develop mentor series
DON BOX SERIES EDITOR

高级Visual Basic编程

Advanced Visual Basic

Matthew Curland 著

涂翔云 刘玉印 刘岩 译

Power Techniques for Everyday Programs

本书附赠光盘一张

Addison-Wesley

中国电力出版社
www.infopower.com.cn

高级Visual Basic编程

Advanced Visual Basic

Matthew Curland 著

涂翔云 刘玉印 刘岩 译

Power Techniques for Everyday Programs

中国电力出版社

内 容 提 要

本书是微软 Visual Studio 的资深专家 Matthew Curland 多年 VB 开发经验的总结，着重于解决 VB 程序员编程时所遇到的困难。由浅入深，循序渐近地讲解了如何开发标准的 VB 代码、高级编程技巧、以及如何通过一些新算法的使用提高编码效率和性能。附带的光盘提供了书中内容的相关代码，稍加扩展即可获得用户自定义类型、轻量对象系统、创建定制窗口以及函数重载等功能。盘中还提供了三个功能强大的类型库，对 VB 中使用的和由 VB 产生的类型库进行了修正。

本书适合高级程序员阅读，也可供专业计算机人士参考。

图书在版编目（CIP）数据

高级 Visual Basic 编程/（美）柯兰德（Curland.）著；涂翔云等译.
-北京：中国电力出版社，2001.5
ISBN 7-5083-0662-7
Ⅰ.高… Ⅱ.①柯…②涂… Ⅲ.Basic 语言-程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2001）第 034248 号

北京版权局著作权登记号 图字 01-2001-2232

本书英文版原名：Advanced Visual Basic 6

中国电力出版社出版、发行
（北京三里河路 6 号 100044 http://www.infopower.com.cn）
三河市实验小学印刷厂 印刷
各地新华书店经售
*
2001 年 7 月第一版 2001 年 7 月北京第一次印刷
787 毫米×1092 毫米 16 开本 27 印张 612 千字
定价 **55.00** 元

To Lynn and Elizabeth,
the two biggest blessings in my life.
I've missed you.

译 者 序

本书的原作者 Matthew 是 VB 编程方面的资深专家，是微软 Visual Studio 开发组的长期成员。他曾经参与了 VB 的设计和改进工作，多年来发表了许多有关 VB 编程的论文和专著。他在这本书中对 VB 编程的一些高级概念进行了深入的探讨，如聚合、单元线程和绑定等。这是一本有关 VB 编程的书，但作者并不是简单地介绍 VB 的特性，而更多的是从 VB、COM 和 OLE Automation 的关系这一更高的层次来探讨如何最优地解决问题的。这正适用于那些在 VB 编程方面有一定基础，希望能深入地理解使用 VB 编程的优点和局限，从而最大限度地发挥 VB 的能力，编写出高质量的专业水平的软件产品的读者。

本书由涂翔云完成了第一、二、六、七、八、九、十六章及附录部分，刘玉印完成了第三、四、五、十三、十四、十五章，刘岩小姐完成了第十、十一、十二章的翻译和校对工作。从本书的篇幅中读者也许能够体会到译者的艰辛，在翻译行将完成之际，我们看着彼此日渐憔悴的面庞，不禁感慨万千。但愿本书给广大读者带来的益处能使心血得以补偿。

在此我们要感谢家人给予我们的支持与理解，翻译本书适逢传统的春节，为了让本书能早日与读者见面，我们不得不放弃与家人团聚的机会。

在此对微软亚洲技术中心的沈小兵先生表示由衷的感谢，他给本书的翻译工作提供了大量技术上的支持，使得我们的专业不足得以弥补。

同时，还衷心感谢周郭飞先生、冯琳小姐、涂辉云先生还有姚欣小姐，他们做了大量的审阅稿件和校对文字工作，并提出了许多宝贵的意见，是大家共同的努力，促成了本书中文版的诞生。

