 m_dwRegister 保存的则是此类厂的魔饼(magic cookie)[①]。

① 某些人说"cookie"并不是一个正式的工业术语,而只是 Microsoft 人所用的一个术语。许多 Web 浏览器将在用户的硬盘中保存 cookie 文件。但不管怎样,在 Microsoft,cookie 这个术语通常是被用来指标示某个对象的数据结构。客户可以向服务器请求某个资源。服务器将分配此资源并给客户传递一个 cookie。客户可以用它来标识相应的资源。就客户而言,cookie 实际上是一个随机数。此数只对服务器才有意义。

上面我们看到,类厂的登记实际上是一个比较简单的过程:只需建立相应的类厂并将其接口指针传给 CoRegisterClassObject 即可。CoRegisterClassObject 的大多数参数的含义都可以从前面的代码明显地看出来。第一个参数是被登记的类的 CLSID,其后是一个指向其类厂的指针。最后一个指针可以传回一个魔饼。此魔饼将在用 CoRevokeClassObject 来取消相应类厂的登记时用到。第三个和第四个参数都是一些 DWORD 标志,它们可以对 CoRegisterClassObject 的行为进行控制。

一、CoRegisterClassObject 标志

CoRegisterClassObject 的第二个和第三个参数是一起使用的,并且其中一个参数的含义是依赖于另外一个的。这使得这两个参数的含义比较容易混淆。

第四个参数表示 EXE 的单个实例能否支持一个组件的多个实例。为了想清楚这个问题,可将 EXE 组件服务器同 SDI(单文档界面)相比较。对于 SDI 应用程序,为了装载多个文档,需要运行此应用程序的多个实例。但一个 MDI 应用程序却可以装载多个文档。若一个 EXE 组件服务器同一个 SDI 应用程序类似,即它只能提供单个组件,那么必须使用 REGCLS _ SINGLEUSER 和 CLSCTX _ LOCAL _ SERVER。

若 EXE 组件服务器可像 MDI 应用程序能够支持多个文档那样支持组件的多个实例,则可以使用 REGCLS _ MULTI _ SEPARATE:

```
hr = ::CoRegisterClassObject(clsid, pIUnknown,
        CLSCTX _ LOCAL _ SERVER,
        REGCLS _ MULTI _ SEPARATE,
        &dwRegister);
```

这将造成一个非常有趣的现象。假定有一个登记了若干个组件的 EXE,并且此 EXE 需要使用它登记的某一个组件。在用上面的语句登记类厂时,系统将装载 EXE 的另外一个实例以提供它自己的组件。可以想到,在大多数情况下,这样做是极为低效的。为将一个 EXE 服务器作为它自己的进程中服务器登记,可以将 CLSCTX _ INPROC _ SERVER 标记同 CLSCTX _ LOCAL _ SERVER 组合起来使用,如下所示:

```
hr = ::CoRegisterClassObject(clsid, pIUnknown,
        CLSCTX _ LOCAL _ SERVER | CLSCTX _ INPROC _ SERVER,
        REGCLS _ MULTI _ SEPARATE,
        &dwRegister);
```

这样一来,EXE 服务器将可以给它自己提供进程中组件。由于在大多数情况下经常需要这么干,因此,COM 提供了另外一个标志 REGCLS _ MULTIPLEUSE。此标志可以在出现了 CLSCTX _ LOCAL _ SERVER 时自动设置 CLSCTX _ INPROC _ SERVER。因此下面的代码同前面的调用是相等同的:

```
hr = ::CoRegisterClassObject(clsid, pIUnknown,
```

```
                CLSCTX_LOCAL_SERVER,
                REGCLS_MULTIPLEUSE,
                &dwRegister);
```

通过修改示例应用程序,将可以看到 REGCLS_MULTIPLEUSE 和 REGCLS_MULTI_SEPARATE 之间的差别。为此可首先用如下的命令取消进程中服务器的登记:

```
regsrv32 /u server.dll
```

这将保证可用的服务器只是本地服务器。然后可运行客户程序并选择第二个选项以运行本地服务器。本地服务器此时将能够正常运行。注意在 CMPNT1.CPP 和 CMPNT2.CPP 的 Init 函数中我们使用了 CLSCTX_INPROC_SERVER 来创建内部组件,但前面我们已经取消了进程中服务器的登记。因此,EXE 将给它自己提供这些组件的进程中版本。

然后我们可以将 CFactory::StartFactories 中的 REGCLS_MULTIPLEUSER 修改为 REGCLS_MULTI_SEPARATE。在重新编译、链接客户和服务器之后,可运行客户并选择第二个选项。此时 create 调用将会失败,因为没有进程中服务器来满足内部组件的创建过程,而 REGCLS_MULTI_SEPARATE 则告诉 COM 库不要自己来满足对进程中组件的请求。

二、类厂的释放

当服务器被关闭时,它必须从内部表格中删除相应的类厂。为此可以使用 COM 库函数 CoRevokeClassObject。CFactory 的成员函数 StopFactories 将为 EXE 所支持的所有类厂调用 CoRevokeClassObject。

```
void CFactory::StopFactories()
{
    CFactoryData* pStart = &g_FactoryDataArray[0];
    const CFactoryData* pEnd =
      &g_FactoryDataArray[g_cFactoryDataEntries - 1];

    for (CFactoryData* pData = pStart; pData <= pEnd; pData++)
    {
    // Get the magic cookie and stop the factory from running.
    DWORD dwRegister = pData->m_dwRegister;
    if (dwRegister != 0)
    {
      ::CoRevokeClassObject(dwRegister);
    }
    // Release the class factory.
    IClassFactory* pIFactory = pData->m_pIClassFactory;
    if (pIFactory != NULL)
    {
      pIFactory->Release();
```

```
    \
  |
}
```

注意在上面的代码中，我们将从 CoRegisterClassObject 得到的魔饼传给了 CoRevoke-ClassObject。

10.3.4 对 LockServer 的修改

进程中服务器将输出函数 DllCanUnloadNow。COM 库将调用此函数以决定能否从内存中将服务器卸载掉。此函数是通过静态函数 CFactory::CoUnloadNow 实现的。CFactory::CoUnloadNow 将检查静态变量 CUnknown::s _ ActiveComponents 的值，决定是否可以将服务器从内存中卸载掉。在建立一个新的组件时，静态变量 CUnknown::s _ ActiveComponents 的值将被相应地增大。但是，在第 7 章中我们已经讨论过，在创建类厂时我们将不增大 s _ ActiveComponents的值。因此，一个服务器可以拥有打开的类厂，但它将仍然能够被关闭掉。

至此，为什么我们不给活动组件中的类厂进行计数应该就已经很清楚了。本地服务器所完成的第一件事情就是创建它所用的所有类厂，而最后做的就是释放这些类厂。若服务器在它被卸载之前需要等待所有这些类厂被释放，那么此等待的时间将是相当长的，这是因为服务器在卸载时需要亲自将它们卸载掉。因此，当客户在创建组件时，若它们需要服务器确实是在内存中，那么可以使用 IClassFactory::LockServer 来保证这一点。

对于本地服务器，需要对 LockServer 进行一些细微的修改。其原因在于：DLL 不能控制它们的生命期，它们的装载和卸载都是由另外的 EXE 文件完成的。但 EXE 则不同，它们本身可以起到控制作用，并且可以装卸或卸载它们自己。正是由于 EXE 的卸载是由它自己完成的，因此需要对 LockServer 进行修改以便当加锁计数值变成零时退出相应的 EXE。为此，我们可以在 CFactory 中加上一个新的成员函数 CloseExe，它将给应用程序的消息循环发送一条 WM _ QUIT 消息。如下所示：

```
#ifdef _OUTPROC_SERVER_
static void CFactory::CloseExe()
{
    if (CanUnloadNow() == S_OK)
    {
      ::PostThreadMessage(s_dwThreadID, WM_QUIT, 0, 0);
    }
}
#else
static void CloseExe() {/* Empty */}
#endif
```

注意，上述函数在进程中服务器中将什么也不做。这样我们只需从 LockServer 中直

接调用 CloseExe 即可,从而可以实现非常高的效率。

```
RESULT __stdcall CFactory::LockServer(BOOL bLock)
{
    if (bLock)
    {
      ::InterlockedIncrement(&s_cServerLocks);
    }
    else
    {
      ::InterlockedDecrement(&s_cServerLocks);
    }
    // If this is an out-of-proc server, check to see
    // whether we should shut down.
    CloseExe();

    return S_OK;
}
```

在组件的析构函数中,也需要调用 CloseExe,这是因为此时组件也可以决定是否可将其从内存中卸载掉。为此可以对 CUnknown 的析构函数加以修改以完成这项工作:

```
CUnknown::~CUnknown()
{
    ::InterlockedDecrement(&s_cActiveComponents);

    // If this is an inactive EXE server, shut it down.
    CFactory::CloseExe();
}
```

一、消息循环

C 和 C++ 程序的标准入口点为 main。程序的执行将从 main 函数开始。当 main 函数退出时,程序也就终止了。与此类似,Windows 程序也有一个相应的 WinMain 函数。因此,为了不致使 EXE 退出,需要在其中加上一个循环。由于我们构造的组件将运行于 Windows 环境下,因此,可以加上一个 Windows 的消息循环。这个消息循环同所有的 Windows 程序所用的消息循环都是相同的。

我们可以将此消息循环放在一个名为 OUTPROC.CPP 的文件中。此文件只应在建立进程外服务器时才被编译和链接。

二、用户的计数

前面我们提到在运行客户之前需要先运行服务器。在客户程序终止之后,服务器将仍然保留在内存中。服务器的用户也是一种类型的客户,因此,也需要有它们的加锁计数

值。所有当用户创建组件时,也需要将 CFactory::s _cServerLocks 计数值增大。这样,只要有用户在使用服务器,它将一直保存在内存中。

那么我们如何决定用户何时启动了服务器而不是 COM 库呢? 当 CoGetClassObject 装载本地服务器的 EXE 时,它将在命令行上加上一个 Embedding 参数。EXE 将检查命令行中的 Embedding 参数。若没有此参数,它将增大 s _cServerLocks 值并创建一个窗口以便同用户进行交互。

当用户将服务器关闭时,客户可能仍然在使用服务器。因此,服务器应在用户退出时将其用户界面元素清除掉,但它不应该退出,这样才能继续为其客户提供相应的服务。因此当服务器收到一个 WM _DESTROY 消息时,它只应在 CanUnloadNow 返回 S _OK 时才发送 WM _QUIT 消息。

10.4 远程访问能力

本章实现的本地服务器最让人感兴趣的地方可能是:它也是一个远程服务器。也就是说我们不需要对 CLIENT.EXE 或 SERVER.EXE 进行任何修改,即可使它们通过一个网络而一起工作。当然这至少需要有两套系统,分别运行 Windows NT4.0 或安装有 Distributed COM for Windows 95(DCOM,分布式 COM)的 Windows 95。并且这两套系统还应通过某种类型的网络连接起来。为让客户使用远程服务器,需要用到 DCOM 配置工具 DCOMCNFG.EXE。Windows NT 包含这一工具。这个工具可以让用户修改系统中安装的应用程序的各种参数,如是本地运行还是远程运行。

表 10-2 列出了让 SERVER.EXE 远程运行的各个步骤。

表 10-2 从远程服务器运行 SERVER.EXE 的步骤

动作	本地系统	远程系统
建立 CLIENT.EXE、SERVER.EXE 以及 PROXY.DLL。这可以使用 nmake -f makefile。这样当已经建立好它们时就不需要重新建立了。(作者是在 Windows 95 系统上建立好它们的,然后将这些程序复制到 Windows NT 系统上。)	√	
将 CLIENT.EXE、SERVER.EXE 以及 PROXY.DLL 复制到远地系统中。		√
使用命令 server /RegServer 登记本地服务器。	√	√
使用命令 regsvr32 Proxy.dll 登记代理。	√	√
运行 CLIENT.EXE,并选择本地服务器选项。这主要是为了保证程序在两个系统中都能够正常工作。	√	√

续表

动作	本地系统	远程系统
运行 DCOMCNFG.EXE。选择建立的组件(如 Inside COM Chapter 10 Example Component 1)并单击 Properties。然后选择 Location 选项卡。取消对 Run Application On This Computer 的选择,并选择 Run Application On The Following Computer 选项,然后输入将要运行 SERVER.EXE 的远程机器名称。	√	
单击 Identity 选项卡,然后选择 Interactive User 单选钮。	√	
根据自己的访问权限,对 Security 选项卡上的选项进行设置。	√	√
运行 SERVER.EXE 以观察其输出。		√
运行 CLIENT.EXE,选择第二个选项以选择本地服务器。	√	
此时在服务器的窗口中应该可以看到相应的输出。		√
输出也应该出现在同 CLIENT.EXE 相关联的控制台窗口中。	√	

运行一下 DCOMCNFG.EXE 这个工具即可将本地服务器变成一个远程服务器。这确实是一件令人兴奋的事情。但现在有一个问题是:这一点是如何实现的呢?

10.4.1 DCOMCNFG.EXE 所完成的工作

若在本地机器上运行 DCOMCNFG.EXE 之后运行 REGEDIT.EXE,就会发现注册表将会有一些变化。为此可以看一下注册表中的如下关键字:

```
HKEY_CLASSES_ROOT\
    CLSID\
        {0C092C29-882C-11CF-A6BB-0080C7B2D682}
```

此时我们将会看到除了组件的易记名称之外,还将出现一个新的命名值(不是子关键字)AppID。我们知道 CLSID 是标识一个组件的,而其注册表项则包含此组件的一些信息。LocalServer32 关键字包含的是实现组件的应用程序的路径名,否则一个 CLSID 将同此应用程序没有任何关系。但是 DCOM 需要将某些信息同包含有组件的应用程序关联起来,因此 DCOM 创建了 AppID。

AppID 的值同 CLSID 值一样,也是一个 GUID。关于 AppID 的信息被保存在注册表中的 AppID 分支中。这一点同 CLSID 也是类似的。为访问注册表中 SERVER.EXE 的 AppID 的信息,可以查找如下的关键字:

```
HKEY_CLASSES_ROOT\
    AppID\
{0C092C29-882C-11CF-A6BB-0080C7B2D682}
```

AppID 至少有三个值。缺省值为一个易记的名称。其他命名值分别是 RemoteServer-

Name(其中包含的是应用程序所在的服务器的名称)以及 RunAs(它将告诉 DCO 如何运行相应的应用程序)。此注册表的结构可以用图 10-6 表示。

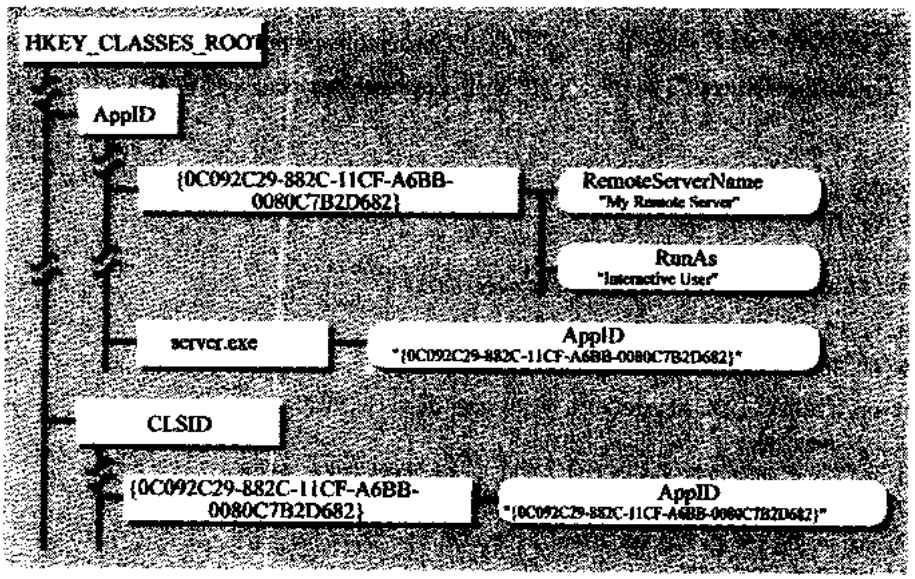


图 10-6 AppID 项的注册表结构

另外应用程序的名称将直接保存在 AppID 关键字下。因此在注册表中可以看到如下的一项:

```
HKEY_CLASSES_ROOT\
    AppID\
        server.exe
```

该项只有一个命名的值,即相应的 AppID。

10.4.2 工作机理

如果没有相应的代码来读取注册表中的项,那么向注册表中加上再多的新项也没有什么意义。为此 DCOM 对 COM 库进行了增强,其中包括 CoGetClassObject 的实现。增强后的 CoGetClassObject 不但功能更为强大,而且更易引起混淆。例如它可以按照多种不同的方式工作。通常 CoGetClassObject 将接受一个 CLSID 并用合适的上下文参数启动相应的组件服务器。若此上下文为 CLSCTX_LOCAL_SERVER,CoGetClassObject 将在注册表中查找相应的组件以得知它是否有一个相应的 AppID。若有的话,此函数将在注册表中查找 RemoteServerName。在找到一个服务器名称之后,CoGetClassObject 将试着以远程方式运行服务器。这样就会出现我们前面所描述的情况。

10.4.3 其他 DCOM 信息

除了可以对注册表进行处理以将一个本地服务器变成一个远程服务器外，还可以在程序中指定访问某个远程服务器。这需要用 CoCreateInstanceEx 来替换 CoCreateInstance 或修改对于 CoGetClassObject 的调用。下面是用 CoCreateInstanceEx 来创建一个远程组件的例子：

```
// Create a structure to hold server information.
COSERVERINFO ServerInfo ;
// Initialize structure to 0.
memset(&ServerInfo, 0, sizeof(ServerInfo)) ;

// Set the name of the remote server.
ServerInfo.pwszName = L"MyRemoteServer" ;

// Set up a MULTI_QI structure with the desired interfaces.
MULTI_QI mqi[3] ;
mqi[0].pIID = IID_IX ; // [in] IID of desired interface
mqi[0].pItf = NULL ; // [out] Pointer to interface
mqi[0].hr = S_OK ; // [out] Result of QI for pointer
mqi[1].pIID = IID_IY ;
mqi[1].pItf = NULL ;
mqi[1].hr = S_OK ;
mqi[2].pIID = IID_IZ ;
mqi[2].pItf = NULL ;
mqi[2].hr = S_OK ;

HRESULT hr = CoCreateInstanceEx(CLSID_Component1,
                    NULL,
                    CLSCTX_REMOTE_SERVER,
                    &ServerInfo,
                    3, // Number of interfaces
                    &mqi) ;
```

可以看到 CoCreateInstanceEx 和 CoCreateInstance 的第一个差别是 CoCreateInstanceEx 将接受一个保存有远程服务器名称的 COSERVERINFO 结构。但 CoCreateInstanceEx 最令人感兴趣之处还是 MULTI_QI 结构。

一、MULTI_QI

对于进程中组件，QueryInterface 调用是非常快的。对于本地服务器中的组件，QueryInterface 调用也是相当快的。但在跨越网络时，调用函数的开销将会显著地增大。当重复地进行此类函数调用时，应用程序会停下来并不是什么奇怪的事情。对于 QueryInterface 调用也是如此。因此为减少 QueryInterface 调用造成的影响，DCOM 建立了一个名为 MULTI_QI 的新的结构。有了这个结构，我们就能同时查询多个接口，从而使开销显著地

降低。

在上面的例子中，我们同时查询了 IX、IY 和 IZ 结构。在 CoCreateInstanceEx 退出时，若它能够查询到 MULTI _ QI 结构中的所有接口，它将返回 S _ OK。若任何接口都无法查询到，它将返回 E _ NOINTERFACE；若只查询到了某些接口，则返回值将是 CO _ S _ NOTALLINTERFACES。

各接口的状态信息将被保存在 MULTI _ QI 结构的 hr 域中。而接口指针则被保存在 pIft 域中。

在查询多个接口时，CoCreateInstanceEx 将在创建组件之后查询其 IMultiQI。IMultiQI 的声明如下：

```
interface IMultiQI : IUnknown
{
    virtual HRESULT __stdcall QueryMultipleInterfaces
      (ULONG interfaces,
      MULTI _QI* pMQIs);
};
```

开发人员并不需要在自己的组件中实现 IMultiQI。此接口的实现将是由组件的远程代理“免费”提供的。

二、在 Windows 95 上 CoCreateInstance 的失败处理

若定义预处理符号：

```
_WIN32 _DCOM
```

或

```
_WIN32 _WINNT > = 0X0400
```

那么在 CLSCTX _ SERVER 和 CLSCTX _ ALL 中将会包含 CLSCTX _ REMOTE _ SERVER。这样，对于没有安装 DCOM 的 Windows 95 系统，CoCreateInstance 将会失败。因此，在为 Windows 95 系统或 Windows NT 3.51 系统开发组件时，一定不要定义这两个符号。

三、如何决定 DCOM 是否可用

为决定 DCOM 服务是否可用，可以先检查一下 OLE32.DLL 是否支持组件能够供所有的线程访问。为此，在静态链接到 OLE32.DLL 的情况下，可以使用如下的代码：

```
if (GetProcAddress(GetModuleHandle("OLE32"),
        "CoInitializeEx") != NULL)
{
  // Free threading is available.
}
```

在动态链接到 OLE32.DLL 的情况下，可以使用如下的代码：

```
hmodOLE32 = LoadLibrary("OLE32.DLL");
if (GetProcAddress(hmodOLE32, "CoInitializeEx") ! = NULL)
{
    // Free threading is available.
}
```

之后即可决定 DCOM 是否可用了：

```
LONG lResult = RegOpenKeyEx(HKEY _ LOCAL _ MACHINE,
            "SOFTWARE \ Microsoft \ Ole",
            0,
            KEY _ ALL _ ACCESS,
            &hKey) ;
assert(lResult == ERROR _ SUCCESS) ;
char rgch[2] ;
DWORD cb = sizeof(rgch) ;
lResult = RegQueryValueEx(hKey,
            TEXT("EnableDCOM"),
            0, NULL, rgch, &cb) ;
assert(lResult == ERROR _ SUCCESS) ;
lResult = RegCloseKey(hKey) ;
assert(lResult == ERROR _ SUCCESS) ;
if (rgch[0] == 'y' || rgch[0] == 'Y')
{
  // DCOM is available.
}
```

10.5 本章小结

跨越进程边界确实是一件非常有趣的事情，特别是在有如此众多的工具（如 MIDL 编译）来简化这一过程的情况下。通过用 IDL 编写接口的描述，可以让 MIDL 编译器生成代理和残根 DLL 所需的代码，以对跨越进程边界的接口参数进行调用。更令人惊奇的可能是只需对注册表中的几个项稍加修改，即可将本地服务器变成为远程服务器。