由于我们对 Visual Basic 使用的时间不长，掌握的程度也有限，书中翻译错误难免，恳请各位同仁和广大读者批评指正。

涂翔云　刘玉印

于 2001 年 2 月 5 日凌晨

前　言

给一本书写前言是一种荣幸，特别当书的作者是一位著名作家而且这本书很有可能会成为 VB 高级编程人员的“圣经”时更是如此。但是，当让我写这几页前言时我的感觉要简单得多：“我终于有机会在 Matt 的书躺在书店里数月之前拜读它了！”，而我感觉到这更是一种荣幸。现在您应该体会到我是怀着怎样迫切的心情来阅读这本您也在阅读的著作了吧。

我甚至不用介绍 Matthew Curland。如果您曾经阅读过他发表在 Visual Basic Programmer's 期刊上的加黑标题的编程专栏或者在 VBITS 以及其他的会议上听过他的报告的话，您不可能不产生一种要把 Visual Basic 推向它能力的极限的冲动。即使您是 VB 的新手，当键入“.”时看到一大串控件或对象的方法和属性时，您也会感谢他。（Matthew 是给 Visual Basic 增加智能提示特性的开发小组的成员。）

基于下面的一些原因，我特别高兴 Matthew 完成了这本书。他在最近几年发表了许多文章，但那些文章由于篇幅所限，不得不将许多高级概念压缩到几页文字中，相比而言，您会更愿意阅读厚厚的一本著作。我在开始阅读本书时，迫不及待的想了解后面的内容，略过了一些我认为已经很好的掌握了的专题，但到后来我发现自己不得不回过头来认认真真的弥补知识上的漏洞。

第二个原因是我现在能够只享受我对 VB 的既爱又恨的感情的好的一方面。正如作者在序中所说的那样，许多高级 VB 编程人员喜欢 VB 是因为 VB 简单，但因为 VB 功能有限他们也憎恶它。这本书告诉我们使用 VB 可以做任何事情。同样重要的是，作者还告诉我们许多编写健壮、高效的应用程序的技巧，您在日常编程中会使用到这些技巧。实际上，在我发现这本书中包含了许多我以前的工程中出现的问题的解决方法时，不禁感到十分吃惊。我相信您也会有同样的感觉。

我喜欢这本书的最后一个原因是 Matthew 写了一本很有层次性的、可重用的书。作为一个程序员，您对代码重用这一概念应该不会陌生，但是书的重用对您来说可能是一个新事物。简单地说，当您可以多次阅读一本书并且每一次阅读都带有不同的目的的时候，这本书就是可重用的。一段时间之前我就认识到可重用的书确实存在，但实际的例子却不太好找。因为我已经说服您购买了这本书，我现在就要告诉您如何最大限度地利用这本《高级 Visual Basic 编程》。

第一次阅读时您不必仔细阅读书中的每一个字。只是对这本书作大概的了解，熟悉一下 Matthew 面对问题和解决问题的方法并且记下一些 VB 的未公开的细节。即使不想在不久的将来使用他介绍的那些编程技巧，这些细节也能省去您不少麻烦。您可以在几小时内

完成这一步，我想即使您三岁的小孩在旁边玩，这些时间也足够了。

现在要重用这本书了。这次您要坐在计算机旁，把您的小孩关在屋中。返回第一页，好了，略过前言，然后集中精力阅读书中的所有东西。这一步可能要花好几天的时间，但是您得到的回报是您可以借助 Matthew 提供的工具（也可不借助工具）使用您心爱的编程语言来创造奇迹。

您要勇敢地回到重用这本书的另一个层次。将这本书摊开在书桌上，浏览光盘中的代码然后搞明白每一行代码完成的任务。您会发现许多至今仍未公开的 VB 的秘密，您就会学会编写那些只有真正的权威才能讲授的代码。这一次您要关上您的 CD 播放器，摆上一杯咖啡并且可能数天不能去上班。

最后给您一个忠告，这来自于我的亲身经历：这本书可能使人上瘾。不要在睡觉之前看这本书，否则您不可能安然入睡。

Francesco Balena 简介
WWW.vb2themax.com的创始人
Visual Basic Programmer's 期刊的编辑
Programming Microsoft's Visual Basic 6 的作者