跨越边界确实是非常有趣的。作者仍然记得那些在东柏林郊区的一个公园中度过的美好时光。在鲜花与绿树从中，孩子们与宠物园中的动物嬉戏得是那么的开心。确实是令人心动的。

第 11 章　调度接口与自动化

人们经常说:剥猫皮可以用 N 种不同的方法。不过作者从来没有试图剥过猫皮,并且也从来没有发现这样做有什么用,因此作者更喜欢说:抓猫可以用 N 种不同的方法。相信猫们一定会更喜欢这句话。作者的一个来自佐治亚 La Grange 的朋友则有另外一种不同的说法:像黄鼠狼那样闻东西有 N 种不同的方法。在他母亲看来,他似乎已经找到了其中的大多数方法了。如果读者愿意的话,一定还能找出上面那句话的其他版本。

本章我们要介绍客户与组件之间通信的各种不同方法。在前面的例子中,COM 客户与组件之间的通信都是通过 COM 接口直接完成的。在这一章中我们将介绍客户控制组件的另外一种方法,即自动化(以前称作是 OLE 自动化)。许多应用程序如 Microsoft Word 和 Microsoft Excel 以及一些解释性的语言如 Visual Basic 和 Java 都使用了这一方法。

自动化使得用解释性语言和宏语言访问 COM 组件更为容易,同时用这些语言编写组件也将更为容易。自动化关注的是运行时的类型检查,这一点是以速度的牺牲和编译时的类型检查为代价的。虽然自动化对于宏语言的开发来说使用起来更为容易,对于 C++ 开发而言,则需要做更多的工作。自动化将在许多不同的方面用开发编写的代码来替换编译器生成的代码。

自动化不是独立于 COM 的,而是建立在 COM 基础上的。一个自动化服务器实际上就是一个实现了 IDispatch 接口的 COM 组件。而一个自动化控制器则是一个通过 IDispatch 接口同自动化服务器进行通信的 COM 客户。自动化控制器不会直接调用自动化服务器实现的那些函数,而是通过 IDispatch 接口中的成员函数实现对服务器中函数的间接调用。

IDispatch 接口同自动化的其他所有方面一样,是作为 Visual Basic 的一部分开发出来的,以专门用于将一些应用程序如 Microsoft Word 和 Microsoft Excel 自动化。Microsoft Office 的开发语言 Visual Basic for Applications 实际上就是从 Visual Basic 发展来的。Visual Basic for Applications 的一个子集 Visual Basic Scripting Edition(VBScript)可用于将 Web 页面上 ActiveX 控件自动化。Microsoft Developer Studio 5.0 也将 VBScript 作为其宏语言。

通过 COM 接口提供的任何服务都可以通过 IDispatch 接口来提供。这意味着 IDispatch 和自动化同 COM 一样也是一个相当大的论题(或许可能更大一些)。由于本书的中心问题是介绍 COM 的使用,因此我们将只讨论自动化的某些方面。但这些方面的问题也是相当多的:IDispatch、调度接口、双重接口、类型库、IDL、VARIANT、BSTR 等等。这些问题对于 C++ 自动化程序员将是最为重要的。

下面我们将从头开始讨论这些问题，看一看通过 IDispatch 进行通信与通过 COM 接口进行通信有一些什么差别。

11.1 一种新的通信方式

那么是什么使 IDispatch 成为如此令人感兴趣的一个接口呢？这主要是由于 IDispatch 为客户和组件提供了另外一种通信方式。有了 IDispatch 之后，COM 组件就可以通过一个标准的接口提供它所支持的服务，而无需提供多个特定于服务的接口。

在详细讨论 IDispatch 之前，我们先来看一下它是如何支持这么多功能的。

11.1.1 旧的通信方式

首先简要回顾一下目前客户控制组件的方法。读者对这个问题可能已经厌烦了。但在此还是需要再重复一下。客户与组件之间的通信是通过接口完成的。接口拥有一个由函数指针构成的数组。客户如何知道它想调用的函数包含在函数指针数组的哪一项中呢？为此，客户代码需要包含一个以抽象基类形式描述接口的头文件。编译器将读取此头文件，然后为抽象基类中的每一个成员函数分配一个索引。此索引实际上就是相应函数的指针在函数指针数组中的索引。这样当编译器读到下面的代码时：

```
pIX->FxStringOut(msg);
```

将把它解释成：

```
(*(pIX->pvtbl[IndexOfFxStringOut]))(pIX, msg);
```

其中 pvtbl 为指向相应类的 vtbl 的指针，而 IndexOfFxStringOut 则为 FxStringOut 的指针在函数指针表中的索引。当然这些处理都是在不知不觉中完成的。在大多数情况下，程序员无需关心这些问题。

但在为应用程序开发一种宏语言时，则需要关心这些问题。若此宏语言能够使用 COM 组件，那么它的功能将更为强大。但是宏语言又如何能够获取函数在 vtbl 中的指针的索引以便能够调用它们呢？此时一种作法是编写一个 C++ 分析程序以对相应接口的头文件进行分析。

当在宏语言中调用 COM 组件中的一个函数时，可以使用三种类型的信息：实现被调用函数的组件的 ProgID、函数的名称以及传给函数的参数。我们需要的是当宏运行时系统能够提供一种通过函数的名称来执行函数的简单方法。这种方法就是 IDispatch 接口所提供的。

11.1.2 IDispatch 接口

简单地讲，IDispatch 将接收一个函数的名称并执行它。关于 IDispatch 接口的 IDL 描

述,可以在文件 OAIDL.IDL 文件中找到(此文件在编译器的 \ Include 目录下)。如下所示:

```
interface IDispatch : IUnknown
{
  HRESULT GetTypeInfoCount([out] UINT* pctinfo);
  HRESULT GetTypeInfo([in] UINT iTInfo,
        [in] LCID lcid,
        [out] ITypeInfo** ppTInfo);
  HRESULT GetIDsOfNames(
        [in] const IID& riid,
        [in, size_is(cNames)] LPOLESTR* rgszNames,
        [in] UINT cNames,
        [in] LCID lcid,
        [out, size_is(cNames)] DISPID* rgDispId);
  HRESULT Invoke([in] DISPID dispIdMember,
        [in] const IID& riid,
        [in] LCID lcid,
        [in] WORD wFlags,
        [in, out] DISPPARAMS* pDispParams,
        [out] VARIANT* pVarResult,
        [out] EXCEPINFO* pExcepInfo,
        [out] UINT* puArgErr);
};
```

IDispatch 中最令人感兴趣的两个函数是 GetIDsOfNames 和 Invoke。GetIDsOfNames 将读取一个函数的名称并返回其调度 ID,或称 DISPID。DISPID 并不是一个 GUID,而只是一个长整数,它标识的是一个函数。对于 IDispatch 的某一个特定的实现,DISPID 是唯一的。IDispatch 的每一个实现都有其自己的 IID(有些人称之为 DIID,即调度接口的 ID)。

为执行某个函数,自动化控制程序将把 DISPID 传给 Invoke 成员函数。Invoke 可以将 DISPID 作为函数指针数组的索引,这一点同常规 COM 接口是相似的。但自动化服务并不需要按此种方式实现 Invoke。一个简单的自动化服务器可以根据 DISPID 用一个 case 语句执行不同的代码。这一点同 MFC 广为流行之前的窗口过程的实现方法是相似的。

IDispatch::Invoke 在其他许多方面同窗口过程也是类似的。将一个窗口过程同一个窗口关联起来的方法同将一个自动化服务器同一个 IDispatch::Invoke 函数关联起来的方法是相同的。Microsoft Windows 将给窗口过程发送各种各样的消息,而自动化控制程序将给 IDispatch::Invoke 传递相应的 DISPID。窗口过程的行为将由它接收到的消息决定,而 Invoke 的行为则是由它接收到的 DISPID 决定。

IDispatch::Invoke 的工作方式同 vtbl 幕后的工作方式也是类似的。Invoke 将实现一组按索引来访问的函数,而 vtbl 也正是一个按索引来访问的函数指针数组。与 vtbl 通过

C++编译器自动工作相反，为了实现 Invoke 的功能，开发人员需要付出大量的工作。但是 C++ vtbl 是静态的，而且编译器只是在运行时才能起作用。若 C++ 程序员想在运行时从头开始构造一个 vtbl，他可以完全自行决定如何构造。另一方面，我们可以比较方便地给出 Invoke 的一个一般性的实现，然后对之动态地定制以提供不同的服务。

一、调度接口

IDispatch::Invoke 的实现与 vtbl 还有另外一个方面是相似的：它们都可以定义接口。IDispatch::Invoke 的一个实现所实现的函数集被称作是一个调度接口。而 COM 接口是一个指向一个函数指针数组的指针，此数组的前三个元素分别是 QueryInterface、AddRef 以及 Release。关于接口还有一个更一般性的定义：接口是一组函数和变量的集合，程序的两个不同部分可通过它进行通信。而 IDispatch::Invoke 的实现也是由一组函数构成的，自动化控制器和自动化服务器可以通过它进行交互。因此，不难看出 Invoke 实现的函数也将构成一个接口，虽然此接口不是一个 COM 接口。

图 11-1 给出了调度接口的一个图示。图的左边是一个传统的 COM 接口 IDispatch，它是以 vtbl 的方式实现的。在图的右边是一个调度接口。IDispatch::Invoke 能识别的 DISPID 对于调度接口具有决定性的意义。图中给出了 Invoke 和 GetIDsOfNames 的一种可能的实现：一个用 DISPID 作为索引的函数名称数组和函数指针数组。这只不过是一种可能的实现而已。对于大的调度接口，可以将传给 GetIDsOfNames 的名称作为一个哈希表的关键字，这样系统的效率将会更高一些。

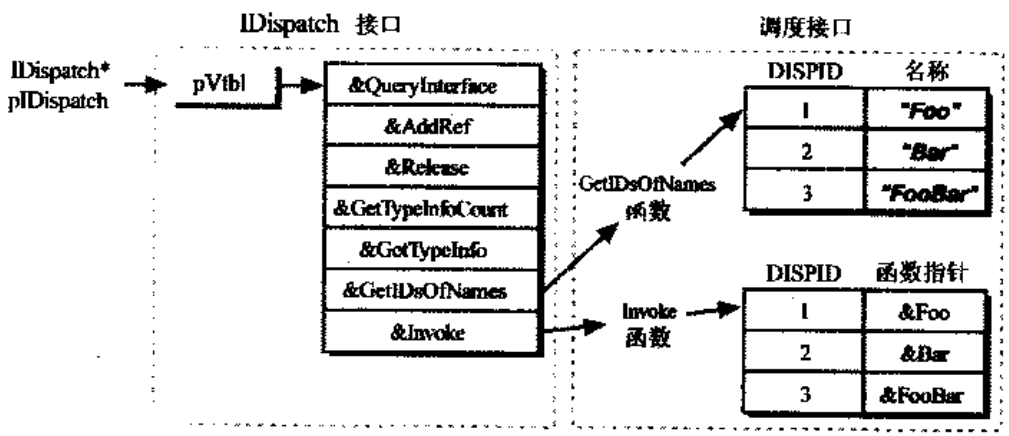


图 11-1 调度接口是由 IDispatch 实现的。它们同 COM 接口是完全不同的。图中给出了 IDispatch::Invoke 的一种可能的实现

当然用一个 COM 接口来实现 IDispatch::Invoke 也是可能的。关于这一点请参阅图 11-2。

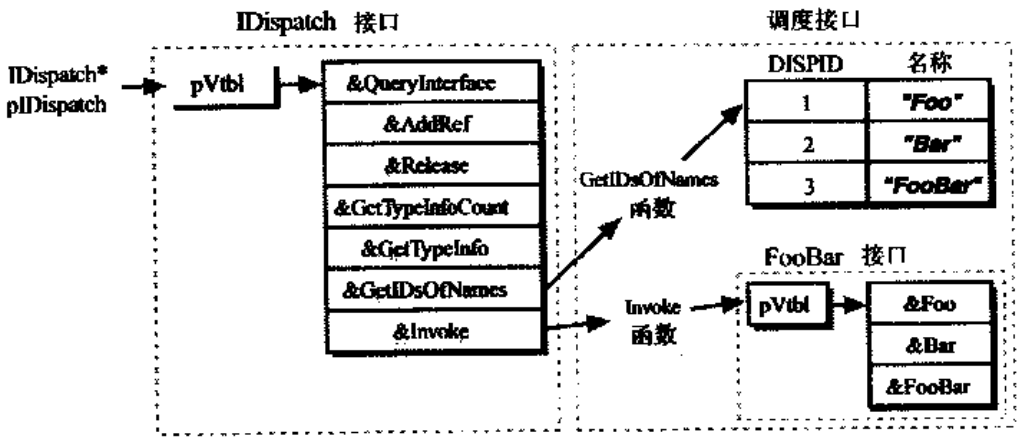


图 11-2 使用一个 COM 接口来实现 IDispatch::Invoke

二、双重接口

图 11-2 表示的不是用 COM 组件实现调度接口的唯一方法。我们也可以用图 11-3 所示的方法。此种方法是让实现 IDispatch::Invoke 的 COM 组件继承 IDispatch 而不是 IUnkown。这是实现一种被称作是双重接口这种类型的接口的方法。双重接口也是一种调度接口,它将使得通过 Invoke 能够访问的函数也能够直接通过 vtbl 而访问到。

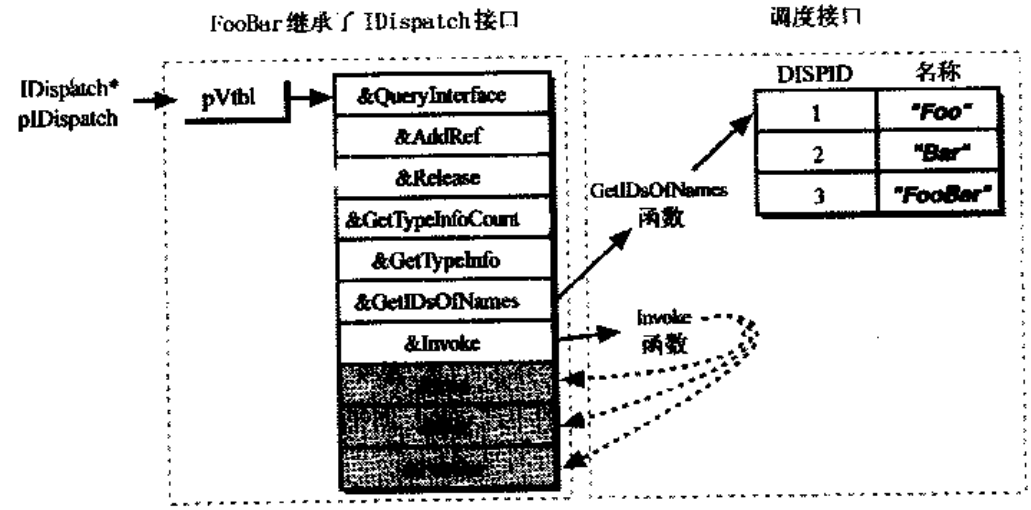


图 11-3 双重接口是一种从 IDispatch 继承的 COM 接口。此接口的成员函数可以通过 Invoke 和 vtbl 这两种方式访问

在实现调度接口时,更好的方法是使用双重接口。双重接口使 C++ 程序员能够通过 vtbl 进行函数调用。这种调用方式不但简化了 C++ 程序员的工作,而且运行起来也会更快一些。宏语言和解释性语言也可以使用实现双重接口的组件提供的服务。此时它们可

以不通过 vtbl,而通过 Invoke 方法。Visual Basic 程序既可以连接到双重接口的调度接口部分,也可以连接到其 vtbl 部分。当将一个 Visual Basic 变量定义为 Object 类型时,它将连接到调度接口上:

```
Dim doc As Object
Set doc = Application.ActiveDocument
doc.Activate
```

但是如果给变量指定某种对象类型,那么 Visual Basic 将通过 vtbl 来完成相应的函数调用:

```
Dim doc As Document
Set doc = Application.ActiveDocument
doc.Activate
```

但任何事情都不会是完美无缺的,双重接口也是。那么使用双重接口有一些什么不足呢? 对于 Visual Basic 而言,似乎确实找不出什么缺点。但对于 C++ 自动化控制程序而言,情况就不一样了,我们将可以举出若干不足,例如,此时参数的类型将受到一些限制。

在介绍双重接口和调度接口参数可以使用的那些数目有限的类型之前,我们先来看一下如何用 C++ 来调用调度接口。

11.2 IDispatch 的使用

考虑下面的 Visual Basic 程序:

```
Dim Cmpnt As Object
Set Cmpnt = CreateObject("InsideCom.Chap11.Cmpnt1")
Cmtnt.Fx
```

这段程序将建立一个 COM 组件并通过组件实现的 IDispatch 接口来调用函数 Fx。下面我们来看一下如何用 C++ 来完成这些工作。首先需要用组件的 ProgID 来创建相应的组件(此过程在第 6 章中介绍过)。这项工作可以用下面的代码来完成。

```
// Initialize the OLE Library.
HRESULT hr = OleInitialize(NULL);

// Get the CLSID for the application.
wchar_t progid[] = L"InsideCOM.Chap11";
CLSID clsid;
::CLSIDFromProgID(progid, &clsid);

// Create the component.
IDispatch* pIDispatch = NULL;
::CoCreateInstance(clsid,
```

```
        NULL, CLSCTX _ INPROC _ SERVER,
        IID _ IDispatch, (void * * )&pIDispatch) ;
```

上面我们指出 CoCreateInstance 返回一个 IDispatch 指针，这样将可以省却一个单独的 QueryInterface 调用。在得到了一个 IDispatch 指针之后，即可以得到调度函数 Fx 的 DISPID。函数 IDispatch::GetIDsOfNames 将可以把一个代表函数名称的串转换成一个 DISPID。

```
DISPID dispid ;
OLECHAR * name = L"FX" ;
pIDispatch-> GetIDsOfNames(
        IID _ NULL,                  // Must be IID _ NULL
        &name,                       // Name of function
        1,                           // Number of names
        GetUserDefaultLCID(),        // Localization info
        &dispid) ;                   // Dispatch ID
```

对于客户而言，DISPID 的作用只不过是避免将字符串来回地传送。而对于服务器而言，DISPID 所标识的将是客户待调用的函数的名称。

在得到了 Fx 的 DISPID 后，即可通过将 DISPID 传给 IDispatch::Invoke 来调用相应的函数。下面的代码给出了一个最简单的 Invoke 的可能调用。在此没有给被调用的函数传递任何有意义的参数。

```
// Prepare the arguments for Fx.
DISPPARAMS dispparamsNoArgs = {
  NULL,
  NULL,
  0,  // Zero arguments
  0   // Zero named arguments
} ;

// Simplest Invoke call.
pIDispatch-> Invoke(dispid,     // DISPID
        IID _ NULL,     // Must be IID _ NULL
        GetUserDefaultLCID(),
        DISPATCH _ METHOD,  // Method
        &dispparamsNoArgs, // Method arguments
        NULL,       // Results
        NULL,       // Exception
        NULL) ;     // Arg error
```

自动化控制程序不需要知道关于自动化服务器的任何情况。因此，它也不需要带有 Fx 定义的头文件。关于 Fx 的信息没有被静态地链接到程序中。可以将这一点同 IDispatch 接口本身进行比较。IDispatch 是一个 COM 接口，它的 IDL 描述可以在 OAIDL.IDL 中

找到。编译器将在编译时生成对 IDispatch 成员调用的代码,此代码将是静态的。但是传给 Invoke 的参数将决定所调用的函数究竟是什么。这些参数同其他函数参数一样,是动态的,可以在运行时灵活变化。

将前面的代码段变成一个可以不用任何参数调用任意调度函数的程序并不是什么困难的事情。为此只需向用户询问两个字符串,一个 ProgID,一个待调用的函数的名称,然后将这些串分别传给 CLSIDFromProgID 和 GetIDsOfNames 即可。调用 Invoke 的代码可以保持不变。

Invoke 的威力在于它可以被一致地使用。任何一个组件,只要它实现了 Invode,我们均可以用相同的代码来调用它。但是为得到此种能力也是需要付出代价的。IDispatch::Invode 的工作之一是将参数传给将要被执行的函数。而 Invoke 可以传给调度接口中的函数的参数类型的数目是有限的。关于此我们将在下面有关的章节中进行详细的讨论。下面我们先来看一下传给 IDispatch::Invoke 本身的参数。

11.2.1 Invoke 函数的参数

现在我们来详细看一下传给 Invoke 函数的参数。前面的三个解释起来是比较容易的。第一个参数是控制程序待调用函数的 DISPID;第二个参数是保留的,必须为 IID_NULL。第三个参数保存的是位置信息。关于第四个参数以后的其他各参数,下面我们将分别讨论。

一、方法和属性

COM 接口中的所有成员都是函数。COM 接口、许多 C++ 类,甚至许多 Win32 API 都使用"Set"和"Get"类函数来模拟对成员变量的访问。例如,假定 SetVisible 可以设置一个窗口的可见性属性,而 GetVisible 则可以获得这一可见性属性:

```
if (pIWindows -> GetVisible() == FALSE)
{
    pIWindows -> SetVisible(TRUE) ;
}
```

但对于 Visual Basic 来说,"Set"和"Get"类函数并不是特别理想。Visual Basic 的主要目的是使所有的事情对于开发者来说都应尽可能的简单。它支持一种名为属性的概念。属性实际上就是 Visual Basic 程序员可以当成变量对待的"Get"/"Set"函数。所以也可以不使用函数的语法而使用如下的变量语法:

```
VB Code
If Window.Visible = False Then
    Window.Visible = True
End If
```

IDL 的 propget 和 propput 属性可以将 COM 函数说明为可以将其当成一个属性对待。如下所示即为一例：

```
[
    object,
    uuid(D15B6E20-0978-11d0-A6BB-0080C7E2D682),
    pointer_default(unique),
    dual
]
interface IWindow : IDispatch
{
.
.
.
    [propput]
    HRESULT Visible([in] VARIANT_BOOL bVisible);
    [propget]
    HRESULT Visible([out, retval] VARIANT_BOOL* pbVisible);
.
.
.
}
```

在上面的代码中我们给接口定义了一个 Visible 属性。被标记为 propput 的函数将接受一作为其值的参数。而被标记为 propget 的函数则通过一个输出参数返回一个值。此时属性的名称同函数的名称是相同的。当 MIDL 生成 propget 和 propput 函数的头文件时，它将在函数名称前加上 get_ 或 put_ 前缀。因此 C++ 程序员可以使用如下的代码来调用这些函数：

```
VARIANT_BOOL vb;
get_Visible(&vb);
if (vb == VARIANT_FALSE)
{
    pub_Visible(VARIANT_TRUE);
}
```

至此，读者也许会大致明白为什么 C++ 程序员不喜欢调度接口。那些对 Visual Basic 用户来说是好的方面，对于 C++ 程序员却变成了坏事。正因如此，我们可以给 C++ 用户提供一个低级的 COM 接口，而为 Visual Basic 和 Java 用户提供一个高级的双重接口。此双重接口可以用低级的 COM 接口来实现。试图编写出好的接口是相当困难的，而要让两种类型极为不同的开发者都满意更为不易。另外顺便提一下，VARIANT_BOOL 是调度接口所用的布尔类型。VARIANT_FALSE 的值为 0，而 VARIANT_TRUE 的值则为 0xFFFF。