序

Microsoft Visual Basic 被微软定位为一种高级软件开发工具，它使无经验的程序员能够创建 Windows 应用程序。VB 中包含了丰富的工具包以支持快速应用程序开发，所以 VB 是一种真正的高级语言。除此之外，VB 也被越来越多的高级程序员们所利用。他们选择 VB 作为开发工具，主要是因为 VB 相对于其他应用程序开发平台而言，工作效率高一些，错误少一些。

高级 VB 程序员，对 VB 可以说是既爱又恨。他们爱 VB 是因为 VB 对底层细节进行了充分考虑，从而可以让他们将精力放在一些用户事务问题上。而他们恨 VB 则是因为当系统需要充分控制时，在 VB 中不能做到在更低层次上对程序进行完全的控制，从而也就不能控制整个系统。由于低层次的代码在一个大的应用中通常只占一小部分，所以 VB 的优点显得更为突出，对于一些存在的问题程序员们也可以忍受。绝大多数应用程序只需要少量的低层编码，例如：用它们来产生特殊的 UI 效应，消除性能瓶颈，减少内存的使用以及解决其他一些问题。这些考虑在一般的应用中可能只占很小的比例，但是他们对于一个专业化的软件产品来说却是绝对必要的。

本书着重于排除许多 VB 程序员在试图完善他们的应用时所遇到的困难。这些问题也是一个管理者在对一个新的软件工程项目中将要用到的技术进行评估时需要考虑的。管理者可能经常会选择使用低级语言进行编码，因为他们害怕整个工程不能在 VB 中完成。实际上，这种选择经常是不适宜的，因为低级语言已经不能适用于当今大量应用的发展。仅仅为了防止日后在工程中需要加入一些低层编码而花费几个星期甚至几个月的时间辛辛苦苦地在 VB 中去构筑传统的 C++代码似乎有些浪费。

消除 VB 中存在的障碍，最简单的方法便是利用 VB 中的隐藏数据类型。从创建对象到管理数组和字符串，几乎做任何事，VB 都始终与 COM 和 OLE 自动联系得十分紧密。尽管在 VB 中我们不能直接看到隐藏的 COM 类型，但实际上它们总是潜伏于表面之下：我们并不需要为了寻找相应的 IUnknown、BSTR、SAFEARRAY 和 VARIANT 等数据类型而去挖掘 VB 中对象、字符串、数组及可变类型变量下面深层的东西。COM 是基于二进制上的，因此，所有 COM 类型的内存布局都易于描述，也易于复制。

通过了解在 VB 中如何使用 COM 类型，我们将打开一个全新的 VB 世界。通过对这些数据类型进行认真考察，本书将打破 VB 表层，从而向读者展示位于标准 VB 代码表层之下的东西，同时也将向读者展示如何来创建自己的定制化对象。一旦让 VB 认识到一个定制化内存块实际上是一个数组或者一个对象，我们便能使用标准的 VB 代码来操作数据。

我并不打算教读者在 VB 中如何编写 C++代码，相反，我所要教读者的是如何去提高 VB，从而使 VB 代码运行得更好，并支持一些其他环境下不可实现的功能。

对于本书的读者，我在此提出三个主要的目标。

（1） 学会如何写出更好的标准 VB 代码，即使在读完本书后，并不打算去使用那些高级技巧，读者也仍然应该能够编写出更为健壮的 VB 代码，并使代码具有更加出色的性能。

（2） 学会一些高级 VB 编程技巧。

（3） 通过最小的代码重用，及一些新算法的使用，读者可以使用这些编程技巧来提高日常的编码效率和性能。与 VB 的指导方针一致，在本书中我也力图将一些底层技术封装，从而使读者可以在无需完全理解其底层结构的情况下很容易地使用这些技巧。

本书所面向的读者

本书特别适用于高级 VB 程序员，同时也适用于那些想进一步了解 VB 工作机理的 VB 和 COM 程序员。我强烈建议阅读本书的人应具备一定的 VB 和 COM 工作经验。如果读者不具备这方面知识和经验的话，那么最好买一本别的书与本书配套使用。