读者可能会问：所有这些同 IDispatch 的第四个参数有什么关系呢？答案就是我们可

以将一个名称(如 Visible)同四个不同的函数关联起来。这四个函数分别是一个常规函数、一个设置属性的函数、一个通过引用设置属性的函数以及一个获取属性的函数。所有这些具有相同名称的函数将具有同一个 DISPID,但它们却是按照完全不同的方式来加以实现的。因此,Invode 需要知道它所调用的函数是哪一种类型的。第四个参数的作用就是提供此种信息。它的值可以是如下几个值之一:

```
DISPATCH_METHOD
DISPATCH_PROPERTYGET
DISPATCH_PROPERTYPUT
DISPATCH_PROPERTYPUTREF
```

二、调度参数

IDispatch::Invoke 的第五个参数包含的是传给被调用函数的参数。这一点似乎容易引起混淆。为此可以举一个例子来说明这个问题。假定我们现在调用 Invode 以将 Visible 属性设置为 True。此时我们可以通过 Invode 函数的第五个参数将相应的参数传递给设置此属性的函数。这里将要传递的参数值为 True。

第五个参数是一个 DISPPARAMS 结构。其定义如下:

```
typedef struct tagDISPPARAMS {
    VARIANTARG* rgvarg;          // Array of arguments
    DISPID* rgdispidNamedArgs;   // DISPIDs of named args
    unsigned int cArgs;          // Number of arguments
    unsigned int cNamedArgs;     // Number of named args
}   DISPPARAMS;
```

Visual Basic 及调度接口支持一种名为命名参数的概念。命名参数是通过提供参数的名称而使程序员能够以任意的次序将参数传递给某个函数。这个概念对于 C++ 程序员来说没有任何用处。因此,在此我们也不准备详细讨论它。在本书中 rgdispidNamedArgs 将永远为 NULL,而 cNamedArgs 将永远是 0。

DISPPARAMS 结构的第一个成员 rgvarg 为一个参数数组。cArgs 成员则是数组中元素的个数。每一个参数的类型都是 VARIANTARG,这就是为什么在自动化控制程序与其服务器之间可以传递的参数类型其数目是有限的原因。只有那些能够被放到 VARIANTARG 结构中的类型才可以通过调度接口进行传递。对于双重接口情形也将是如此。因为为了使通过 vtbl 能够访问的函数与通过 Invoke 能够访问的函数相匹配,双重接口也将被限制为只能使用那些可以用 VARIANTARG 结构表示的类型。

VARIANTARG 同 VARIANT 是相同的。关于 VARIANT 的定义在 OAIDL.IDL 中可以找到。如下所示:

```
typedefstruct tagVARIANT {
```

```
union {
  struct _tagVARIANT {
    VARTYPE vt;
    WORD  wReserved1 ;
    WORD  wReserved2 ;
    WORD  wReserved3 ;
    union {
      LONG     lVal ;     // VT_I4
      BYTE     bVal ;     // VT_UI1
      SHORT    iVal ;     // VT_I2
      FLOAT    fltVal ;   // VT_R4
      DOUBLE    dblVal ;  // VT_R8
      VARIANT_BOOL boolVal ;     // VT_BOOL
      SCODE     scode ;     // VT_ERROR
      CY     cyVal ;     // VT_CY
      DATE     date ;     // VT_DATE
      BSTR     bstrVal ;     // VT_BSTR
      IUnknown *     punkVal ;     // VT_UNKNOWN
      IDispatch *     pdispVal ;     // VT_DISPATCH
      SAFEARRAY *     parray ;     // VT_ARRAY
      BYTE *     pbVal ;     // VT_BYREF|VT_UI1
      SHORT *     piVal ;     // VT_BYREF|VT_I2
      LONG *     plVal ;     // VT_BYREF|VT_I4
      FLOAT *     pfltVal ;     // VT_BYREF|VT_R4
      DOUBLE *     pdblVal ;     // VT_BYREF|VT_R8
      VARIANT_BOOL * pboolVal ;     // VT_BYREF|VT_BOOL
      SCODE *     pscode ;     // VT_BYREF|VT_ERROR
      CY *     pcyVal ;     // VT_BYREF|VT_CY
      DATE *     pdate ;     // VT_BYREF|VT_DATE
      BSTR *     pbstrVal ;     // VT_BYREF|VT_BSTR
      IUnknown * *     ppunkVal ;     // VT_BYREF|VT_UNKNOWN
      IDispatch * *     ppdispVal ;     // VT_BYREF|VT_DISPATCH
      SAFEARRAY * *     pparray ;     // VT_BYREF|VT_ARRAY
      VARIANT *     pvarVal ;     // VT_BYREF|VT_VARIANT
      PVOID     byref ;     // Generic ByRef
      CHAR     cVal ;     // VT_I1
      USHORT     uiVal ;     // VT_UI2
      ULONG     ulVal ;     // VT_UI4
      INT     intVal ;     // VT_INT
      UINT     uintVal ;     // VT_UINT
      DECIMAL *     pdecVal ;     // VT_BYREF|VT_DECIMAL
      CHAR *     pcVal ;     // VT_BYREF|VT_I1
      USHORT *     puiVal ;     // VT_BYREF|VT_UI2
      ULONG *     pulVal ;     // VT_BYREF|VT_UI4
      INT *     pintVal ;     // VT_BYREF|VT_INT
```

```
        UINT *      puintVal ;     // VT_BYREF|VT_UINT
      } _VARIANT_NAME_3 ;
    } _VARIANT_NAME_2 ;

    DECIMAL decVal ;
  } _VARIANT_NAME_1 ;
} VARIANT ;
```

可以看到 VARIANT 只不过是许多不同类型值的一个大联合。在 Visual Basic 中经常使用 VARIANT 来按相同的方式保存不同类型变量的值。这是 Visual Basic 开发小组给我们带来的一个极好的思想。下面我们将介绍如何使用这些变量。但应记住的是，能够传递给调度接口及双重接口的参数类型将仅仅是那些可以用 VARIANT 表达的类型。

三、返回值的获取

第六个参数 pVarResult 为指向一个 VARIANT 结构的指针，此结构将被用于保存 Invoke 所执行的函数或 propget 的结果。对于没有返回值的成员函数或 propputs 以及 propputrefs，此参数值可以为 NULL。

四、例外情形

IDispatch::Invoke 的倒数第二个参数为指向一个 EXCEPINFO 结构的指针。若 Invoke 执行的函数或属性遇到一个例外情况，则此结构中将被填入关于此例外情况的一些信息。至于在哪些情况下可以使用 EXCEPINFO 结构，可以参考在 C++ 中使用异常处理的情况。

在下面的 EXCEPINFO 的定义中，BSTR 是一个具有特殊格式的字符串。后面我们将详细讨论它。

```
typedef struct tagEXCEPINFO {
  WORD  wCode ;              // Error code
  WORD  wReserved ;
  BSTR  bstrSource ;         // Source of the exception
  BSTR  bstrDescription ;    // Description of the error
  BSTR  bstrHelpFile ;       // Full pathname
  DWORD dwHelpContext ;      // Help context inside help file
  ULONG pvReserved ;
  ULONG pfnDeferredFillIn ;  // Function to fill in structure
  SCODE scode ;     // Return value
} EXCEPINFO ;
```

在错误代码(wCode)或返回值(scode)中必须包含一个标识错误的值，而另外一个则必须被设置为 0。下面是使用 EXCEPINFO 结构的一个例子。

```
EXCEPINFO excepinfo ;
HRESULT hr = pIDispatch-> Invoke(..., &excepinfo) ;
```

```
if (FAILED(hr))
{
  // Invoke failed.
  if (hr == DISP_E_EXCEPTION)
  {
    // Method raised an exception.
    // Server can defer filling EXCEPINFO.
    if (excepinfo.pfnDeferredFillIn != NULL)
    {
      // Fill the EXCEPINFO structure.
      (*(excepinfo.pfnDeferredFillIn))(&excepinfo);
    }
    strstream sout;
    sout << "Exception information from component:" << endl
        <<" Source:    "<< excepinfo.bstrSource << endl
        <<" Description:" << excepinfo.bstrDescription
        << ends;
    trace(sout.str());
  }
}
```

五、参数错误

若 IDispatch::Invoke 的返回值为 DISP_E_PARAMNOTFOUND 或 DISP_E_TYPEMISMATCH,此时 Invoke 将把与此错误相应的参数的索引保存在最后一个参数 puArgErr 中。

在介绍了 Invoke 的所有参数之后,下面我们来看一下调用调度接口中函数的另外一个例子。然后我们将更详细地讨论 VARIANT。同时我们还将讨论能够由 VARIANT 表达的另外两种数据类型:BSTR 和 SAFEARRAY。

11.2.2 示例

在本章的例子中我们将构造一个实现双重接口 IX 的组件。接口 IX 的 IDL 定义如下:

```
// Interface IX
[
    object,
    uuid(32BB8326-B41B-11CF-A6BB-0080C7B2D682),
    helpstring("IX Interface"),
    pointer_default(unique),
    dual,
    oleautomation
]
interface IX : IDispatch
```

```
{
    import "oaidl.idl";
    HRESULT Fx();
    HRESULT FxStringIn([in] BSTR bstrIn);
    HRESULT FxStringOut([out, retval] BSTR* pbstrOut);
    HRESULT FxFakeError();
};
```

然后我们可以构造两个客户以连接到此组件上。其中一个客户将像本章前面例子中的客户那样通过 vtbl 连接到此组件上，另外一个客户则可以通过调度接口连接到组件上。本章前面我们已经介绍了它可以如何调用函数 Fx。下面我们来看一下此客户如何调用 FxStringIn。如下面的代码所示。

```
// Get the dispid for the method.
DISPID dispid;
OLECHAR* name = L"FxStringIn";
HRESULT hr = pIDispatch->GetIDsOfNames(IID_NULL,
                &name,
                1,
                GetUserDefaultLCID(),
                &dispid);

// Pass the following string to the component.
wchar_t wszIn[] = L"Test string from client.";

// Convert the wide-character string to a BSTR.
BSTR bstrIn;
bstrIn = ::SysAllocString(wszIn);

// Allocate and initialize a VARIANT argument.
VARIANTARG varg;
::VariantInit(&varg);    // Initialize the VARIANT.
varg.vt = VT_BSTR;    // Type of VARIANT data
varg.bstrVal = bstrIn;  // Data for the VARIANT

// Fill in the DISPPARAMS structure.
DISPPARAMS param;
param.cArgs = 1;    // One argument
param.rgvarg = &varg;  // Pointer to argument
param.cNamedArgs = 0;  // No named args
param.rgdispidNamedArgs = NULL;

hr = pIDispatch->Invoke(dispid,
            IID_NULL,
            GetUserDefaultLCID(),
            DISPATCH_METHOD,
            &param,
            NULL,
```

```
            NULL,
            NULL) ;
    // Clean up
    ::SysFreeString(bstrIn) ;
```

对于 VARIANTARG 和 DISPPARAMS 结构的填充需要许多代码。幸运的是可以编写一些帮助函数以减少调用 Invoke 的繁琐程度。在 MFC 中也可以发现一些能够使对 Invoke 调用处理更为容易的一些帮助函数。ClassWizard 也可以为调度接口生成一个 C++ 包装类。在这些包装类中,包含一些 C++ 风格的函数,它们可以把各个参数转换成适合于 Invoke 调用的格式。

下面我们来更详细地看一个 VARIANT 类型。同时我们还将讨论一下 BSTR 类型及 SAFEARRAY 类型。

11.2.3 VARIANT 类型

前面我们已经给出了 VARIANT(及 VARIANTARG)结构的定义。下面我们来看一下如何使用这一结构。在前面的代码中我们已经看到,对于 VARIANT 结构的初始化是借助于函数 VariantInit 完成的。VariantInit 将把 vt 域设置为 VT_EMPTY。在调用了 VariantInit 之后,vt 域将被用于指示在 VARIANT 的联合体中保存的数据的类型。在前面的例子中,由于我们需要保存一个 BSTR 值,故使用了 bstrVar 域。

一、运行时联编

在使用 C++ 类或 COM 接口时,可以用一个头文件来描述相应类或接口中函数的所有参数。编译器可以保证在编译时传给每一个函数的参数的类型都是正确的。这种强类型检查对于生成健壮的代码是一个重要的特性。但在编写一个灵活性与简单性比类型安全更为重要的简单的宏时,这种特性可能会成为一个致命的缺点。

读者可能尚未注意到,我们没有为 Visual Basic 提供一个与 C++ 的头文件相等价的东西。Visual Basic 程序在调用一个调度接口方法之前,并不需要知道此方法需要哪些参数。正是 VARIANT 使这一点成为可能。假定一个 Visual Basic 程序中有如下的代码:

```
Dim Bullwinkle As Object
Set Bullwinkle = CreateObject("TalkingMoose")
Bullwinkle.PullFromHat 1, "Topolino"
```

对于 Bullwinkle,Visual Basic 除了知道它支持 IDispatch 接口之外可以说是一无所知。由于 Buulwinkle 支持 IDispatch,因此 Visual Basic 通过调用 IDispatch::GetIDsFromNames 来获取 PullFromHat 的 DISPID,但它并不知道 PullFromHat 参数的任何情况。此时我们可以求助于 C++ 头文件的一种与语言无关的替换物:类型库。关于这一点我们将在本章的后面加以讨论。

但实际上，Visual Basic 并不需要头文件或相等价的东西来告诉它哪些类型是可以使用的。它可以接收用户输入的任何参数并将其填充到 VARIANT 结构中。在前面的例子中，Visual Basic 可能会认为第一个参数为 long 类型的，而第二个参数为一个 BSTR 值。然后它将把这些 VARIANT 值传给 Invoke 函数。若所传的参数有误，自动化服务器将返回一个错误，它可能会带有一个不正确参数的索引。为了避免这些问题，程序员自然需要一些关于这些函数的一些文档，以便能够正确地调用它们。但程序却不需要任何类型信息。这样，使用 VARIANT，几乎完全可以摆脱静态类型检查，而让组件在运行时进行类型检查。这一点同没有类型检查的 Smalltalk 更为类似，而与 C++ 这种强静态类型检查语言大相径庭。

将类型检查推迟到运行时完成需要调度接口的方法和函数对它们接受到的参数的类型进行检查。它们必须保证传给它们的值都是正确的，否则将可能导致自动化服务器的崩溃。

二、类型转换

虽然好的调度接口可以为它所不处理的类型返回相应的错误，但一个真正优秀的调度接口却可以为程序员完成某些类型的转换。以前面代码中的 PullFromHat 函数为例。Visual Basic 可能会认为此函数将接收一个 long 参数及一个 BSTR 参数。但此函数真正需要的参数也许是一个 double 值而不是一个 long 值。此时调度接口需要能够进行这种转换。另外 BSTR 值的转换应由调度接口完成。例如，若用 IDL 定义如下的属性设置函数：

```
[propput] HRESULT Title([in] BSTR bstrTitle);
```

在用如下的代码调用此函数时：

```
component.Title = 100
```

应将数 100 转换成一个 BSTR 值并将其作为标题值。与此类似，对于按如下方式定义的属性设置函数：

```
[propput] HRESULT Age([in] short sAge);
```

也应该能够正确地处理如下的 Visual Basic 调用：

```
component.Age = "16"
```

相信记者一定不希望自己来编写这些类型转换代码。即使读者自己愿意，也一定有人不愿意。糟糕的是，如果每个人都编写这些代码，那么这些代码将可能是各不相同的。这将使得某些函数和属性是按一种方式进行转换的，而其他的函数和属性则将按其他的方式进行转换。为此，自动化提供了一个名为 VariantChangeType 的函数，以完成这些转换：

```
HRESULT VariantChangeType(
    VARIANTARG *        pVarDest, // Coerced argument
    VARIANTARG *        pVarSrc,  // Argument to be coerced
    unsigned short      wFlags,
    VARTYPE             vtNew     // Type to coerce to
  );
```

使用这个函数的方法是相当简单的。例如下面的函数将使用 VariantChangeType 将一个 VARIANT 值转换成一个 double 值：

```
BOOL VariantToDouble(VARIANTARG * pvarSrc, double * pd)
{
    VARIANTARG varDest ;
    VariantInit(&varDest) ;
    HRESULT hr = VariantChangeType(&varDest,
                pvarSrc,
                0, VT_R8) ;
    if (FAILED(hr))
    {
      return FALSE ;
    }
    *pd = varOut.dblVal ;
    return TRUE ;
```

三、可选的参数

调度接口中的方法可能有一些可选的参数。若不想给这些参数提供一个值，则只需传递一个 vt 域被设置为 VT_ERROR 而 scode 域被设置为 DISP_E_PARAMNOTFOUND 的 VARIANT 结构即可。而相应的方法则应提供它自己的缺省值。

下面我们来看一下 BSTR。

11.2.4 BSTR 数据类型

BSTR 是 Basic 字符串(Basic String)或二进制字符串(binary string)的缩写，它实际上是指向一个宽字符串的指针。关于 BSTR 有三个令人感兴趣的方面。首先 BSTR 是一个带有字符计数值的字符串。BSTR 的第二个重要的特性是字符计数值保存在字符数组的开头(参见图 11-4)。因此，不能按下面的方法声明一个 BSTR 变量并给它赋一个字符数组值：

```
BSTR bstr = L"Where is the count?" ; // Wrong!
```

这是因为上面的方法没有正确地设置计数值。为给 BSTR 变量赋值，可以使用 Win32 API 函数 SysAllocString：

```
wchar_t wsz[] = L"There's the count." ;
BSTR bstr ;
bstr = SysAllocString(wsz) ;
```

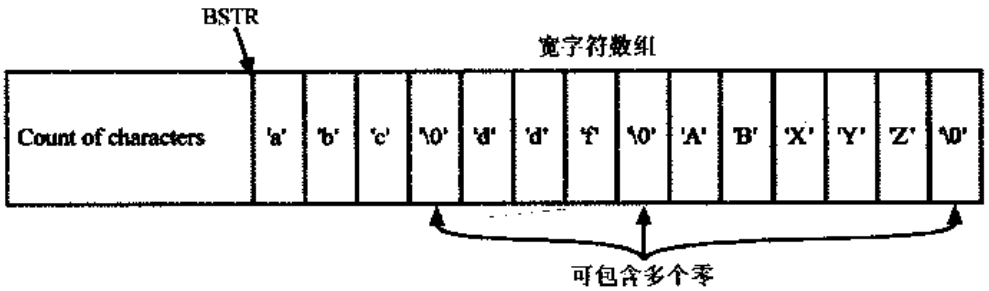


图 11-4 字符计数值将被保存在指向字符的指针的前面

在分配一个 BSTR 结构之后，需要用 SysFreeString 将其释放。将一个 BSTR 转换成一个 wchar_t 串也是相当容易的。这是因为 BSTR 指向的实际上就是一个 wchar_t 串。但 BSTR 第三个令人感兴趣的方面在于在它所包含的字符串中可以有多个'\0'。因此，当这一点可能对程序产生影响时，需要编写额外的代码对之进行处理。

11.2.5 SAFEARRAY 类型

调度接口可以传递的另外一种特殊的数据类型是 SAFEARRAY。这种类型如同其名字所表示的那样是一个包含有边界信息的数组。下面是从 OAIDL.IDL 中摘录出来的此类型的定义。

```
typedef struct tagSAFEARRAY{
    unsigned short cDims ;     // Number of dimensions
    unsigned short fFeatures ;
    unsigned long cbElements ;  // Size of each element
    unsigned long cLocks ;     // Lock count
    BYTE* pvData ;          // Pointer to data
    [size_is(cDims)] SAFEARRAYBOUND rgsabound[] ;
} SAFEARRAY ;

typedef struct tagSAFEARRAYBOUND{
    ULONG cElements ;// Number of elements in dimension
    LONG lLbound ;  // Lower bound of dimension
} SAFEARRAYBOUND ;
```

其中 fFeatures 域表示的是保存在 SAFEARRAY 中的数据的类型。此域的值可以是：

FADF_BSTR	BSTR 数组
FADF_UNKNOWN	IUnkown* 数组
FADF_DISPATCH	IDispatch* 数组
FADF_VARIANT	VARIANT 数组

下列 fFeatures 值描述的是相应的数组是如何分配的：

FADF _ AUTO	数组是分配在栈上的。
FADF _ STATIC	数组是静态分配的。
FADF _ EMBEDDED	数组嵌入在一个结构中。
FADF _ FIXEDSIZE	数组的大小不能被改变或被重新分配。

在自动化库 OLEAUT32.DLL 中包含许多操作 SAFEARRAY 的函数。这些函数的名称前面都有一个 SafeArray 前缀。在需要用到它们时 可以从联机帮助系统中查找。

前面我们介绍了如何填充 VARIANT 结构以构造 DISPPARAMS 结构。此结构将被传递给 IDispatch::Invoke 以调用调度接口中的方法或访问其中的属性。下面我们将介绍类型库，它是一种与语言无关的 C++ 头文件替换物。

11.3 类型库

前面我们已经看到，Visual Basic 或 C++ 程序可以通过一个调度接口在不需要知道此接口或其方法任何类型信息的情况下控制相应的组件。但正如我们可以将豌豆放到自己的鼻孔里并不意味着可以这样做一样，在不需要类型信息的情况下可以编写出一个 Visual Basic 程序并不意味着可以这样做。

这是因为，在前一节我们介绍的运行时类型检查与转换的开销是非常大的，并且可能会导致一些难以发现的错误，如程序员在调用某个函数时可能会不经意地互换两个参数，经过组件的转换，这两个参数可能都符合被调用函数的需要。C++ 比之 C 的一大优点就是更为严格的运行时类型检查，这可以程序的正常运行提供某种程度的保证。

因此，我们需要一种与语言无关的、适合于解释性语言和宏编程语言使用的 C++ 头文件的等价物。COM 解决这一问题的方案是类型库。类型库将提供有关组件、接口、方法、属性、参数及结构的类型信息。类型库中的内容同 C++ 头文件中的内容是相同的，它实际上是 IDL 文件的一个编译版本，并且可以用编程的方法来访问。类型库不是一个与语言无关的需要被分析的文本文件，而是一个二进制文件。自动化库为创建和读取此二进制文件提供了一些标准的组件。

没有类型库，Visual Basic 通过调度接口与组件进行通信将会受到一些限制。有了类型库之后，Visual Basic 将可以通过组件双重接口的 vtbl 部分来访问此组件。通过 vtbl 的访问将是类型安全的，而且更为高效。

类型库中也可以包含库中组件、接口以及函数的帮助串。使用一个对象浏览器（如 Visual Basic 所提供的），程序员将可以方便地获取任意属性或方法的帮助说明。

11.3.1 类型库的创建

自动化库函数 CreateTypeLib 可以创建一个类型库，该函数将返回一个可以用相应的

信息填充类型库的 ICreatetypeLib 接口。但程序员一般很少使用此接口，但可以使用 IDL 及 MIDL 编译器。在第 10 章中我们介绍了如何用 IDL 和 MIDL 编译器以生成代理/残根 DLL 的代码。实际上 IDL 和 MIDL 也可以用于建立类型库。

ODL 和 MkTypLib

以前 MIDL 编译器不能用于建立类型库。那时描述类型库所用的不是 IDL，而是另外一种名为 ODL 的语言。相应地有一个名为 MkTypLib 的工具可以将 ODL 文件编译成类型库，ODL 同 IDL 是类似的，但差别也是相当大的，同时使用这两种语言不但容易造成混淆，而且要维护两个不同的文件以使之包含相同的信息将是非常费时的。所幸的是，在开发 Windows NT 4.0 时，IDL 和 MIDL 都被进行相应的扩展以支持类型库的建立。因此，现在已经不再需要 ODL 和 MkTypLib 了。

一、library 语句

使用 IDL 建立类型库的关键是 library 语句。library 块中出现的所有内容——library 关键字之后的方括号中的内容——都将被编译到类型库中。程序清单 11-1 列出了本章例子中所用的 IDL 文件。可以看到，一个类型库包含一个 GUID、一个版本号和一个帮助串。

程序清单 11-1 生成类型库 SERVER.TLB 的 IDL 文件。

SERVER.IDL

```
//
// Server.idl - IDL source for Server.dll
//
// This file will be processed by the MIDL compiler to
// produce the type library (Server.tlb) and marshaling code.
//

// Interface IX
[
    object,
    uuid(32BB8326-B41B-11CF-A6BB-0080C7B2D682),
    helpstring("IX Interface"),
    pointer_default(unique),
    dual,
    oleautomation
]
interface IX : IDispatch
{
    import "oaidl.idl";
    HRESULT Fx();
```

```
    HRESULT FxStringIn([in] BSTR bstrIn);
    HRESULT FxStringOut([out, retval] BSTR* pbstrOut);
    HRESULT FxFakeError();
};
//
// Component and type library descriptions
//
[
    uuid(D30115E1-B997-11CF-A6BB-0080C7B2D682),
    version(1.0),
    helpstring("Inside COM, Chapter 11 1.0 Type Library")
]
library ServerLib
{
    importlib("stdole32.tlb");
    // Component
    [
        uuid(0C092C2C-882C-11CF-A6BB-0080C7B2D682),
        helpstring("Component Class")
    ]
    coclass Component
    {
        [default] interface IX;
    };
};
```

coclass 语句将定义一个组件。本例中所定义的组件为具有一个 IX 接口的 Component。MIDL 编译器将生成一个包含 Component 和 IX 的类型库，Component 之所以会被加入到类型库中是因为它的 coclass 语句是包含 library 语句中的。而接口 IX 将被包含在类型库中是因为在 library 语句中引用了它。

当 MIDL 遇到 IDL 文件中的 library 语句时，它将自动生成一个类型库。在第 10 章中我们已经看到 MIDL 生成了一个名为 SERVER.TLB 的类型库，只不过在那里我们不需要它。

二、类型库的分发

在生成了类型库之后，既可以将其作为一个单独的文件发行，也可以将其作为一个资源包含在 EXE 或 DLL 中。大多数开发者一般采用的是后一种方法，以便使安装过程得以简化。

11.3.2 类型库的使用

使用类型库的第一步是装载它。类型库的装载有几个不同的函数。首先一个是LoadRegTypeLib,它将试图从Windows的注册表中装载指定的类型库。若此函数失败,则可以试一下LoadTypeLib(这个函数将从硬盘上装载指定文件名称的类型库)或者是LoadTypeLibFromResource(它将从一个EXE或DLL中装载指定的类型库)。LoadTypeLib将在装载类型库时为之在注册表中进行注册。但若提供给LoadTypeLib的是一个完整的路径名称,那么它将不会进行注册(参阅PSS ID Number Q131055)。此时在LoadTypeLib成功之后需要调用RegisterTypeLib。完成这些操作的代码如程序清单11-2所列。

程序清单11-2 类型库的装载、注册和使用。

```
HRESULT CA::Init()
{
  HRESULT hr;
  // Load TypeInfo on demand if we haven't already loaded it.
  if (m_pITypeInfo == NULL)
  {
    ITypeLib* pITypeLib = NULL;
    hr = ::LoadRegTypeLib(LIBID_ServerLib,
          1, 0, // Version numbers
          0x00,
          &pITypeLib);
    if (FAILED(hr))
    {
      // Load and register the type library.
      hr = ::LoadTypeLib(L"Server.tlb",
          &pITypeLib);
      if (FAILED(hr))
      {
        trace("LoadTypeLib Failed.", hr);
        return hr;
      }
      // Ensure that the type library is registered.
      hr = RegisterTypeLib(pITypeLib,
          L"Server.tlb", NULL);
      if (FAILED(hr))
      {
        trace("RegisterTypeLib Failed.", hr);
        return hr;
      }
    }
    // Get type information for the interface of the object.
```

```
    hr = pITypeLib->GetTypeInfoOfGuid(IID_IX,
            &m_pITypeInfo);
    pITypeLib->Release();
    if (FAILED(hr))
    {
      trace("GetTypeInfoOfGuid failed.", hr);
      return hr;
    }
  }
  return S_OK;
}
```

在装载了一个类型之后，即可使用它。LoadTypeLib 及其他函数将返回一个 ITypeLib 接口指针，此指针可以用于对类型的处理。此时我们经常需要做的是获取关于某个接口或组件的信息。而类型库中保存的正是关于组件和接口的信息。为获取关于某个组件或接口的信息，可以将其 CLSID 或 IID 传给函数 ITypeLib::GetTypeInfoOfGuid。此函数将返回一个指向所需项目的 ITypeInfo 指针。

使用 ITypeInfo 指针，可以获取所需的关于组件、接口、方法、属性、结构或其他类似对象的任何信息。但大多数 C++ 组件程序员除了在实现 IDispatch 接口时之外从来不用这些信息，下面我们将会看到 ITypeInfo 可以自动实现 IDispatch。

上述接口最常见的用途之一是在类型库浏览器——为程序员显示类型库中信息的程序——中，Visual Basic 带有一个名为 Object Browser 的类型库浏览器。使用 Object Browser 我们可以找到属于某个接口的特定方法，并获取同它相关联的帮助信息。OleView 也有一个类型库浏览器。OleView 的好处在于它可以读取类型库中的信息，并据此建立与 ODL/IDL 相似的文件。这是一个很有用的特性。

11.3.3 注册表中的类型库

作者非常喜欢那些能够在注册表中登记它们的组件，而对于编写向注册表中保存信息的代码则感到一阵阵的“恐惧”。所幸的是，类型库可以自己进行注册。读者或许非常想知道类型库将把哪些信息放到注册表中。为此可运行 REGEDIT.EXE，在 HKEY_CLASSED_ROOT\TypeLib 关键字下，我们可以看到一个 LIBID 列表（LIBID 是一个标识类型为的 GUID）。打开其中的某个 GUID 可以看到与图 11-5 所示类似的信息。

有一样东西类型库没有注册：组件所需的指向其注册表中类型库信息的指针。为此需要在组件的 CLSID 下加上一个名为 TypeLib 的关键字，并将相应类型库的 GUID 作为它的值。例如下面的关键字应该包含相应类型库的 LIBID 值。

```
HKEY_CLASS_ROOT\
    CLSID\
        {0C092C29-882C-11CF-A6BB-0080C7B2D682}\
```

TypeLib

但对于类型库中的接口，类型库却将为之注册一个相应的 TypeLib 关键字。

在实现 IDispatch 接口时，类型库将是非常重要的。关于这一点我们将在下一节介绍。

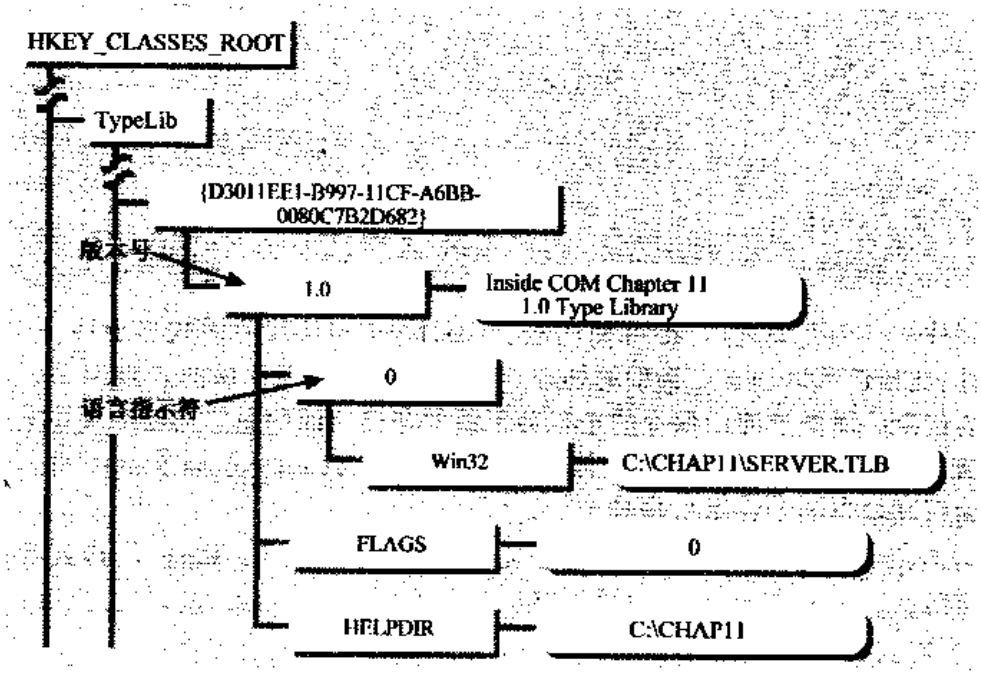


图 11-5 类型库加入到 Windows 注册表中的信息

11.4 IDispatch 接口的实现

实现 IDispatch 接口的方法可能同剥猫皮的方法一样多。MFC 将建立其自己的名称及函数指针表。但使用 MFC 来实现双重接口却是极不好的一种方法。本节我们将讨论实现 IDispatch 接口的最简单也是最为常用的一种方法。这种方法的基本策略是将 GetIDsOfNames 及 Invoke 转发给与我们的接口相应的 ITypeInfo 接口指针。

前面我们已经介绍了如何获取一个接口的 ITypeinfo 指针。这只需装载相应的类库，然后调用 ITypeLib::GetTypeInfoOfGuid，并将相应接口的 IID 传给此函数。GetTypeInfoOfGuid 将返回一个 ITypeInfo 接口指针，此指针可以用来实现 IDispatch 接口。如下面的代码所示：

```
HRESULT _stdcall CA::GetTypeInfoCount(UINT* pCountTypeInfo)
{
  *pCountTypeInfo = 1;
  return S_OK;
}
```

```
HRESULT _stdcall CA::GetTypeInfo(
  UINT iTypeInfo,
  LCID, // This object does not support localization.
  ITypeInfo * * ppITypeInfo)
{
  * ppITypeInfo = NULL ;
  if (iTypeInfo ! = 0)
  {
    return DISP _E _BADINDEX ;
  }
  // Call AddRef and return the pointer.
  m _pITypeInfo-> AddRef() ;
  * ppITypeInfo = m _pITypeInfo ;
  return S _OK ;
}
HRESULT _stdcall CA::GetIDsOfNames(
  const IID& iid,
  OLECHAR * * arrayNames,
  UINT countNames,
  LCID, // Localization is not supported.
  DISPID * arrayDispIDs)
{
  if (iid ! = IID _NULL)
  {
    return DISP _E _UNKNOWNINTERFACE ;
  }
  HRESULT hr = m _pITypeInfo-> GetIDsOfNames(arrayNames,
               countNames,
               arrayDispIDs) ;
  return hr ;
}
HRESULT _stdcall CA::Invoke(
  DISPID dispidMember,
  const IID& iid,
  LCID,  // Localization is not supported.
  WORD wFlags,
  DISPPARAMS * pDispParams,
  VARIANT * pvarResult,
  EXCEPINFO * pExcepInfo,
  UINT * pArgErr)
{
  if (iid ! = IID _NULL)
  {
```

```
    return DISP_E_UNKNOWNINTERFACE;
  }
  ::SetErrorInfo(0, NULL);
  HRESULT hr = m_pITypeInfo->Invoke(
    static_cast<IDispatch*>(this),
    dispidMember, wFlags, pDispParams,
    pvarResult, pExcepInfo, pArgErr);
  return hr;
}
```

可以看到上述方法确实是非常简单的,但它也有一些不足。例如它不支持应用程序的国际化。所幸的是,许多组件并不需要被国际化。对于那些需要被国际化的组件,可以根据传递给 Invoke 的 LCID 来装载相应的类型库。

11.4.1 异常的引发

前面我们已经介绍过,传给 IDispatch::Invoke 的倒数第二个参数为一个 EXCEPINFO 结构。为使 ITypeInfo::Invoke 填充此结构,需要如下的一些步骤:

1 在组件中实现 ISupportErrorInfo 接口。此接口只有一个成员函数:

```
// ISupportErrorInfo
virtual HRESULT _stdcall InterfaceSupportsErrorInfo(const IID&iid)
{
    return (iid == IID_IX) ? S_OK : S_FALSE;
}
```

2 在 IDispatch::Invoke 的实现中,于 ITypeInfo::Invoke 调用之前调用 SetErrorInfo(0, NULL)。

3 当发生异常时,调用 CreateErrorInfo 以获取一个 ICreateErrroInfo 接口指针。

4 使用 ICreateErrorInfo 填充关于错误的信息。

5 最后调用 SetErrorInfo,并将 ICrateErrroInfo 接口指针作为第二个参数传给它。第一个参数是保留的,恒为 0。其余的参数取决于 ITypeInfo 和客户。

下面给出的是一个引起异常的例子。

```
// Create the error info object.
ICreateErrorInfo* pICreateErr;
HRESULT hr = ::CreateErrorInfo(&pICreateErr);
if (FAILED(hr))
{
    return E_FAIL;
}
// pICreateErr->SetHelpFile(...);
// pICreateErr->SetHelpContext(...);
```

```
    pICreateErr->SetSource(L"InsideCOM.Chap11.Cmpnt");
    pICreateErr->SetDescription(
        L"This is a fake error generated by the component.");
    IErrorInfo* pIErrorInfo = NULL;
    hr = pICreateErr->QueryInterface(IID_IErrorInfo,
                    (void**)&pIErrorInfo);

        if (SUCCEEDED(hr))
        {
            ::SetErrorInfo(0L, pIErrorInfo);
            pIErrorInfo->Release();
        }
        pICreateErr->Release();
        return E_FAIL;
    };
```

11.4.2 参数调整

在本章我们构造的将不再是一个代理/残根 DLL 了。这是由于系统,更具体地讲是 OLEAUT32.DLL 将为与自动化兼容的接口实现参数调整[①]。一个与自动化兼容的接口将继承 IDispatch,并只使用那些可以被包含在一个 VARIANT 结构中的类型作为其参数。可以被包含在 VARIANT 中的类型将被 OLEAUT32.DLL 进行调整。

为搞清楚这一点是如何工作的,可以看一下注册表中的一个接口 IX:

```
HKEY_CLASSES_ROOT\
    Interfaces\
        {32BB8326-B41B-11CF-A6BB-0080C7B2D682}\
        ProxyStubClsid32
```

它应具有如下的 CLSID:

```
{00020424-0000-0000-C000-00000000046}
```

当在 CLSID 关键字下查找此 CLSID 值时:

```
HKEY_CLASSED_ROOT\
    CLSID\
        {00020424-0000-0000-C00000000000046}
            InprocServer32
```

将可以看到 InprocServer32 的值为 OLEAUT32.DLL。

---

① 应该指出的是,前面我们用 MIDL 生成的代码来构造代理/残根 DLL 的方法在这种情况下是不能正常工作的,至少是在 Windows 95 或 Windows NT 4.0 以前的系统上是如此。MIDL 编译器不能为这些系统中的 VARIANT 或 BSTR 生成调整代码。所以如果不想用 OLEAUT32.DLL 的话,则需要自己编写调整代码。

11.5 本章小结

至此我们就介绍完了另外一种组件和客户之间进行通信的方法。当完成某件事情的方法不只一种时，就需要决定到底应使用哪一种方法。那么对于我们可以使用的三种方法：vtbl 接口、双重接口以及调度接口，到底应使用哪一种方法呢？为些可以遵循如下的一些原则。

若组件只被编译型语言如 C 或 C++ 访问，则可以使用 vtbl 或常规的 COM 接口。vtbl 接口比调度接口要快得多，并且对于 C++ 程序员来说，使用起来也更为容易。若组件将被 Visual Basic 或 Java 访问，则应该实现一个双重接口。此时 Visual Basic 和 Java 将能够通过双重接口的调度接口部分或 vtbl 部分来访问它。C++ 用户也以通过双重接口的这两个部分来访问组件。

即便是专门设计来供 Visual Basic 访问的调度接口的 vtbl 部分也不能让大多数 C++ 程序员满意(除非是使用 Visual C++ 5.0 中的编译器扩展)。因此可以实现低级的 vtbl 接口以及一个高级的双重接口。低级接口可以给 C++ 程序员提供对组件进行有效的聚合所需要的额外信息。

如果不是确实需要在运行时创建组件，则不要去实现纯的双重接口。双重接口比调度接口要脆弱得多。

另外一个决定选择的因素可能是速度问题。对于进程中组件，vtbl 接口比调度接口快上 100 倍。(实际的差别将同函数参数的类型有一些关系。)对于进程外组件，参数调整的开销将比 IDispatch::Invoke 开销更大，此时 vtbl 接口仅比调度接口快 2.5 倍。对于远程组件，各种不同类型的接口几乎没有什么差别。

第12章 多 线 程

到作者办公室来的人经常会将他们的头碰到天花板上挂着的一个长约12英寸的黑色物体上。它是一个1:32的Bell 206B-III Jet Ranger(一种喷气式飞机)的比例模型。它的螺旋桨是后向推进式的,而不是侧向的尾部型,因此,它并不是一个精确的模型。这种后向推动式的螺旋桨将使它能够按照一个圆圈"飞行"。

它的工作方式是:打开电源之后,一个小电动机将驱动后部的螺旋桨转动。螺旋桨提供的推动力并不足以使它飞起来。为此需要将它轻轻地推动一下,以使之摇晃起来。随后,螺旋桨提供的推动力将逐渐增大以使整个模型旋转起来。悬挂线与天花板之间的夹角将越来越小,最后飞机模型将在靠近天花板的下方飞快地旋转。

这个小飞机模型是非常特别的。它是Ruediger Asche送给作者的。Ruediger曾经和作者一起为Microsoft Developer Network撰写文章。他对于Windows NT的内核有着极深的造诣,这些是GUI绝不会涉及到的。Reudiger的一个专长之一是多线程编程。而这也正是本章要介绍的主要内容。

若想编写一个飞机模型的模拟程序,可能需要用到多线程。其中一个线程用以管理用户界面,以使用户能够操作此模型的三维视图。另外一个线程可以计算此模型在转动过程中的位置。

但模拟一个旋转的飞机模型并不一定需要使用多线程。多线程最有用的地方是提供具有比较好响应性的用户界面。例如,将某些计算任务交给后台的工作线程来完成将使用户界面能够更快地供用户使用。这一点在Web浏览器中得到了最好的体现。大多数Web浏览器都用一个线程来下载页面,而用另外一个线程来显示它,为使在下载过程中用户能够对页面进行操作,还可以借助第三个线程,例如大多数人在浏览过程中都不会等着那些无用的图片的下载,而是一个劲地单击他们感兴趣的超链。这一点正是借助多线程才成为可能。

由于线程对于反应快的应用程序是非常重要的,因此我们自然也希望能够让多个线程同时访问某个组件。但让多个线程同时访问组件将会带来一些其他地方不会遇到的问题。关于这一点我们将在本章详加讨论。当然这些问题同使用多线程会遇到的一些更具一般性的问题比较起来,可以说是次要的,或者说是不重要的。本章将不详细讨论多线程编程的问题,而只是介绍多线程对程序的开发和COM组件的使用带来了一些什么样的影响。有关多线程编程的详细介绍,可以参阅MSDN中Ruediger Asche的文章。

12.1 COM线程模型

COM并没有定义新的线程和进程模型，而是直接使用了Win32线程。COM也没有它自己的同步原语。COM中线程的创建和同步使用的都是常规的Win32 API。COM中线程的使用同在Win32应用程序中使用线程基本上是类似的，当然这并不排除它们之间的一些细微差异。本章将讨论这些差异。但此前我们将先讨论一下Win32的线程。

12.1.1 Win32线程

典型的Win32线程具有两种不同类型的线程：用户界面线程和工作线程。用户线程是同一个或多个窗口关联着的。这些线程具有自己的消息循环，以便能对用户的输入作出反应。工作线程用于后台处理，它们没有相关联的窗口，通常也没有消息循环。一个进程可以具有多个用户界面线程和多个工作线程。

用户界面线程的行为是令人感兴趣的。前面我们已经提到，每一个用户界面线程都有一个或多个窗口。线程将调用它所拥有的窗口的窗口过程。对于任一窗口，它将被创建它的线程所拥有。所以窗口过程总是在同一线程中被执行的，而不论被处理的消息是哪个线程发出的。这样做的结果是所有的消息被同步。窗口将可以保证它能按正确的顺序获取各个消息。

这样，程序员将用不着编写线程安全的窗口过程。编写线程安全的代码不是一件容易的事情，可能极费时间。由于Windows保证了消息的同步，因此程序员用不着考虑两个或多个线程同时访问同一个过程时会出现什么问题。这种同步对于管理用户界面是非常有益的，因为我们本来就希望用户事件能够按它们被执行的顺序到达窗口。

12.1.2 COM线程

COM使用的线程的类型与Win32的两种类型线程是相同的，只不过是换了不同的名称而已。与Win32中的用户界面线程相对应，COM有一个“套间线程”，而与工作线程相对应的则的一个自由线程。COM线程最为困难的部分在于其术语，而这些术语最困难的部分在于没有任何文档是一致的。Win32 SKD使用的是与COM规范完全不同的术语集。作者将尽量避免使用那些术语，而是定义自己的术语。在这一章中我们将把用户界面风格的线程称为是套间线程，而将一个工作线程称作是自由线程。

既然COM线程与Win32并没有什么差别，那么为什么COM还需要定义自己的线程呢？其原因在于参数调整与线程同步。关于这一问题，我们将在定义了套间、套间线程以及自由线程后详细讨论。

12.1.3 套间

虽然作者确实不想引进新的术语，但我们首先还是需要定义术语“套间”。它实际上是一个由用户界面风格的线程(套间线程)和一个消息循环构成的概念性实体。

我们知道一个 Win32 应用程序是由一个进程、一个消息循环和一个窗口过程组成的。任一进程至少有一个线程。一个 Windows 应用程序可以用图 12-1 表示。图中虚线框表示的是一个进程。其中有一个环的框表示消息循环，另外一个框表示的是窗口过程和程序中的其他代码。所有这些都位于一个表示执行线程的箭头的上方。