即使读者并不认为自己是一个高级的 VB 程序员，而且也并不像我一样对底层操作有较好的理解，那么通过学习本书，读者也将受益非浅。本书附带的光盘还提供了部分相关的代码，这些代码分别实现了子类化、对象聚合结构、函数指针调用、直接内存共享以及工作线程的创建等一些较常用的功能。只要再添入一些自己的代码，读者即可获得诸如用户自定义类型、轻量对象系统、定制化创建窗口以及函数重载之类的扩展功能。并且，读者还可获得三个功能强大的类型库，这些库对 VB 中使用的以及由 VB 产生的类型库进行了修正。

尽管本书中的许多代码都是底层代码，但本书的重点并不在于如何编写这一层次的代码。本书的目的是：从对整个接口实现的重用到优良的层次化对象模型，为用户提供一整套实用性的工具。读者可以利用聚合技术，并通过使用以普通接口实现为主体的组合对象来提高代码的重用。并且，使用这一技术的时间越长，我就越发现它能够简化我的一些日常代码。事实上，每一位 VB 程序员都能够从本书中找出对自己有用的代码并将其应用到日常工作中去。

未来的可兼容性

VB 7 的发行已开始，所以读者可能已经注意到：我所提供的一些技术在 VB 7 中是很有用的。例如，VB 7 支持具有继承性的重载函数、函数指针调用以及为类函数提供指针。如果读者现在利用本书所提供的技术来实践这些设计原则的话，那么，只要通过在 VB 7 中等价地替代这些框架调用，就可以将应用程序安全地转移到 VB 7 中。这里需要改变的只是框架代码：整个设计结构和代码逻辑都不需要作任何变化。在 http//www.PowerVB.com

网站上我尽我全力向读者提供了关于 VB7 兼容性的一些信息，另外，用户还可以通过 Matt@PowerVB.com与我联系。

本书内容摘要

本书共有十六章和一个附录，主要是从底层察看了隐藏于数组、对象及字符串类型之下的一些数据，并对这些结构的实际应用进行了讨论。

1. 创建块

本书中的许多技巧在很大程度上都依赖于对 VB 中数组和对象类型下的内存进行操作的能力。对底层数据的读和写需要直接对内存进行访问，或者称为“指针操作”。VB 中并没有为指针操作提供太明确的支持，但对于启动我们的旅行来说已经足够了。在第一章中不仅讲述了 VB 是如何稳定地来处理指针的，而且说明了在 VB 中如何对实际的指针进行访问。另外，从本章中读者还将了解到对支持性 VBoost 对象的有关介绍。本书中的许多样例程序都需要用到 VBoost 对象，所以读者在运行样例程序之前需要知道如何对 VBoost 进行初始化。

2. 使用数组

我以数组作为开始，原因并不在于我认为数组是用得最为普遍的一种结构，而是因为数组允许我们采用标准代码来使 VB 对任意内存位置进行修正。VB 中的数组变量实际上是指向一个被称为 SAFEARRAY 的数组描述符结构的指针。VB 自动地允许对数组的数据部分进行写操作，但除此之外，我们还可以对描述符、数组变量进行读写，从而进行直接内存访问。另外，读者还可以看到在简单的 VB 代码中如何来最好地利用数组。本章所讲述的技巧将贯穿全书，并且还可用于其他技术的优化中。

3. IUnknown:未知量

VB 要不断地与 COM IUnknown 接口打交道，但在通常的 VB 代码中我们是不会看到这个接口的。IUnknown 调用在运行时间和代码产生上的花费很大，因此理解 IUnkknown 接口是很有必要的。即使读者自己不调用 IUnknown，VB 也会频繁地调用它。从后续章节中读者将看到，熟悉 IUnknown 有助于在最底层上构建 COM 对象。如果读者不具备 VB 与 QueryInterface、AddRef、 和 Release 函数相交互的知识，就不能充分用户化 VB 的 COM 对象。

4．绑定函数到对象上

“绑定”是用来描述一个函数是如何调用另一个函数的代码的一般术语。绑定到对象上意味着将与对象类有关的代码应用到这一类的一个特定实例上。读者将看到 VB 编译器如何去确定什么时候、如何去绑定以及如何在运行时间动态的进行绑定。读者也将看到如何绕过 VB 放置用户控件的层来直接与 VB 的固有 VTable 接口进行对话，从而将运行时间减至最少。