除了可以表示一个线程外，图 12-1 也可以表示一个套间。此处的唯一的线程即为套间线程。

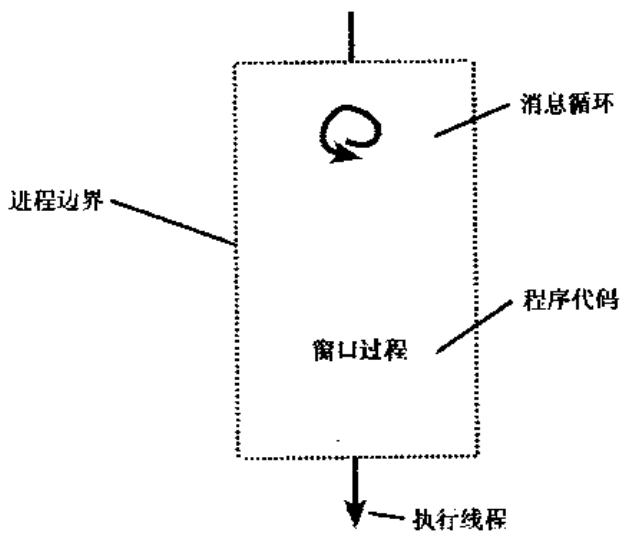


图 12-1 Windows 应用程序图示：它包括线程执行、消息循环、进程边界以及程序代码

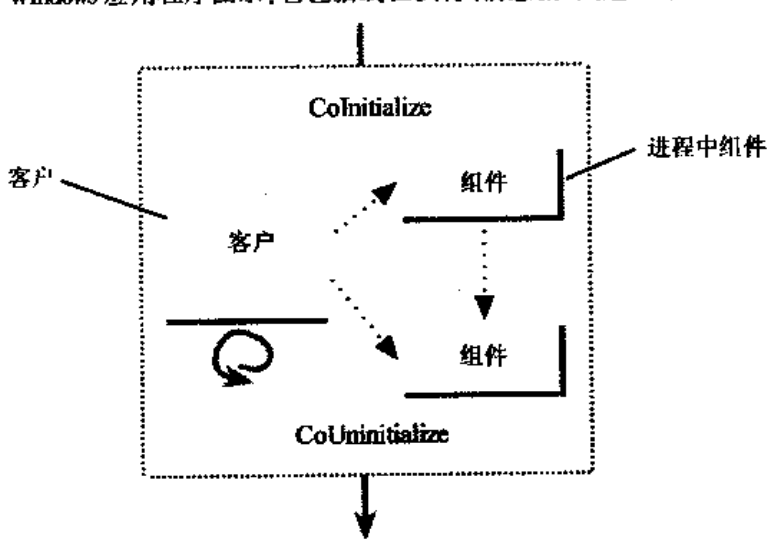


图 12-2 一个客户和两个进程中组件的图示。此处只有一个线程，而组件将共享客户的消息循环

图 12-2 表示的是一个由一个客户和两个线程中组件构成的典型 COM 应用程序。这个程序将驻留在只有一个执行线程的进程中。进程中组件没有自己的消息循环,而是共享了其 EXE 客户的消息循环。同样,图 12-2 也表示了一个套间。

使用进程中 COM 组件不会改变 Windows 应用程序的基本结构。此种程序与没有使用任何 COM 组件的程序相比,最大的区别在于程序在使用任何 COM 库服务之前必须先调用 CoInitialize,而在退出之前调用 CoUninitialize。

一、加入进程外组件

当客户连接到进程外组件时,情况会发生一些变化。图 12-3 表示的是一个客户连接到进程外组件时的情形。在此种情况下组件同客户在不同的进程中运行。每个进程都有它自己的执行线程。进程外服务器有它自己的消息循环。在第 10 章的 OUTPROC.CPP 文件中,我们可以看到此消息循环。同使用进程中组件进程的另外一个重要的差别在于跨越进程边界的函数调用的参数必须被调整。图 12-3 中的"闪电"表示的就是一个被调整的调用。在第 10 章中我们介绍了如何建立一个代理/残根 DLL,它们的作用就是对客户传给进程外组件的数据进行调整。

图 12-3 包含两个套间。其中一个是客户,而组件则在另外一个套间中。这样看来,套间同进程似乎就是同一件东西了,但实际上并不是这样。因为一个进程可以具有多个套间。

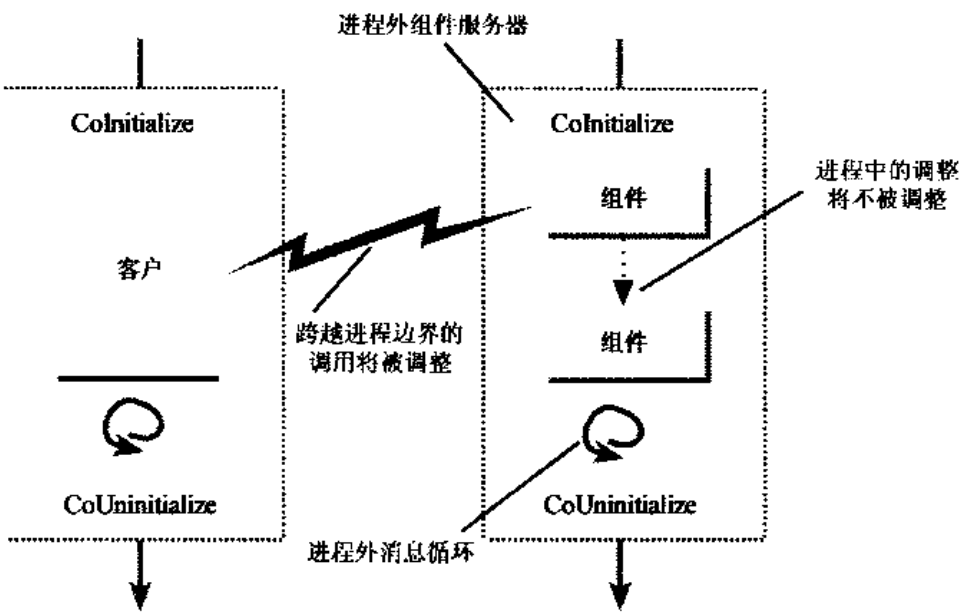


图 12-3 进程外组件具有自己的消息循环和执行线程

在图 12-4 中,我们将图 12-3 中的进程外组件修改为一个在另外一个套间的进程中组件。此处长虚线框表示的是不同的套间,而点线仍然表示进程边界。

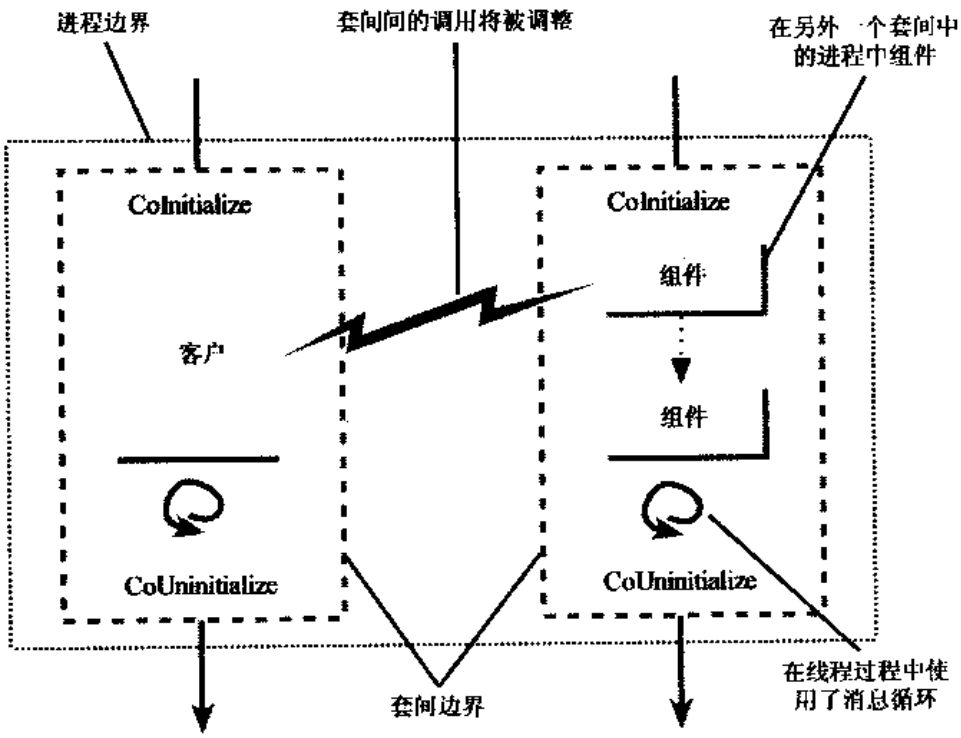


图 12-4 一个客户与一个进程中组件在不同套间通信

可以注意到这两个图是非常相似的。实际上作者在做这幅图时是在原图的外面加上了一个框,然后就宣称所有的对象现在是在同一进程中。这样做的目的是为了说明,套间同单线程进程在如下方面是类似的:每一个进程都有它自己的消息循环。一个套间也可以有它自己的消息循环。同一(单线程)进程中的函数调用和同一套间中的函数调用将不被调整。由于进程和套间只有一个执行线程,因此消息的同步是有保证的。跨越进程或套间边界的函数调用的同步将通过消息循环而得以实现。另外,每个进程都必须初始化 COM 库。与此类似,每一个套间也必须初始化 COM 库。回过头再看一下图 12-2,可以发现客户和它所用的两个组件是共存于同一个套间中的。

12.1.4 套间线程

一个套间线程指的是套间中仅有的唯一线程。对于套间线程,可以将其想象成一个用户界面线程。我们知道用户界面线程拥有它所创建的窗口。一个套间线程同它创建的组件之间的关系也是如此:套间线程将拥有它所创建的组件。一个套间中的组件只能由相应的套间线程调用。

若某个线程给另外一个线程所拥有的窗口发送一条消息,Windows 将把此消息放到此窗口的消息队列中。窗口的消息循环将在创建此窗口的线程上运行。当消息循环取出一条消息并调用窗口过程时,此窗口过程也将在创建相应窗口的线程上运行。

对于一个套间中的组件，情形也是如此。假定另外一个线程调用一个套间中某个组件的方法，COM将把此调用请求放到套间的队列中。消息循环将取出此调用请求并在套间线程上执行相应的方法。

这样就可以保证某个套间中的组件将只在相应的套间线程中被调用，因而组件不需要考虑线程同步的问题。由于COM将保证所有对组件的调用均被同步，因此组件无需是线程安全的。这样组件的编写将会容易得多。实际上本书中给出的所有组件都不是线程安全的。但只要它们是由套间线程创建的，我们将可以确信它们不会被不同的线程同时访问。

这就是自由线程同套间线程之间的区别。

12.1.5 自由线程

COM将在套间线程上同步对组件的调用。对于由自由线程创建的组件的调用，COM不能使之同步。若某个组件是由自由线程创建的，则任意线程均可在任意时候调用它。此时组件开发者应保证对组件访问的同步。也就是说，此时组件应是线程安全的。实际上使用自由线程，同步的工作将由COM移至组件本身。

此时由于COM并不同步对组件的调用，因此自由线程不需要消息循环。由自由线程创建的组件将是一个可以供任意线程访问的组件。此时创建此组件的线程并不拥有此组件，此组件将被所有的线程共享，并可供所的线程自由访问。

在Microsoft Windows NT 3.51和Windows 95中，只有套间线程能够同COM一块使用。在Windows NT 4.0和安装有DCOM的Windows 95系统中，则可以使用自由线程。

上面就是对自由线程的一个概貌性介绍。在讨论调整与同步时，我们还将看到一些有趣的现象。

12.1.6 调整与同步

为能够正确地调整和同步对COM组件的调用，COM需要知道组件所运行的线程的类型。一般而言，在使用套间线程的情况下，所有的调整与同步处理将由COM完成。而在自由线程的情况下，调整处理可能是不需要的，而同步则需要组件自己完成。

下面是一些需要记住的一般性规则：

- 进程间的调用将被调整。这一点我们在第10章中已经讨论过。
- 同一线程中的调用将不被调整。
- 对于套间线程中组件的调用将被调整。
- 对于自由线程中组件的调用并不是总被调整。
- 对于套间线程的调用将被同步。
- 对于自由线程的调用将不被同步。
- 同一线程中的调用将由此线程本身完成同步。

下面我们将讨论套间线程与自由线程之间调用的一些组合情况。我们将从最简单的情况开始。除非特别指明，所有线程将在同一进程中运行。

一、同一线程中的调用

若任一线程中的客户调用同一线程中的一个组件，由于此种调用是在同一线程中，此调用将被自然地同步。COM 不需要作任何同步工作，而组件也不需要是线程安全的。另外此种调用也不需要被调整。这一点我们在本书中一直在使用着。

二、套间线程对套间线程的调用

若某个套间线程中的客户调用另外一个套间线程中的组件，COM 将对此调用进行同步。虽然此时这两个线程是处在同一进程中，但 COM 也将对接口进行调整。在某些情况下，需要程序员自己完成对套间线程间接口的调整。关于这一点我们将在讨论套间线程的实现时讨论。对套间线程中组件的调用与进程外组件的调用是相似的。

三、自由线程对自由线程的调用

当自由线程中的客户调用另外一个自由线程中的组件时，COM 不会对此调用进行同步。此调用将在客户线程中被执行。此时组件需要同步对它的访问，因不同线程中的客户可能会同时对它进行访问。当两个线程在同一进程中时，调用将不被调整。

四、自由线程对套间线程的调用

当自由线程中的客户调用另外一个套间线程中的组件时，COM 将对此调用进行同步。此时组件将在套间线程中被调用。而且不论此时这两个线程是否在同一进程中，都需要对接口进行调整。在大多数情况下，此种调整将由 COM 完成。但在某些情况下，仍需程序员完成此种调整。这一点我们下面将会看到。

五、套间线程对自由线程的调用

当套间线程中的客户调用一个自由线程时，COM 不会对此调用进行同步。同步工作将由组件完成。接口将被进行调整。但若两个线程在同一进程中，COM 将会对调整进行优化，以将指针直接传给客户。这一点我们在讨论自由线程的实现时还将详述。

从上面的讨论我们可以看到，COM 组件的线程与 Win32 线程从概念上讲并没有什么太大的差别。一个进程可以具有任意数目的线程，这些线程可以是套间线程，也可以是自由线程。从程序员的角度看来，COM 线程只有两个令人感兴趣的方面，即同步和调整。COM 将同步对套间线程中组件的调用。而对自由线程中组件的同步则需由开发者自己完成。这种同步处理并不是特定于 COM 的，而是一个一般性的多线程编程问题。但参数调整则是 COM 所特有的从多个线程中处理 COM 组件的独特领域。关于如何自己完成调

整处理，我们将在下面实现套间线程与客户线程时讨论。

12.2 套间线程的实现

使用套间中的组件的最大的好处在于这些组件无需是线程安全的，COM将同步对套间中组件的访问，这些访问可以是来自套间线程的，也可以是来自自由线程的。从内部实现上讲，COM将使用一个隐藏的Windows消息队列来同步这种调用。这一点使得为单线程套间编写组件同编写窗口过程是极为类似的（消息循环的作用是同步对窗口过程的访问，COM将使用同样的机制来同步对单线程套间的访问）。下面所列是对套间的一些关键性的需求：

- 必须调用CoInitialize或OleInitialize。
- 只能有一个线程。
- 必须有一个消息循环。
- 在将接口指针传给其他套间时，必须对之进行调整。
- 若套间是一个进程中组件，则其DLL入口点必须是线程安全的。
- 可能需要一个类型安全的类厂。

下面我们将对其中的某些需求进行更为详细的解释。

组件只能运行于一个线程中。单线程套间中的组件只能运行于一个线程中。此种组件只能供创建它的线程访问。这与窗口过程的工作方式是类似的——窗口过程只能被创建相应窗口的线程访问。由于组件只能被一个线程访问，它将在一个单线程环境中满足各种实际的需求，并需考虑同步的问题。但组件仍应保护其全局数据，因为同窗口过程一样，它是可重入的。

在跨越套间边界时必须对接口进行调整。从其他线程发来的调用请求必须被调整，以使它们像是从组件所在线程中发出的那样。由于一个套间只有一个线程，因此所有其他线程都在此套间之外。跨越套间边界的调用必须被调整。关于接口指针的调整，我们将在稍后讨论。

DLL入口点必须是类型安全的。单线程套间中的组件无需是线程安全的，因为它们只会被创建它们的线程所调用。但进程中服务器DLL中的入口点，如DllGetClassObject和DllCanUnloadNow则必须是线程安全的。这是因为多个不同线程中的多个客户可能会同时调用它们。为使这些函数是线程安全的，需要同步对共享数据的访问。在某些情况下，这意味着类厂也必须是线程安全的。

类厂可能需要是线程安全的。若为所建立的每一个组件都创建一个不同的类厂，那么这些类厂无需是线程安全的，这是因为同一时刻将只会有一个客户来访问它们。但若DllGetClassObject只创建一个类厂并用它来创建所有的组件实例，则将需要保证此类厂是线程安全的，因可能会有多个线程同时访问此类厂。

进程外组件可能会合用类厂的一个实例来创建所有组件实例。此种类厂也应是线程安全的。对于大多数类厂而言，使之成为线程安全的并不是什么困难的事情，因除了引用计数外，它们不会修改其他共享数据。为此，可以使用 InterlockIncrement 和 InterlockDecrement 来同步对引用计数的访问。关于这一点我们在第 4 章中已经介绍过。

在满足这些需求的进程中组件将在注册表中指示它可以支持套间线程。关于组件如何登记其线程模型，我们将在本章末的“12.4 关于线程模型的注册表关键字”一节中讨论。此处我们要详细介绍的是在将一个接口指针传给另外一个线程时需要如何对之进行调整。

当一个套间线程中的组件将其接口指针传递给另外一个线程中的客户时，此接口指针必须调整，不论接收线程是套间线程还是自由线程。

12.2.1 自动调整

在许多情况下，COM 将自动为程序员进行接口的调整。第 10 章中讨论过的代码和残根 DLL 就将对在进程之间传递的接口进行调整。从程序员的角度来看，线程并没有改变这些代理/残根 DLL 的作用。它们将自动完成跨越进程边界的调整。

COM 也使用这些代理/残根 DLL 来完成套间线程和同一进程中其他线程之间的接口调整。所有当访问另外某个套间中组件的接口时，COM 将自动地通过此代理来进行此种调用，这样接口将被相应地进行调整。

12.2.2 手工调整

那么什么情况需要程序员自己对接口指针进行调整呢？答案是在跨越套间边界但并没有通过 COM 进行通信时。

为说明这一点，我们先来看两个例子。第一个例子是，客户创建一个套间线程，在此线程中将创建一个组件并对之进行相应的处理。主线程也可会访问此组件。套间线程将拥有一个此组件的接口指针，因此组件是它创建的。但主线程并不能直接使用此接口指针，因它所处的套间与创建组件的套间是不同的。主线程如果要使用此组件，套间线程必须对相应的接口指针进行调整并将其传给主线程。而在主线程使用此接口指针之前，它必须对之进行反调整。

第二个例子是进程中组件的类厂在其他的线程中创建此组件的实例。这种情况同上面所述的例子基本类似，不同之处在于，在前一例子中，线程是由客户创建的，而在此例中，组件将在不同的线程中创建它自身。客户将调用 CoCreateInstance，这最终将导致组件的类厂被创建。当客户调用 IClassFactory::CreateInstance 时，类厂将创建一个新的套间线程。此新线程将创建相应的组件。IClassFactory::CreateInstance 必须给客户传回一个接口指针。但 CreateInstance 并不能将在新的套间中所创建的接口指针直接传回给客户，因为客户是在另外一个线程中。所以套间线程必须对传给 CreateInstance 的接口指针进行调

整,然后 CreateInstance 将对此指针进行反调整并将其传回给客户。

一、最长的 Win32 API 名称

当弄清楚了何时需要对接口进行调整之后,下面我们来看一下如何对之进行调整。程序员可以使用 CoMarshalInterface 及 CoUnMarshalInterface 来完成这些处理。但更好的方法是使用帮助函数 CoMarshalInterThreadInterfaceInStream 及 CoGetInterfaceAndReleaseStream。这两个函数的名称可以算得上是 Win32 API 中名称最长的了。(按照此种速度,我们将得到一个段落而不是一个函数名称了。)

如何使用这些函数是非常简单的。例如为对 IX 接口指针进行调整,可以使用下面的代码:

```
IStream* pIStream = NULL;
HRESULT hr =
CoMarshalInterThreadInterfaceInStream(
    IID_IX,      // IID of interface to marshal
    pIX,         // Interface to marshal
    &pIStream    // Stream to put marshaled interface
);
```

为对上面的接口进行反调整,可以使用下面的代码:

```
IX* pIXmarshaled;
HRESULT hr =
CoGetInterfaceAndReleaseStream(
    pIStream,        // Stream containing interface
    IID_IX,          // IID of interface to unmarshal
    (void**)&pIXmarshaled // Unmarshaled interface pointer
);
```

这样看来对接口的调整和反调整将是相当简单的。这主要是由于 COM 将自动地在幕后为我们使用代理/残根 DLL 的结果。

12.2.3 编码

到此为止,我们已经介绍了不少概念性的东西了。之所以这样安排,是因为概念性的东西比实现要麻烦得多。下面我们将介绍一个简单的实现例子。假定我们需要在后台增大对某个组件的一个计数器,并偶尔需要对显示进行刷新。如果用常规的 Win32 程序来解决这个问题,则可以创建一个工作线程在后台完成计数处理。

此处我们将用一个套间线程而不是工作线程来完成这件事。主线程将创建此套间线程。在套间线程中将创建一个组件并周期性地修改此组件中的计数器。套间线程将给主线程传递一个接口指针,以便主线程可以获得组件的当前计数值并将其显示出来。整个过程同常规的 Win32 多线程编程是极为类似的,只不过套间线程将:

- 初始化 COM 库
- 具有一个消息循环
- 对传回给主线程的接口进行调整

此处用到的组件与前面我们编写的组件是相同的。

现在编写一个单线程套间最困难之处可能在于套间只是一个概念，而不是一段代码。我们可以像通常那样创建一个线程和一个消息循环。为使套间这个概念更为具体，作者定义了一个名为 CSimpleApartment 来完成这些工作。

一、CSimpleApartment 和 CClientApartment

CSimpleApartment 是一个简单的类，它可以把在另外一个线程中创建组件的过程封装起来。其中的成员 CSimpleApartment::StartThread 将创建并启动一个新的线程；而 CSimpleApartment::CreateComponent 将接收一个被创建组件的 CLSID 值并在由 StartThread 启动的线程上创建它。

这正是令人感兴趣的地方，同时也可能是令人容易混淆的地方：CSimpleApartment 将跨越两个不同的线程。它的一部分将由原线程调用，而另外一部分则由新的线程调用。这样一来，CSimpleApartment 将给原线程与新线程之间的通信提供一种新的方法。由于 CSimpleApartment::CreateComponent 将由原线程调用，因此它将不能直接创建新组件。新的组件必须在新的线程中创建。所有 CreateComponent 将使用一个事件以指示新套间线程来创建新的组件。套间线程将调用 CreateComponentOnThread 来完成组件的创建。CSimpleApartment::CreateComponentOnThread 是一个纯虚函数，派生类必须定义之。在第一个例子中，派生类 CClientApartment 将实现 CreateComponentOnThread。它将使用 CoCreateInstance 来完成组件的创建。

12.2.4 对套间线程例子的说明

表 12-1 套间线程例子中的代码流程

主线程		套间线程
WinMain	CSimpleApartment	CSimpleApartment
InitializeApartment	StartThread	RealThreadProc
		ClassThreadProc
		CClientApartment::WorkerFunction
	CreateComponent	CreateComponentOnThread()
		CClientApartment::
		CreateComponentOnThread

表12-1列出了我们将要介绍的代码的大致结构。表中右部的所有代码将在CSimpleApartment::StartThread创建的套间线程中运行。

一、CSimpleApartment::StartThread

```
BOOL CSimpleApartment::StartThread(DWORD WaitTime)
{
  if (IsThreadStarted())
  {
    return FALSE ;
  }
  // Create the thread.
  m_hThread = ::CreateThread(NULL, // Default security
              0,  // Default stack size
              RealThreadProc,
              (void*)this,
              CREATE_SUSPENDED,
              &m_ThreadId) ;
  if (m_hThread == NULL)
  {
    trace("StartThread failed to create thread"., GetLastError()) ;
    return FALSE ;
  }
  trace("StartThread successfully created thread.") ;
  // Create an event to signal the thread to create the component.
  m_hCreateComponentEvent = ::CreateEvent(NULL, FALSE, FALSE, NULL) ;
  if (m_hCreateComponentEvent == NULL)
  {
    return FALSE ;
  }
  // Create an event for the thread to signal when it is finished.
  m_hComponentReadyEvent = ::CreateEvent(NULL, FALSE, FALSE, NULL) ;
  if (m_hComponentReadyEvent == NULL)
  {
    return FALSE ;
  }
  trace("StartThread successfully created the events.") ;
  // Initialize the wait time.
  m_WaitTime = WaitTime ;
  // Thread was created suspended; start the thread.
  DWORD r = ResumeThread(m_hThread) ;
  assert(r != 0xffffffff) ;
  // Wait for the thread to start up before we continue.
```

```
    WaitWithMessageLoop(m_hComponentReadyEvent);
    return TRUE;
}
```

从上面的代码中我们可以看到，CSimpleApartment::StartThread 将使用::CreateThread 来创建一个新的线程。同时它将创建两个事件对新创建的两个线程进行同步。在套间线程中运行的函数 CSimpleApartment::ClassThreadProc 将两次使用 m_hComponentReadyEvent。第一次用以指示新线程已经启动，后一次则指示此线程已经结束了。函数 CSimpleApartment::CreateComponent 将使用事件 m_hCreateComponentEvent 来指示它何时需要套间线程调用 CSimpleApartment::CreateComponentOnThread 来创建相应的组件。在创建了组件之后，CreateComponentOnThread 将设置 m_hComponentReadyEvent 以告诉 CreateComponent 它已经完成了组件的创建。

CSimpleApartment::WaitWithMessageLoop 是一个帮助函数，其作用是等待某个事件的发生。但其作用并不止于此，它还将处理 Windows 消息。这是因为若在等待某个事件的过程中不处理 Windows 消息，则用户界面将被挂起。因此用户界面在等待过程中都应对消息进行处理。WaitWithMessageLoop 将使用 Win32 API MsgWaitForMultipleObjects。关于此函数我们将在下面讨论。

二、CSimpleApartment::ClassThreadProc

当线程启动时，它将调用静态函数 RealThreadProc，而此函数将调用 ClassThreadProc。由于 Windows 不能调用 C++ 函数，因此所有的 Win32 回调函数都必须为静态函数。当创建线程过程时，可以传递一个指向 CSimpleApartment 的指针以便它能够调用 ClassThreadProc。ClassThreadProc 的实现代码如下。

```
DWORD CSimpleApartment::ClassThreadProc()
{
  // Initialize the COM Library.
  HRESULT hr = CoInitialize(NULL);
  if (SUCCEEDED(hr))
  {
    // Signal that we are starting.
    SetEvent(m_hComponentReadyEvent);
    // Wait for the signal to create a component.
    BOOL bContinue = TRUE;
    while (bContinue)
    {
      switch(::MsgWaitForMultipleObjects(
          1,
          &m_hCreateComponentEvent,
```

```
            FALSE,
            m_WaitTime,
            QS_ALLINPUT))
        {
        // Create the component.
        case WAIT_OBJECT_0:
          CreateComponentOnThread();
          break;
        // Process Windows messages.
        case (WAIT_OBJECT_0 + 1):
          MSG msg;
          while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
          {
            if (msg.message == WM_QUIT)
            {
              bContinue = FALSE;
              break;
            }
            DispatchMessage(&msg);
          }
          break;
        // Do background processing.
        case WAIT_TIMEOUT:
          WorkerFunction();
          break;
        default:
          trace("Wait failed.", GetLastError());
        }
      }
      // Uninitialize the COM Library.
      CoUninitialize();
    }
    // Signal that we have finished.
    SetEvent(m_hComponentReadyEvent);
    return 0;
  }
```

套间必须初始化 COM 库并可以具有消息循环。ClassThreadProc 将满足这些需求。在它的实现中，不是简单地使用了一个 GetMessage/DispatchMessage 循环，而是使用了 MsgWaitForMultipleObjects。MsgWaitForMultipleObjects 将等待三件事情的发生：m_hCreateComponentEvent，一个 Windows 消息和一个超时事件。当事件 m_hCrateComponentEvent 被设置时，MsgWaitForMultipleObjects 将停止等待，然后 ClassThreadProc 将调用 CreateComponentOnThread。当向此线程发送一个 Windows 消息时，ClassThreadProc 将使用一个 Peek

Message/DispatchMessage 循环来取出此消息(即队列中的其他消息)并将其分发出去。当等待超时时,ClassThreadProc 将调用 CSimpleApartment::WorkerFunction。此函数是由 CSimpleApartment 的派生类实现的。关于它我们将在下面讨论。

当然我们也可以使用 GetMessage/DispatchMessage 来完成上述处理。此时将不再使用一个事件来创建组件,而可以使用 PostThreadMessage。但 MsgWaitForMultipleObjects 比 GetMessage/DispatchMessage 的效率高得多。

三、CSimpleApartment::Createcomponent

在创建了线程之后,即可创建相应的组件了。这一点通过主线程调用 CSimpleApartment::CrateComponent 而开始的。Createcomponent 的代码如下:

```
HRESULT CSimpleApartment::CreateComponent(const CLSID& clsid,
                const IID& iid,
                IUnknown * * ppI)
{
  // Initialize the shared data.
  m_pIStream = NULL;
  m_piid = &iid;
  m_pclsid = &clsid;

  // Signal the thread to create a component.
  SetEvent(m_hCreateComponentEvent);

  // Wait for the component to be created.
  trace("Wait for the component to be created.");
  if (WaitWithMessageLoop(m_hComponentReadyEvent))
  {
    trace("The wait succeeded.");

    if (FAILED(m_hr)) // Did GetClassFactory fail?
    {
      return m_hr;
    }

    if (m_pIStream == NULL) // Did the marshaling fail?
    {
      return E_FAIL;
    }

    trace("Unmarshal the interface pointer.");
    // Unmarshal the interface.
    HRESULT hr = ::CoGetInterfaceAndReleaseStream(m_pIStream,
              iid,
              (void * *)ppI);
    m_pIStream = NULL;
    if (FAILED(hr))
```

```
    {
      trace("CoGetInterfaceAndReleaseStream failed.", hr);
      return E_FAIL;
    }
    return S_OK;
  }
  trace("What happened here?");
  return E_FAIL;
}
```

CreateComponent 函数将完成四个主要操作。首先它将把传给它的参数复制到它的成员变量中。其次它指示线程创建 相应的组件。第三,它将等待组件被创建生成。最后它将对所请求的指向组件的接口指针进行反调整。

四、CSimpleApartment::CrateComponentOnThread

当 Createcomponent 设置 m_hCreatecomponentEvent 时,ClassThreadProc 将调用 CreateComponentOnThread 的内部私有版本。后者将完成两个主要操作。首先它将调用 CreateComponentOnThread 的纯虚版本,并将传递给 CreateComponent 的参数传递给它。将参数直接传递给 CreateComponentOnThread 可以简化派生类中函数的实现。其次 CreateComponentOnThread 将对接口进行调整:

```
void CSimpleApartment::CreateComponentOnThread()
{
  IUnknown* pI = NULL;
  // Call the derived class to actually create the component.
  m_hr = CreateComponentOnThread(*m_pclsid, *m_piid, &pI);
  if (SUCCEEDED(m_hr))
  {
    trace("Successfully created component.");
    // Marshal the interface pointer to the server.
    HRESULT hr = ::CoMarshalInterThreadInterfaceInStream(
            *m_piid,
            pI,
            &m_pIStream);
    assert(SUCCEEDED(hr));

    // Release the pI Pointer.
    pI->Release();
  }

  else
  {
    trace("CreateComponentOnThread failed.", m_hr);
  }
```

```
    trace("Signal the main thread that the component is ready.");
    SetEvent(m_hComponentReadyEvent);
  }
```

可以看到对传递给其他线程的接口指针的调整是通过 CoMarshalInterThreadInterfaceInStream 完成的。CreateComponent 将对此接口指针对行反调整。

五、CClientApartment

在第一个例子中，CClientApartment 将实现两个虚拟函数 CreateComponentOnThread 和 WorkerFunction。CClinetApartment 将供希望在另外一个线程中创建组件的客户使用。它将重载 CreateComponentOnThread 以调用 CoCreateInstance。

```
HRESULT CClientApartment::CreateComponentOnThread(
        const CLSID& clsid,
        const IID& iid,
        IUnknown** ppI)
{
  HRESULT hr =
    ::CoCreateInstance(clsid, NULL,
            CLSCTX_INPROC_SERVER,
            iid, (void**)ppI);
  if (SUCCEEDED(hr))
  {
    // Query for the IX interface to use in WorkerFunction.
    hr = (*ppI)->QueryInterface(IID_IX, (void**)&m_pIX);
    if (FAILED(hr))
    {
      // If we can't use it, don't let anybody use it.
      (*ppI)->Release();
      return E_FAIL;
    }
  }

  return hr;
}
```

ClientApartment::CreateComponentOnThread 将查询它创建的组件的 IX 接口以便能够对 WorkerFunction 进行处理：

```
void CClientApartment::WorkerFunction()
{
    m_pIX->Tick();
}
```

六、CLIENT.CPP

至此我们已经介绍了如何创建线程和组件。在每过 CSimpleApartment::m_WaitTime 时间之后，CSimpleApartment::ClassThreadProc 均将调用 CClientApartment::WorkerFunction。所以每隔一段时间组件均将被刷新。为在窗口中显示出这些变化，客户创建了一个定时器。当它接收到一个 WM_TIMER 消息时，它将调用 OnTick。后者将调用 IX::GetCurrentCount 并将此计数值显示在窗口中。当客户调用 IX::GetCurrentCount 时，此调用将被调整，以便能够正确地跨越套间边界。当 WorkerFunction 调用 IX::Tick 时，由于此调用是在同一个套间线程中，因此这个调用将不被调整。

客户并不是可以创建套间线程的唯一对象。我们也可以编写出能够创建套间线程的组件。实际上我们还可以编写出能够在另外的套间线程中创建组件的类厂。

上面介绍的就是实现套间线程的所有内容。我们看到，实现过程中最困难的部分是对线程的处理。在熟悉了套间线程之后，下面我们将介绍自由线程。

12.3 自由线程的实现

若读者经常编写多线程程序，那么自由线程的实现并不会带来什么新的困难。我们可以使用常规的 Win32 线程函数，如 CreateThread、ResumeThread、WaitForMultipleObjects、WaitForSingleObject、CreateMutex 以及 CreateEvent 来完成自由线程的创建和管理。使用标准的线程对象，如互斥量、临界区以及信号量等，可以同步对组件内部数据的访问，以使组件成为是线程安全的。虽然保证一个组件是 100%线程安全的决不是一件简单的事情，但一个定义良好的 COM 接口将使得它在被访问时其类型安全性能够明显地表示出来。

若读者刚开始学习编写多线程程序，本节中的示例程序将提供一个良好的开端。本例中我们将使用一对互斥量来防止两个线程同时访问同一数据。

对于自由线程，除了要保证组件是线程安全的外，还有三个需求。首先一点是操作系统必须支持 COM 自由线程。Windows NT 4.0 就是这样一种操作系统。在安装了 DCOM 扩展之后，Windows 95 也将支持 COM 自由线程。在第 10 章中我们已经讨论过如何通过编程的方法来决定操作系统是否支持自由线程。（基本上是通过 OLE32.DLL 中的函数 CoInitializeEx 函数。）

上面提到 CoInitializeEx，实际上实现自由线程的第二个需求就是线程要用参数 COINIT_MULTITHREADED 来调用 CoInitializeEx 以将其标识为可供所有自由线程访问的。将一个线程标识为可供所有自由线程访问的是什么含义呢？创建组件的线程将决定一个组件如何处理来自其他线程的调用。对于自由线程创建的组件，任何其他自由线程均可以在任何时候调用它。

当一个自由线程用 COINIT_MULTITHREADED 调用了函数 CoInitializeEx 之后，它将不

能再用另外一个参数来调用它。于是 OleInitialize 将用参数 COINIT _ APARTMENT-THREADED 来调用 CoInitializeEx,所以从一个自由线程中不能使用 OLE 库。

第三个需求并不是针对自由线程的,而是针对套间线程的。当将接口指针传递给套间线程时,它们必须被相应地调整。但这一点只适用于不是通过 COM 接口来传递接口指针的情况。在通过 COM 接口来传递接口指针时,COM 将为程序员完成此种调整。若客户在另外一个进程中,COM 也将为程序员完成接口的调整。当然,要使 COM 能够完成此种调整,需要给它提供一个代理/残根 DLL。在前一节中我们讨论了套间线程之间的调整处理,对于自由线程进行此种调整,也可以使用同样的函数,即 CoMarshalInterThreadInterfaceInStream 和 CoGetInterfaceAndReleaseStream 来处理接口的调整问题。COM 可以在幕后对此种调整进行优化。关于这一点后面我们将会介绍。

对于进程中组件,还有第四个需求。即它们必须在注册表中将它们登记为可供所有自由线程访问。关于这一点请参阅"12.4 关于线程模型的注册表关键字"。

这里我们看到,除了需要对传递给其他套间的接口指针进行调整外,对于自由线程的其他需求是非常直接的。自由线程带来的一个最大的负担是组件此时必须是线程安全的。但这一点并不是 COM 的要求而是多线程编程中一个标准问题。

12.3.1 对自由线程例子的说明

创建自由线程的代码与创建套间线程的代码没有什么太多的差别。在本节的例子中我们将创建两个自由线程。其中一个自由线程将增大组件的引用计数(如同前面的套间线程中的那样)。另外一个自由线程将减小此引用计数。另外我们现在可以将引用计数值想象为一个物体在左右两只手中的情形。第一个自由线程可以把此物体移至左手中,而另外一个线程则可以将其移到右手中。作为主线程的套间线程将得到一个被调整的接口指针。主线程将用此指针周期性地决定和报告组件的状态。这个例子的大部分代码与前面创建套间线程的代码是相同的。此处我们将不再重复,而只是指明其中的一些差别。

一、明显的差别

最明显的差别是我们将 CSimpleApartment 的名称修改为 CSimpleFree。当创建一个自由线程时,此时创建的不再是一个线程而是一个套间。与此类似,CClientApartment 的名称也将相应地改为 CClientFree。

此处我们要特别指出的是 CSimpleFree 并不是创建和管理自由线程的一个一般性方法。CSimpleFree 本身并不是线程安全的。此处它只能被在一个套间线程中运行的客户用来创建自由线程。当然我们这样设计的目的只是为了使其尽可能的简单。

二、CSimpleFree::ClassThreadProc

在 CSimpleFree 中与 CSimpleApartment 中区别最大的函数是 ClassThreadProc。CSimple-

Free将调用CoInitializeEx(0,CEINT _MULTITHREADED),而不是像CSimpleApartment那样调用此函数。在能够调用CoInitializeEx之前,需要做两件事情。首先必须定义_WINNT32_WINNT为0x0400或定义_WIN32_DCOM。若这两个宏都没有定义,则OBJBASE.H将不会包含CoInitializeEx的定义。其次需要在运行时检查组件运行的操作系统支持CoInitializeEx。所有这些可以用如下的代码表示:

```
DWORD CSimpleFree::ClassThreadProc()
{
  // Check for the existence of CoInitializeEx.
  typedef HRESULT (_stdcall * FPCOMINITIALIZE)(void*, DWORD);
  FPCOMINITIALIZE pCoInitializeEx =
    reinterpret_cast<FPCOMINITIALIZE>(
      ::GetProcAddress(::GetModuleHandle("ole32"),
          "CoInitializeEx"));
  if (pCoInitializeEx == NULL)

  {
    trace("This program requires the free-thread support in DCOM.");
    SetEvent(m_hComponentReadyEvent);
    return 0;
  }

  // Initialize the COM Library.
  HRESULT hr = pCoInitializeEx(0, COINIT_MULTITHREADED);
  if (SUCCEEDED(hr))
  {
    // Signal that we are starting.
    SetEvent(m_hComponentReadyEvent);

    // Set up array of events.
    HANDLE hEventArray[2] = { m_hCreateComponentEvent,
          m_hStopThreadEvent };

    // Wait for the signal to create a component.
    BOOL bContinue = TRUE;
    while (bContinue)
    {
      switch(::WaitForMultipleObjects(2,
          hEventArray,
          FALSE,
          m_WaitTime))
      {
      // Create the component.
      case WAIT_OBJECT_0:
        CreateComponentOnThread();
        break;
```

```
        // Stop the thread.
        case (WAIT_OBJECT_0 + 1):
          bContinue = FALSE;
          break;

        // Do background processing.
        case WAIT_TIMEOUT:
          WorkerFunction();
          break;

        default:
          trace("Wait failed.", GetLastError());
        }
      }
      // Uninitialize the COM Library.
      CoUninitialize();
    }

    // Signal that we have finished.
    SetEvent(m_hComponentReadyEvent);
    return 0;
  }
```

由于 CSimpleFree 创建的是自由线程,因此它不需要消息循环。所以上面我们将 MsgWaitForMultipleObjects 替换为 WaitForMultipleObjects。其中事件 m_hStopThreadEvent 代表的是一个终止线程的 WM_QUIT 消息。

虽然在 ClassThreadProc 中不再需要 MsgWaitForMultipleObjects,但在 CSimpleFree::StartThread 和 CSimpleFree::CreateComponent 中将仍然用到它。这些函数将被主线程调用,所以它们仍然需要对消息进行处理,以使用户界面不致被锁起。

上面所述即为 CSimpleFree 和 CSimpleApartment 的仅有的差别。

三、CClientFree

下面我们将要展示的是两个自由线程如何在不对其接口指针进行调整的情况下共享同一组件。为此,需要在 CClientFree 中加上两个成员函数。本例中的 CClientFree 与前一个例子中的 CClientApartment 是等价的。CClientFree 将继承 CSimpleFree 并实现了 CreateCompontneOnThread 和 WorkerFunction 虚拟函数。CClientFree 的两个新函数分别是 ShareUnmarshaledInterfacePointer 和 UserUnmarshaledInterfacePointer。(作者受到某些 COM 函数的长名称的鼓舞,也为这两个函数分别取了一个长名称。)第一个函数 ShareUnmarshaledInterfacePointer 将返回 CClientFree 在其 WorkerFunction 中所用的 IX 接口指针。此接口未经调整,因此,仅能从一个自由线程中使用它。第二个函数 UserUnmarshaledInterfacePointer 将设置 CClientFree 对象将在其 WorkerFunction 中使用的 IX 接口指针。下面我们来看一下这些函数是如何使用的。

函数 InitializeThread 将被用来创建一个自由线程和一个组件。此函数与单线程套间情况下所用的 InitializeApartment 调用是相似的。在调用了 InitializeThread 之后，客户将调用 InitializeThread2。此函数将建立第二个线程。但在此线程中并没有创建第二个组件，而是共享了第一个线程创建的组件。InitializeThread2 的代码如下：

```
void InitializeThread2()
{
  if (g_pThread == NULL)
  {
    return;
  }

  // Create the second thread.
  // This thread has a different WorkerThread function.
  g_pThread2 = new CClientThread2;

  // Start the thread.
  if (g_pThread2->StartThread())
  {
    trace("Successfully started second thread.");

    // Get the same pointer used by the first thread.
    IX* pIX = NULL;
    pIX = g_pThread->ShareUnmarshaledInterfacePointer();
    assert(pIX != NULL);

    // Use this pointer in the second thread.
    g_pThread2->UseUnmarshaledInterfacePointer(pIX);
    pIX->Release();
  }
}
```

InitializeThread2 将创建一个 CClientFree2 对象而不是一个 CClientFree 对象。CClientFree 将继承 CClientFree。它与 CClientFree 的唯一差别是 WorkerFunction 的实现有所不同。这两个 WorkerFunction 函数分别如下所列。

```
void CClientThread::WorkerFunction()
{
    CSimpleLock Lock(m_hInterfaceMutex);

    m_pIX->Tick(1);
    m_pIX->Left();
}

void CClientFree2::WorkerFunction()
{
    CSimpleLock Lock(m_hInterfaceMutex);

    m_pIX->Tick(-1);
```

```
    m_pIX->Right();
}
```

关于 CSimpleLock 我们将在下面讨论。修改后的 IX::Tick 将接受一个参数并用之增大计数值。同时我们还给它增加了一个 Left 方法和 Right 方法。这两个函数可以控制计数值在哪一只"手"中。CClientFree 将增大计数值并将其放在左手中，而 CClientFree2 则将减少计数值并将其放在右手中。当计数值在右手中时，InRightHand 函数将返回 TRUE。所以 InRightHand 的返回值将可以说明上次使用组件的线程为何。

四、对组件的修改

除了给组件增加新的成员函数外，还应使之成为线程安全的，这是因为我们有两个线程将分别增大和减小组件的计数值。为此我们实现了一个名为 CSimpleLock 的简单类：

```
class CSimpleLock
{
public:
    // Lock
    CSimpleLock(HANDLE hMutex)
    {
        m_hMutex = hMutex;
        WaitForSingleObject(hMutex, INFINITE);
    }
    // Unlock
    ~CSimpleLock()
    {
        ReleaseMutex(m_hMutex);
    }
private:
    HANDLE m_hMutex;
};
```