5．对象的设计结构

一个设计良好的结构，不管采用何种技术，都是程序成功的最重要的因素。读者可以参考面向对象、基于接口、重用优化以及其他一些程序设计原理。无论读者的程序设计的原理是什么，目的都是写出稳定的、易于维护的程序。本章着重介绍可插入组件的 tried-and-true 原理、抽象和代码重用原理在 VB 中的实现。VB 本身提供了“Implements”关键字来为对象提供一个雏形，这一关键字在读者使用基于接口的语言来达到设计目标的时候起着很大作用。但是，读者会发现自己不能容易地重用实现代码。我将带读者摆脱 VB 的限制，通过聚合现在的对象以充分地重用实现代码。这给予读者将多个实现合并到一个对象中去的能力。

6．循环引用

每次设计一个对象模型的时候，几乎都会遇到循环引用的问题。如果读者事先知道如何通过强引用（标准对象变量）和弱引用（对象指针）来处理这种现象的话，那么便可以利用所需工具来设计对象系统并保证不会在系统拆分时产生问题。为了帮助读者对比各种不同的方法，我将向读者说明，在无需使用指针的情况下，解决循环引用问题所需的一些步骤（包括解决方案不够完善或过于繁琐的情况）。最后，我将把弱引用这项技术应用于集合、对象所有权以及层次对象模型中。

7．外部对象的创建

可将所有 COM 对象分为两大类：一类为工程内部对象，另一类为工程外部对象。COM 对象不能直接创建，相反，COM 使用注册表来创建类工厂对象，其中类工厂对象用来创建实际的类实例。因为根据应用程序路径而非注册表来直接从 ActiveX DLL 和 OCX 组件中载入对象，直接对类工厂进行处理将十分简单。在本章中，读者还将看到如何利用聚合及类工厂装载来动态地将控件指定为 MDI 页。本章的最后，将向读者讲述如何使用 CoRegisterClassObject 和 CoGetClassObject API 函数来构筑自己的应用程序级对象，并创建一个跨多组件的且易于取得的上下文对象。

8. 轻量 COM 对象

即使并不需要用到所有的 COM 支持，VB 仍然会创建一个完善的 COM 对象。轻量 COM 对象已经具备被 VB 识别为一个对象的功能，但没有 VB 所创建对象那么大的内存开销。令人激动的 COM 对象和令人生厌的用户自定义类型（UDT）之间的唯一区别便是 COM 对象中具有 VTable 指针。因此，可以为 UDT 提供一个自己的 VTable 从而把 UDT 转换成 COM 对象。在本章中，我们将看到几个简单的轻量对象，并展示了基于堆栈的局部对象以及局部或模块级结构中的终止代码，并且这些对象还具有实现任意可聚合到标准 VB 对象中的接口的能力。接下来，读者还将看到创建任意对象的所有步骤：VTable 布局、内存分配选择、引用计数、利用 QueryInterface 进行接口识别以及用于产生和处理错误的一系列可选项。

9. 大型多对象系统

对于创建少量的十分复杂的对象来说，采用 VB 类模块是完美的选择。但是如果以同样的方法去创建大量的简单对象，那么创建每一对象所花费的开销合在一起，其总开销将变得十分惊人。从日程安排工具到编译器，大量算法通常需要的都是成千上万个小对象而非少量的大对象。读者将在没有使用这些对象进行编码的经验的情况下，学习如何使用一个定制化内存管理器来分配轻量对象。当使用完这些对象之后，可以一个个地将它们释放或者利用单个调用来回收整个对象系统所使用的内存。

10. VB 对象和运行对象表

我们不能使用 GetObject 关键字来取得 VB 对象，因为 VB 并没有提供一种在运行对象表（ROT）中注册公有对象的机制。本章将细致地阐述轻量对象的使用和实现，从而使得可将对象放在 ROT 中，并且在对象终止后会自动地将其移除。关于实现细节，读者可以参看高级轻量对象专题，例如：在单个轻量对象中对多个接口的支持以及如何安全地保持对辅助接口的弱引用。

11. 调用函数指针

在 VB 的新版本中，除保留了 VB5 中的 AddressOf 操作之外，还提供了对函数指针的支持。其主要目的是提供对从 VB 中的 