在使用这个类时，需要给它的构造函数提供一个互斥量的句柄。构造函数将一直等待，直至指定的互斥量成为有信号的。CSimpleLock 的析构函数将在定义此种对象的函数执行完毕之后释放指定的互斥量。有了 CSimpleLock 之后，为对某个函数进行保护，只需创建一个 CSimpleLock 对象即可：

```
HRESULT _stdcall CA::Tick(int delta)
{
    CSimpleLock Lock(m_hCountMutex);
    m_count += delta;
    return S_OK;
}
```

```
HRESULT _stdcall CA::Left()
{
    CSimpleLock Lock(m_hHandMutex);
    m_bRightHand = FALSE;
    return S_OK;
}
```

在上面的组件中我们使用了两个不同的互斥量 m_hHandMutex 和 m_hCountMutex。后者将用于保护对 m_count 的访问，而前者将用于对 m_bRightHand 的访问。使用两个不同的互斥量将使得一个线程在访问 m_count 的同时另外一个线程可以访问 m_bRightHand。这一点正是使用自由线程所具有的而使用套间线程所没有的优点之一。套间线程中的组件只能被一个线程访问：即套间线程。如果组件在另外一个套间线程中，那么当一个线程在调用 Tick 时，另外一个线程将不能调用 Left。但是在使用自由线程时，线程之间的同步将由程序员来完成。此时可以使用组件的内部知识对同步处理进行优化。

12.3.2 自由线程参数调整的优化

参数调整与同步处理都是比较慢的。所以如果可能的话，应尽可能地避免使用它。对于套间线程的一条规则就是程序员需要对传给其他套间线程的接口进行调整。但当一个套间线程中的客户想使用同一进程中的一个自由线程中的组件上某个接口时，由于这两个线程是在同一个进程中，因此我们可能不希望进行接口的调整。同时，由于我们的组件是线程安全的，因此我们也不希望 COM 来同步对于组件的调用。这样看来，自由线程中的组件似乎可以直接将其指针传递给同一进程中的套间线程。后面我们将会看到，这一点实际上是可以实现的。

不仅上述优化是可能的，而且 COM 库中提供了一个特殊的聚合组件可以为程序员完成这件工作。CoCreateFreeThreadedMarshaler 实现的是一个具有 IMarshal 接口的组件，它可以决定接口的客户是否在同一进程中。若客户在同一线程中，在对接口进行调整时，所有的指针都将被原封不动地传递过去。若客户是在另外一个进程中，接口将被标准的调整程序进行调整处理。使用 CoCreateFreeThreadedMarshaler 的优点在于程序员无需关心客户到底在什么地方。此种优化对于 CoMarshalInterThreadInterfaceInStream 和 CoGetInterfaceAndReleaseStream 也是适用的。所以不论在何种情况下均对接口进行调整，然后让 COM来完成对此调整的优化。在下面的代码中，将创建一个自由线程化的调整程序。同时我们还给出了 QueryInterface 的实现，它将把 IMarshal 查询转发给所创建的调整程序。

```
HRESULT CA::Init()
{
    HRESULT hr = CUnknown::Init();
    if (FAILED(hr))
```

```
    {
        return hr ;
    }

    // Create a mutex to protect the count.
    m_hCountMutex = CreateMutex(0, FALSE, 0) ;
    if (m_hCountMutex == NULL)
    {
        return E_FAIL;
    }

    // Create a mutext to protect the hand.
    m_hHandMutex = CreateMutex(0, FALSE, 0) ;
    if (m_hHandMutex == NULL)
    {
        return E_FAIL ;
    }

    // Aggregate the free threaded marshaler.
    hr = ::CoCreateFreeThreadedMarshaler(
            GetOuterUnknown(),
            &m_pIUnknownFreeThreadedMarshaler) ;
    if (FAILED(hr))
    {
        return E_FAIL ;
    }
    return S_OK ;
}

HRESULT __stdcall
CA::NondelegatingQueryInterface(const IID& iid, void** ppv)
{
    if (iid == IID_IX)
    {
        return FinishQI(static_cast<IX*>(this), ppv) ;
    }
    else if (iid == IID_IMarshal)
    {
        return m_pIUnknownFreeThreadedMarshaler->QueryInterface(iid, ppv) ;
    }
    else
    {
        return CUnknown::NondelegatingQueryInterface(iid, ppv) ;
    }
}
```

对术语的说明

本章的开头我们提到过。COM 的线程术语在不同的文档中是各不相同的。Win32 SDK 的作者们所用的"套间"这一词的含义与本章中给出的"套间"的含义是有一些差别的。本章中所说的套间与 Win32 SDK 中的"单线程套间"的含义是相同的。在 Win32 SDK 中还有一个"多线程套间",它指的是所有的自由线程。按照此种定义,一个进程可以具有任意数目的"单线程套间",而只能有一个"多线程套间"。希望此处的说明能够消除读者在阅读 Win32 SDK 文档时可能会产生的混淆。

12.4 关于线程模型的注册表关键字

COM 需要知道进程中组件支持的线程模型以便能在跨越线程边界时对其接口进行合适的调整与同步。为登记进程中组件的线程模型,可以给组件的 InprocServer32 关键字加上一个名为 ThreadingModel 的名称值(注意不是子关键字)。ThreadingModel 有三个可能的值:Apartment、Free 和 Both。

对于可以在套间线程中使用的组件,很显然应将 ThreadingModel 的值设置为 Apartment。而那些可以在自由线程中使用的组件则应将此值设置成 Free。而既可以在套间线程中使用、也可以在自由线程中使用的组件则应将此值设置成 Both。若组件不关心有关线程的任何问题,它可以不指定任何值。此时 COM 将假定此组件不支持多线程。另外一个进程中服务器提供的所有组件都应具有相同的线程模型。

12.5 本章小结

本章我们不仅介绍了如何实现套间线程和自由线程,同时也介绍了套间的概念。我们知道套间是由一个线程和一个消息循环构成的概念性实体。套间线程则同典型的 Win32 进程有些类似:它们都有一个执行线程和一个消息循环。一个进程可以具有任意数目的套间线程和自由线程。

套间线程必须初始化 COM,必须具有一个消息循环,必须对传递给其他线程的接口指针进行调整。在套间线程中创建的组件必须只能被创建它的线程所调用。这条规则对于窗口过程也是一样的。进程中服务器必须提供具有线程安全的入口点,但它所提供的组件则无需是线程安全的,因 COM 将同步对这些组件的访问。

自由线程必须使用 CoInitializeEx 来初始化 COM。它们无需具有消息循环,但对于传递给套间线程和其他进程的接口指针仍应进行调整。对于传递给同一进程中其他自由线程的接口指针则无需进行调整。自由线程所创建的组件必须是线程安全的,因 COM 对于同一进程中自由线程之间的调用是不进行调整处理的。

这样一来就会产生一个问题:我们应使用何种类型的线程呢?对于用户界面,必须使用套间线程。这可以保证用户输入所产生的消息将总能被及时地处理,从而不会使用户感到程序被锁住。若只需要在后台执行某个简单的操作,此时可以使用套间线程。套间线程的实现比较容易,因程序员无需使其中所用的组件是线程安全的。

所有对套间线程的调用均必须被调整。这可能会引起严重的性能问题。所以如果在不同线程之间进行通信的代码比较多,可以将这些代码均放到同一个线程中或者换用自由线程。同一进程中自由线程之间的调用将不被调整,根据组件对其同步代码的实现不同,其速度也将会有不同程度的提高。

再回过头来看一下,我们将如何实现我们的模拟直升机呢?这个问题就留给读者自己去回答了。

第 13 章　一个完整的例子

一千多年前的中国唐代就出现了一种名为七巧板(Tangram)的玩具。它由七个小块组成,人们可以用它来构成不同的形状(如图 13-1 所示)。作者非常喜欢这种玩具,因为

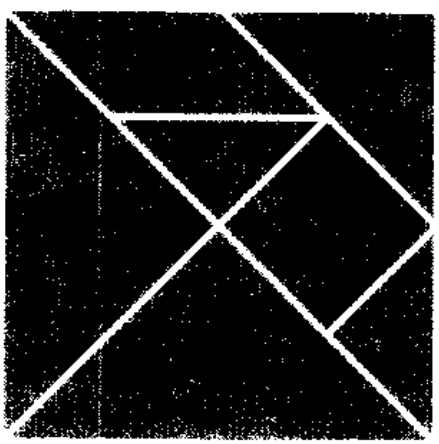


图 13-1　七巧板中的 7 个小块都具有简单的几何形状

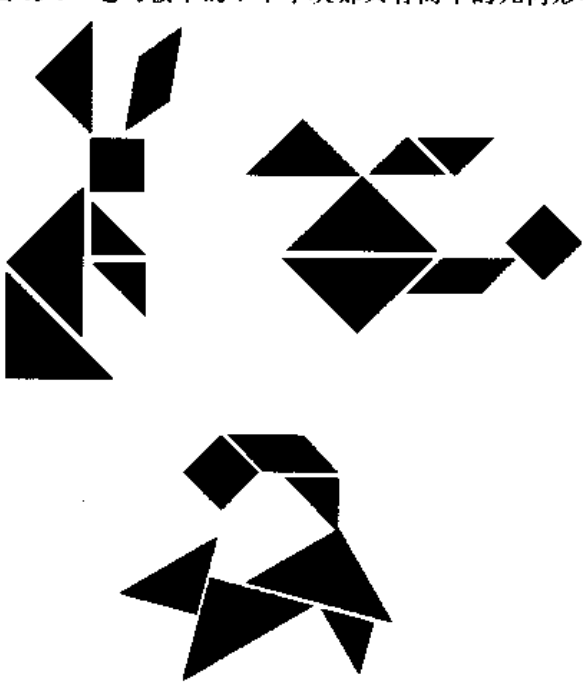


图 13-2　可以用七巧板构成各种类型的图案,图中所示为一只兔子、一架直升机和一只猫

七个小块虽然非常简单，但由此构成的图案可以说不仅数目上没有任何限制，而且其复杂程度也可以非常高。

可以看到七巧板中有五个等边直角三角形：两个大的、两个小的、一个中等大小的。另外还有一个正方形和一个斜长方形。用这七个小块可以构成的形状既可以是一些抽象的几何图案，也可以是人物、动物、树、机器，甚至可以是字母表中的所有字母。图 13-2 给出了用七巧板构成的图案的几个例子。可以看到这些图案不仅可以激发人的想象力，而且都是非常精巧的。例如将代表头部分的小方块稍微转动一下，即可以增大七巧板划艇中划手所承受的拉力。

同七巧板类似，COM 也是非常简单的，但由 COM 构成的应用程序的功能却可以非常强大。作者认为以七巧板作为基础，构造一个示例程序，以将本书中介绍的各种思想集成到一个应用程序中将是再合适不过的。

13.1 Tangram 程序

作者原来的计划是用一个例子来贯穿全书的始终，而不是每章分别给出单独的例子。但审阅者返回来的意见很快显示这种方法是不可行的。COM 只是应用程序是的一个骨架，而应用程序本身才是支撑此骨架的血肉——肌肉和皮肤。正是由于有了血肉，我们才很难看到骨架，但不可否认，正是血肉赋予了骨架以生气，而生物体正是由于有了皮肤的保护和帮助才得以发育和被识别。与之类似，COM 是隐藏在真实的应用程序中的，它将隐藏在使应用程序完成其工作的其他代码的后面。既然本书的目的是为了尽可能地突出 COM，因此作者决定在每一章中分别给出不同的简单的例子。

但作者相信在更为广阔的环境中将有助于发现一些新的思想。因此我们编写了一个在 Microsoft Windows 环境下运行的完整的 COM 程序 Tangram。它展示了本书中介绍的大多数技术，另外它还展示了我们未曾介绍过的某些 OLE、AcitveX 以及 COM 接口。

13.1.1 Tangram 的运行

在运行 Tangram 之前需要登记有关的组件。在启动此程序之后，它将给出一个对话框，以让用户选择如何运行它：

- 在一个列表框中，用户可以选择使用哪个“World”组件以在屏幕上显示七巧板。TangramGdiWorld 组件将绘制七巧板的一个二维视图，而 TangramGLWorld 则将给出一个三维视图。若用户系统中没有安装 OpenGL 库，TangramGdiWorld 选项将是不可用的。
- 在一个复选框中，用户可以选择代表七巧板的“model”组件是以进程中方式运行还是以进程外方式运行。进程外组件也将在本地机器上运行。（读者可以如第 10 章所述那样将其设置成在远程运行。）

在 Tangram 开始运行之后，即可以在屏幕上见到七巧板的七个小块。此时可以使用

鼠标来移动这些小块。右击某个小块时,它将逆时针旋转。若在右击的同时按住 Shift 键,则可以使它顺时针旋转。

13.1.2 所用的组件

Tangram 程序是由几个不同的组件和许多接口构成的。所有特定于 Tangram 的组件和接口前面都分别加有前缀 Tangram 和 ITangram。这些前缀将使我们很容易识别哪些接口是属于 Tangram 的。

为使在 Windows 注册表中查找与 Tangram 相应的接口与组件尽可能简单,所有 Tangram 的 GUID 均具有如下的形式:

b53313xxx-20c4-11d0-9c6c-00a0c90a632c

Tangram 包含的主要组件和接口如表 13-1 所列。

表 13-1 Tangram 应用的主要组件和接口

Components	Interfaces	概要
TangramModel	ITangramModel	包含每一个小块的形状和位置
	ITangramTransform	信息
	IConnectionPointContainer	
TangramGdiVisual	ITangramVisual	绘制一个小块
	ITangramGdiVisual	
	ITangramModelEvents	
TangramGdiWorld	ITangramWorld	对整体绘制进行协调
	ITangramGdiWorld	
	ITangramCanvas	
TangramCanvas	ITangramCanvas	处理显示需求

在上表的名称中带有 Gdi 的组件和接口具有相应的 OpenGL 版本,例如 TangramGdiVisual 的 OpenGL 版本是 TangramGLVisual。组件的 GID 版本给出的是七巧板的一个二维显示,而 OpenGL 版本给出的则是一个三维显示。

在下面各节中我们将对 Tangram 程序中的一些主要组件进行一下说明。图 13-3 给出的是此应用程序的一个简化的体系结构图。

13.1.3 客户程序

客户程序中包含的是将所有的组件组合起来以形成应用程序的代码。它不包含任何接口,而只是一个常规的 Win32 C++ 程序,但它可以使用 MFC 以简化代码的实现。客户程序中包含一系列指向它所能控制的接口的指针,这样它可以通过 ITangramWorld 同 Tangram * World 组件通信,通过 ITangramVisual 与当前选定的直观表示通信,而与 TangramMod-

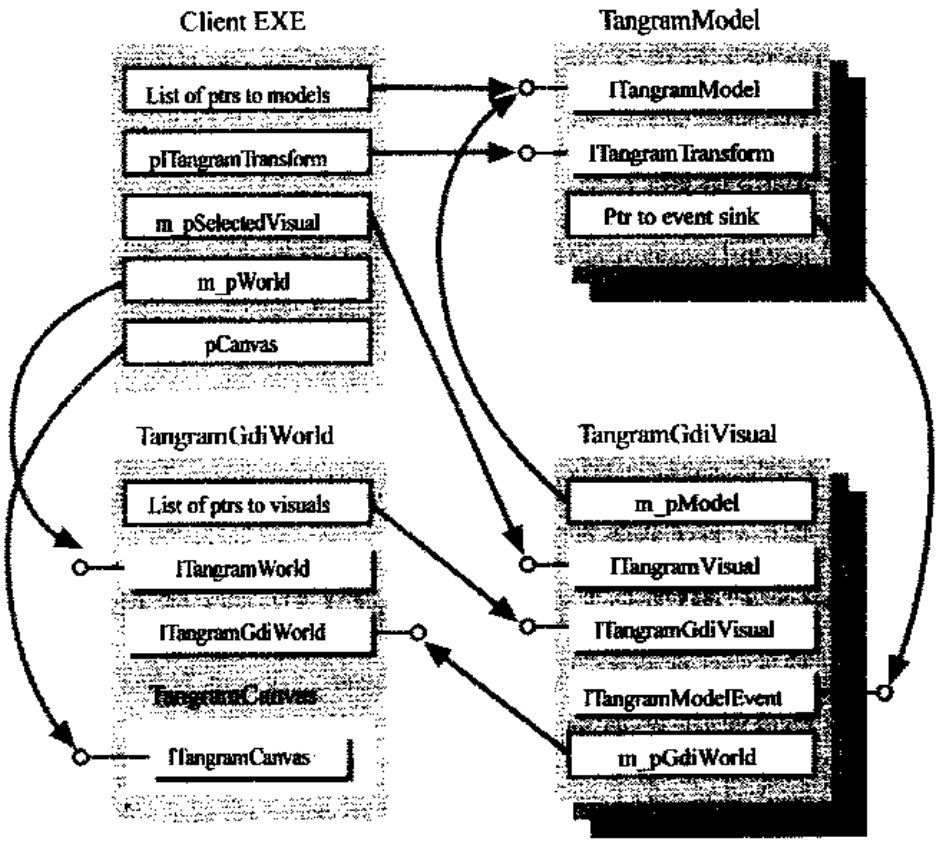


图 13-3 Tangram 体系结构图

el 的通信则是借助于 ITangramModel 和 ITangramTransform 来完成的。

Tangram 程序中包含 TangramModel 组件的七个实例，这七个实例分别表示七个七巧板块。每一个 TangramModel 具有一个相应的 Tangram * Visual。Tangram * Visual 与 TangramModel 的通信是借助 ITangramModel 接口完成的。Tangram * World 组件中包含这七个 Tangram * Visual。Tangram * World 通过 Tangram * Visual 的 ITamgram * visual 接口对之实施有关的控制。Tangram * World 也聚合了 TangramCanvas 以获取 ITangramCanvas 的一个实现，而后者将为客户程序所用。

13.1.4 TangramModel 组件

TangramModel 是 Tangram 程序的基础。一个 TangramModel 表示一个七巧板块的多边形。有时我们也将其称作是“模型”。客户可以通过 ITamgramModel 和 ITamgramTransform 对此组件进行处理。

一、ITamgramModel 接口

ITamgramModel 封装的是一个表示七巧板块的多边形的坐标。Tangram 把所有七巧板

块显示在一个 20×20 的虚拟活动范围内，并只通过此范围内的坐标对板块进行处理。如果将此活动范围内的顶点转化成可以在屏幕上显示的形状将由实现 ITangramWorld 和 ITangramVisual的组件决定。

二、ITangramTransform 接口

客户将使用 ITangramTransform 来移动和旋转一个七巧板块。在 Tangram 中，转换是通过虚拟的活动领域内的坐标完成的，而旋转则是按角度完成的。

三、IConnectionPointContainer 接口

这是一个标准的 COM/AcitveX 接口。关于此接口的详细讨论，我们将在本章中的“13.4 事件和连接点”中给出。这个接口给 TangramModel 通知相应的 Tangram * Visual 组件其位置的改变提供了一种灵活的方法。

13.1.5 TangramGdiVisual 和 TangramGLVisual 组件

每一个 TangramModel 组件都有一个相应的 Tangram * Visual 组件(简称为“图示”)。TangramDgiVisual 组件将使用 Windows GID 来绘制七巧板块的一个二维表示。而 TangramGLVisual 则使用 OpenGL 来给出七巧板的一个三维显示。这些组件是均包含一个指向 TangramModel 的 ITangramModel 接口的指针。使用此接口，Tangram * Visual 组件将可以获取同其相关联的 TangramModel 的顶点信息并将其转换成相应的屏幕坐标。Tangram * Visual 组件实现了三个接口：ITangramVisual、ITangram * Visual 以及 ITangramModelEvent。

一、ITangramVisual 接口

Tangram 将使用 ITangramVisual 接口将一个模型和一个图示关联起来，并用它来获取某个特定的图示。不同的图示将对相关联的七巧板块的显示产生影响。

二、ITangramGdiWorld 和 ITangramGLWorld 接口

TangramGdiWorld 组件使用 ITangramGdiWorld 以在屏幕上显示 TangramModel 的一个二维表示。TangramGLWorld 组件使用 ITangramGLVisual 接口同每一个 TangramGLVisual 进行通信，以绘制出 TangramModel 的一个三维表示。

使用多个接口的好处是可以将客户同绘制操作的具体实现分开。GangramGdiWorld 和 TangramGLWorld 将分别把二维和三维的绘制细节封装起来，并将客户同这些细节完全隔离开。这样客户就可以把各板块当成是在一个平板状的 20×20 的活动范围那样自由地操作它们，而不必须理会它们是如何被 Tangram * World 和 Tangram * Visual 组件实际显示出来的。

虽然客户和 TangramModel 可以忽略七巧板块的显示方法，但 Tangram * World 和 Tan-

gram * Visual 必须注意它们在显示七巧板块的过程中所扮演的角色。Tangram * World 将提供 Tangram * Visual 所能够用的绘制的屏幕。这些组件是作为协作对而一块编写出来的。根据这两个组件的定义,如果没有其中一个组件的代码,想要编写出另外一个组件几乎是不可能的。在此种情况下,我们可以将这两个 COM 类想象成是一个"组件"。

三、ITangramModelEvent

Tangram * Visual 组件需要知道与它相应的 TangramModel 的顶点何时发生了变化。为此,TangramModel 定义了一个名为 ITangramModelEvent 的事件接口。当 顶点发生变化时,TangramModel 将调用函数任意监听有关事件的组件中的 ITangramModelEvent:: OnChanged-Model 函数(在此种情况下,此组件即为相关联的图示组件)。关于事件,我们将在本章后面的"13.4 事件和连接点"一节中讨论。

13.1.6 TangramGdiWorld 和 TangramGLWorld 组件

每一个 Tangram * Visual 组件都包含在相应的 Tangram * world 组件中。Tangram * World 的作用是给 Tangram * Visual 准备一个能够在上面绘制图形的显示设备,同时它还将处理所有的屏幕刷新与调色板问题。Tangram * World 支持三个接口:ITangramWorld、ITangram * world以及 ITangramCanvas。

一、ITangramWorld

客户程序将通过包罗万象的 ITangramWorld 接口对 Tangram * world 组件进行控制。客户程序几乎不会同 Tangram * Visual 组件进行通信,而只是同 Tangram * world 通信并让它同 Tangram * Visual 打交道。

二、ITangramGdiWorld 和 ITangramGLWorld 接口

这些接口被 Tangram * Visual 组件用来同它们各自的 Tangram * world 进行通信。组件中指向客户的后向指针虽然非常有用,但它们可能会导致引用循环,从而使组件的引用计数永远不会减小为 0,最终导致它永远不会被从内存中释放掉。关于这一点我们将在后面的"13.3 循环引用计数"一节中详细讨论。

三、ITangramCanvas 接口

客户程序把它的显示需求转发给 TangramWorld 的 ITangramCanvas 接口,如屏幕的绘制与刷新以及调色板的处理等。虽然 TangramGdiWorld 和 TangramGLWorld 都支持 ITangram-Canvas 接口,但它们都没有实现此接口,而是均聚合了实现此接口的 TangramCanvas 组件。

13.2 展示

前面我们说过,Tangram 展示了本书中介绍的大多数技术。下面我们将简要地说明一下在哪些地方可以找到最有趣的一些技术。

聚合 Tangram * World 聚合了组件 TangramCanvas 以给客户程序提供 ITangramCanvas 的实现。

包容 在 Tangram 中大量使用了包容。从图 13-3 中可以看出 Tangram * World 包容了 Tangram * Visual,而后者又分别包容了一个 TangramModel 组件。

组件类别 Tangram 定义了一个 Tangram World 组件类别。Tangram World 组件类别的成员是实现了 ITangramWorld 和 ITangramCanvas 接口的一个组件。客户程序使用组件类别来查找那些实现了 ITangramWorld 和 ITangramCanvas 的被登记的组件,然后让用户选择待使用的组件。

可互换的组件 COM 的设计目标之一是能用一个组件替换另外一个支持相同接口的其他组件。TangramGLWorld 和 TangramGLVisual 与 TangramGdiWorld 和 TangramGdiVisual 是可以互换的。这两组组件均可以在屏幕上显示出七巧板的形状,但其各自的显示方式却是极不相同的。

进程中、本地及远程组件 TangramModel 组件可以作为进程中、本地或远程组件运行。客户程序将向用户询问他们希望按何种方式运行 TangramModel 组件。

下面三节我们将讨论 Tangram 中一些在本书前面的章节中未曾波及到的方面。

13.3 IDL 文件

在前面的两章中我们使用一个 IDL 文件来描述示例程序中的所有接口和组件。虽然这样做对于一个例子来说是可以的,但对于一个大型的程序而言,我们希望每个组件只能看到它所使用的那些接口。因此,Tangram 将把每一个接口或一组相关的接口放到一个单独的 IDL 文件中。我们可以给这些 IDL 文件加上一个 _I 后缀。例如 MODEL_I.IDL 包含的是 ITangramModel 和 ITangramTransform 的定义。为建立类型库,可以在 IDL 文件中加上一个 coclass 语句和一个 library 语句。这些用 coclass 语句描述的组件将被放到单独的 IDL 文件中,我们可以给它们加上一个 _C 后缀。这些 IDL 文件将引入它们所用的 _I IDL 文件。

使用上述方法的优点是灵活性。但是由于每一个 IDL 文件都将生成其他的好几个文件,因此会由于文件众多而造成一些混乱。为避免此种混乱,我们可以将 _C 想象成是 CLSID,而将 _I 想象成是 IID。若在代码中使用一个 IID,则应包含相应的 _I 头文件。例如若将 IID_ITangramModel 定义在文件 MODEL_I.IDL 中,那么在查询 IID_ITangramModel 接口的代码中,必须包含头文件 MODEL_I.H 并将其同 MODEL_I.C 一块链接。

在创建一个 TangramModel 组件时，需要一个 CLSID _ TangramModel 值。若 TangramModel 组件是在文件 MODEL _ C.IDL 中描述的，则需要包含头文件 MODEL _ C.H 并将其同 MODEL _ C.C 一块链接。若在一个 IDL 文件中引入了另外一个 IDL 文件，则在 C++ 代码中包含与此引入的 IDL 文件相应的头文件。例如假定 MODEL _ I.IDL 引入了 EVENTS _ I.IDL，则在包含 MODEL _ I.H 时，还需要包含文件 EVENTS _ I.H。

上面的 _ I 和 _ C 后缀都只是作者所用的一些约定，读者完全可以按照自己的习惯命名这些文件。在没有后缀的情况下，作者经常弄不清楚某个定义在哪个文件中。在加上了后缀之后，当编译报告说它找不到某个 CLSID 时，我们就能马上知道需要包含哪个头文件或将其同哪个 C 文件一块链接。

13.3.1 DLLDATA.C 文件

MIDL 编译器并不是在所有的情况下都会生成一个新版本的 DLLDATA.C。在许多情况下，程序员可能希望用一个代理/残根 DLL 来支持多个接口。但是这些接口是在不同的 IDL 文件中定义的。若 MIDL 编译器找到一个已有的 DLLDATA.C 文件，那么它将把新的接口加入到此文件中而不是创建一个新的文件。所以应随时检查此文件以保证其中只列有所需的接口。

13.4 循环引用计数

当 TangramGdiWorld 组件创建一个 TangramGdiVisual 组件时，它将给被创建的组件传递一个 ITangramGdiWorld 接口的指针。TangramGdiVisual 组件将使用此接口指针以将模型单位转化成像素。但不幸的是，这样将引起一个循环引用(参阅图 13-4)。可以看到 TangramGdiWorld 将指向 TangramGdiVisual，而后者也将包含一个指向 TangramGdiWorld 的指针。

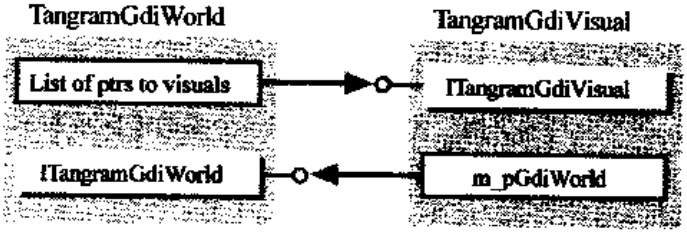


图 13-4 Tangram 程序中的循环引用

循环引用与引用计数的作法是有冲突的，因循环引用可能产生一些永远也不会被从内存中释放的组件。例如，假定 TangramGdiWorld 创建一个 TangramGdiVisual 组件，并获取一个 ITangramGdiVisual 接口指针，然后用此指针调用 AddRef。TangramGdiWorld 组件也将把它的 ITangramGdiWorld 接口传递给 TangramGdiVisual，后者将根据调用此接口上的 AddRef。这样 TangramGdiVisual 和 TangramGdiWorld 组件的引用计数均将至少为 1。

TangramGdiWorld 将在其析构函数中释放 ITangramGdiVisual(此析构函数将在组件的引

用计数值为0时被调用)。但TangramGdiWorld具有一个指向TangramGdiWorld的ITangramGdiWorld接口的指针,它只是在此引用计数值为0时才会释放此接口。这样将会导致一个死锁:TangramGdiWorld将只是在TangramGdiVisual释放了它之后才会释放TangramGdiVisual,而TangramGdiVisual也将只在TangramGdiWorld释放了它之后才会释放TangramGdiWorld。

为解决这个问题,可以选用三种不同的方法:不调用AddRef、显式地终止一个组件或使用另外一个组件。

13.4.1 不调用AddRef

第一种解决方法是最为简单的。这种方法实际是不对循环引用中的接口进行引用计数。在TangramGdiVisual组件中我们就使用了此种方法。TangramGdiVisual并没有调用TangramGdiWorld传给它的ITangramGdiWorld接口指针上的AddRef。由于TangramGdiVisual知道它是包含在TangramGdiWorld的生命期内的,因此它可以确信只要它活着,TangramGdiWorld一定是活着的,因而后向指针也将是有效的。

这种技术的使用是非常频繁的,因此我们可以给它定义一个相应的名称。对于一个没有被进行引用计数的接口的引用被称作是一个弱引用。弱引用不会将一个组件保持在内存中。与此相对,强引用指的是被进行引用计数的引用,这种类型的引用可以把组件保持在内存中(参阅图13-5)。

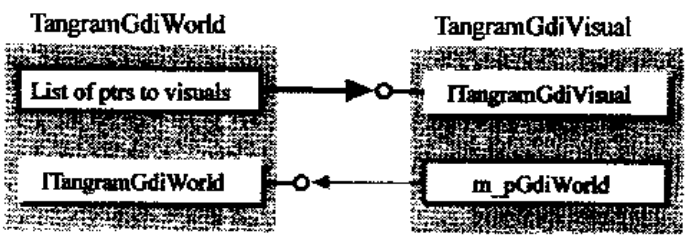


图13-5 TangramGdiVisual将具有对ITangramGdiWorld的一个弱引用。上图中我们用一个细的箭头来表示此种类型的引用

虽然此种方法是最为简单的,但并不是在所有的情况下都能使用它。具有指向另外一个组件的一个弱引用的组件需要知道此引用何时将成为无效的。上面的TangramGdiVisual因其生命期包含在TangramGdiWorld的生命期内,因此无需关心这个问题。但若两个组件的生命期没有嵌套,则需要另外一种方式来指示这一点。

13.4.2 使用显示终止

打破死锁的另外一种方法是给两个组件中的某一个(或两个)提供一种将另外一个组

件显式终止的方法。此时组件将不再等待被引用的组件的引用计数值变成为0,而是告诉此组件释放它的所有接口指针。这一点可以通过建立一个新的具有能够将组件终止的接口而方便地加以实现(参阅图 13-6)。

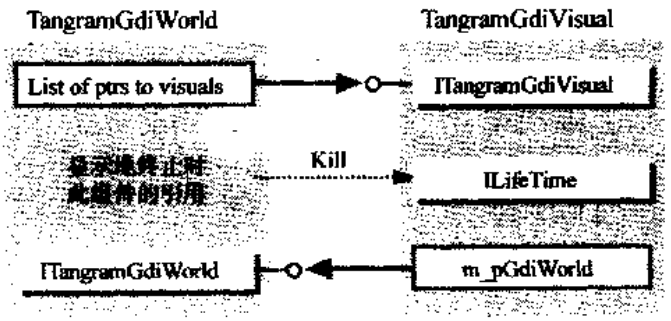


图 13-6 若使用一个单独的函数强制组件在其引用计数值变成 0 之前即释放对其他组件的引用,引用循环也可以被打破

此时应注意,在实际的程序中,另外某些人可能仍然需要被显式终止的组件。所以一种比较好的想法是在传统的引用计数值之外,再实现另外一个真正的强引用计数。IClassFacroty::LockServer就是一个实现真正强引用计数的例子。类厂组件的引用计数值不会将一个 DLL 保持在内存中。但如果需要确保在使用某个类厂时此 DLL 在内存中,可以使用 IClassFacroty::LockServer。关于这些问题的详细讨论,可以参阅第 7 章和第 10 章。强引用计数的其他例子有 ILleContainer::LockContainer 和 IExternalConnecton::AddConnection。这些函数将给其客户提供对组件生命期加以显式控制的方法。

13.4.3 使用一个单独的组件

打破循环引用的另外一种方法是给循环中的某个组件提供一个单独的对象或子组件。此子组件将维护一个对其外部对象的一个弱引用。图 13-7 显示了这一点。图中 TangramGdiWorld 将控制 TangramGdiVisual 的生命期,而后者将控制 TangramGdiWorld 子组件的生命期。

子组件是避免出现循环引用的最为灵活的方法。此时程序员无需组件的源代码或关于组件的其他一些额外知识,只需知道支持实现子组件的接口即可。插入一个具有弱引用的子组件将可以避免出现循环引用。

TangramGdiWorld 和 TangramGdiVisual 没有使用子组件来避免循环引用问题。但 TangramGdiVisual 本身就像是一个子组件,并且维护了一个对 TangramGdiWorld 的弱引用。但 TangramGdiVisual 和 TangramModel 确实使用了子组件以避免在其连接点的实现中出现循环引用。关于这一点我们将在下面的一节中讨论。

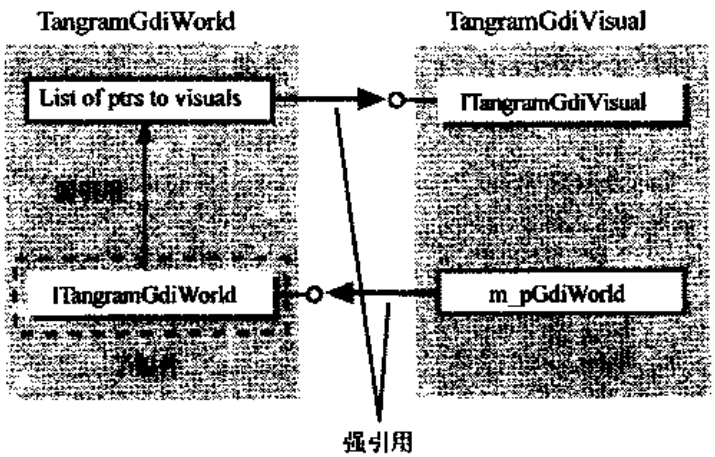


图 13-7 使用维护对其父组件弱引用的子组件，也可以打破循环引用

13.5 事件和连接点

到此为止，我们介绍的客户与组件之间的通信都是单向的：客户通过组件的接口对之实施所需的控制。但组件也可以充当一个客户并对其他组件实施控制。除了在聚合的情况下，本书中给出的所有组件都没有指向其客户的指针。在 Tangram 中这一点被无情地改变了。

在前一节中我们看到 TangramGdiVisual 具有一个指向 TangramGdiWorld 的后向指针。后向指针最常见的用途是可以通知客户各种各样的事件。在前一节我们已经看到，通知客户某个事件发生的最为简单的方法是使用一个弱引用，但更灵活的方法则是引入一个具有对外部组件弱引用的子组件。

在开发 AcitveX(OLE)控件时，需要一种通用、灵活的事件通知机制。解决这个问题的方法是使用连接点。一个连接点同一个套接字类似。客户可以实现某个接口，并将其插入到一个连接点中。然后组件即可以调用客户实现的这个接口。这些接口将被称作是外向或源接口。在 IDL 中需要用 source 属性来标记一个外向接口。之所以将其称作是外向接口，是因为在此种情况下，实际上是组件在调用客户代码。而将其称作是源接口则是因为在此种情况下组件调用的是此接口的源头。

下面我们来看一下一种非常简单的回调方式。在图 13-8 中，TangramModel 将在它遇到一个感兴趣的事件时调用接口 ITangramModelEvent。此接口是由 TangramGdiVisual 实现的。为避免出现循环引用，我们用一个子组件实现了 ITangramModelEvent 并将调用请求转发给 TangramGdiVisual。（关于此种作法我们在前一节已经看到过了。）

在图 13-8 中，TangramModel 是 ITangramModelEvent 接口调用的源头。而 TangramGdi

Visual 则是此接口的最终的事件接收者。图中我们假定 ITangramModel 具有一个初始化 m _pEvents的函数。此种解决方法虽然简单又很实用,但并非那么通用。这是因为,首先此处没有一种能让客户知道组件支持的所有事件的方法。其次此种方法只支持一个事件接口。第三,Tangram 只能给一个客户触发事件。而在许多情况下我们需要在某事件发生时通知几个客户。而连接点则正好可以解决这个问题。

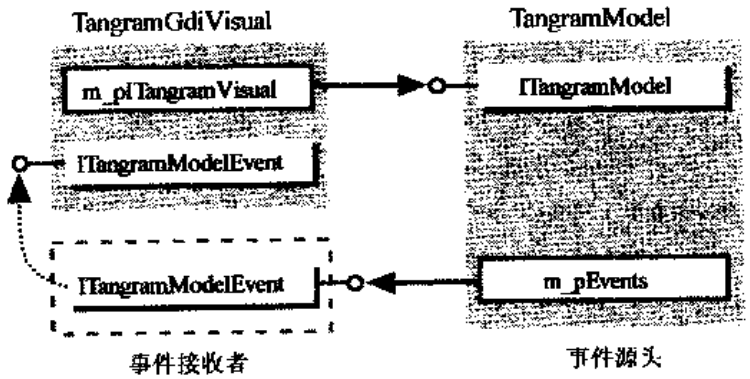


图 13-8 一种简单的回调方式

第一个问题可以用接口 IConnectionPointContainer 加以解决。IConnectionPointContainer 实现了两个函数:FindConnectionPoint 和 EnumConnectionPoints。函数 FindConnectionPoint 将接收一个外向接口的 IID 并返回指向此接口连接点的一个指针。EnumConnectionPoints 则返回一个可以列举出此组件支持的所有连接点的对象。这一点将可以满足组件需要支持多个连接点的需求。关于此对象,我们将在下一节中讨论。

连接点实际上是一个实现了 IConnectionPoint 的对象。每一个外向接口都同一个连接点相对应。每一个连接点均可以有多个接收者。IConnectionPoint::EnumConnections 返回一个指向某对象的 IEnumConnections 指针以列举出所有的连接。每一个接收者也可以具有多个源,所有这些都取决于接收者的实现。

图 13-9 显示出了 TangramModel 连接点的体系结构。TangramGdiVisual 使用 IConnectionPointContainer 来查找与 IID _ITangramModelEvent 相应的 IconnectionPoint。

TangramGdiVisual 把它的 ITangramModelEvent 接口传递给 IConnectionPoint::Advise。这将告知连接点 TangramGdiVisual 希望当发生事件时能够收到相应的通知。TangramModel 使用几个简单的 COM 对象来实现连接点。这些对象是用 C++ 的 new 操作符创建的,它们没有 CLSID,也没有在 Windows 的注册表中登记。在图 13-9 中用虚线框表示出了这些组件。这些对象实现连接点集合、连接集合以及连接点组件本身的集合的列举接口。虽然 TangramModel 只有一个连接点,但它仍应实现一个支持 IEnumconnectionPoints 接口的对象。关于列举接口,我们将在下一节讨论。

图 13-9 所示的连接点结构是比较复杂的,每一个连接点都是一个单独的对象,其中给出了两个列举接口。这种复杂性带来的正是系统的灵活性。

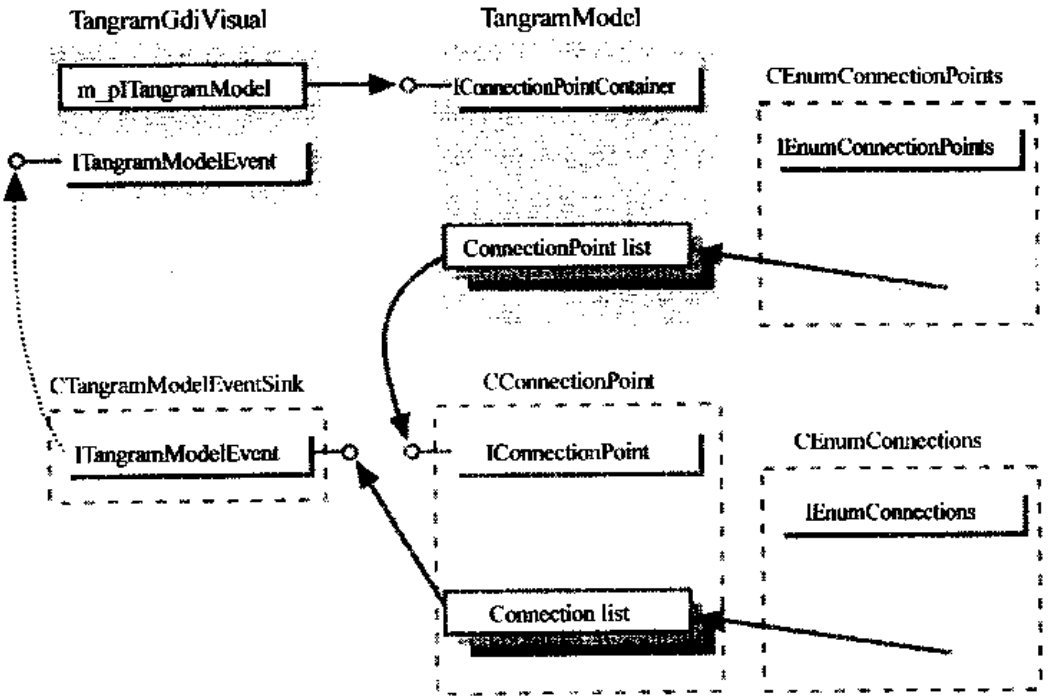


图 13-9 连接点的体系结构

13.5.1 IEnumXXX

在组件架构中,接口指针及其他数据的集合是非常重要的。因此 COM 为列举一个集合定义了一个标准的模式。此模式不需要一个标准的接口,因它的不同版本处理的是不同类型的数据。此种列举框是由接口 IEnumXXX 定义的,这个接口的成员函数分别是 Reset、Next、Skip 以及 Clone。在前一个例子中,我们已经看到了这种接口的两个例子:IEnumConnectionPoints 和 IEnumConnections。在第 6 章中我们实际上已经用到了一个列举接口 IEnumCATEGORYINFO 来列举所有可用的组件类别。

列举接口中的 Next 返回集合中的元素。Next 方法的令人感兴趣的方面在于它一次可以返回集合中任意数目的元素。这样程序员可以指定一次需要从组件中返回多少外元素。这一点在使用远程组件的情况下将可以显著地提高系统的性能,因它可以减小在网络上来回传送数据的次数。

列举接口的实现和使用都是非常简单的。不过此时应记住,在列举时都应调用相应接口指针上的 AddRef 和 Release。在实现 Next 方法时,在将接口指针返回给客户之前需要调用其 AddRef。在使用 Next 时,必须调用相应接口上的 Rlease。

列举接口要么是快照方式的,要么是活动的,但大多数列举接口都是快照方式的。例如 IEnumConnections 和 IEnumConnectionPoints 都是快照方式的。当调用 IConnectionPoint::EnumConnections 时,获得的列举对象是连接点建议的当前连接的一个快照。若由于其他

客户调用 Advise 增加一个连接或调用 Unadvise 删除一个连接而导致连接的改变，此时获得的快照不会被刷新。

13.6 本章小结

前面我们已经不只一次地说过，COM 是关于接口的。组件所用的相同的接口越多，就越有可能一致地使用这些组件。COM、OLE、AcitveX 控件、AcitveX 文档以及自动化已经定义了许多接口。COM 组件的开发者需要很好地研究这些接口。这样，即使在自己的应用程序中不用这些接口，也可以从中学到许多如何设计灵活组件架构的技巧。

759776
图书馆

结 束 语

到此为止，我们把所有的内容都介绍完了。读者此时应该知道如何在C++中创建COM接口；如何实现IUnkown和IClassFacroty；如何在Windows的注册表中登记自己的组件。同时读者也应知道如何用包容和聚合其他组件的组件构造应用程序；如何使用C++类和智能指针以简化应用程序代码的编写；如何用IDL文件来描述接口以便能够方便地实现参数的调整与类型库。IDispatch的实现只不过是使用ITypeInfo的一个简单过程。最后我们还介绍了如何实现套间线程中的组件。

有了上面的知识，读者完全应该可以编写出自己的组件了。本书没有给出的只不过是对那些特定COM接口的介绍。在读者知道如何实现一个组件之后，可以创建一个自己的接口或实现某个标准的接口。Microsoft已经为AcitveX、DirectX和OLE技术定义了成百上千的接口。一个AcitveX控件实际上就是某个接口集的实现；一个AcitveX文档也不过是一组接口的实现，并且AcitveX控件和文档共享了许多标准的接口。

AcitveX、DirectX以及OLE接口的实现并不是那么简单的。但是这些只不过是一些实现细节而已，其中的困难并不在于COM本身。实际上在理解了本书介绍的所有内容之后，读者应该可以算得上是一个COM专家了。若读者经常玩Tangram这个程序的话，你将会成为一个七巧板专家。希望读者能喜欢它！

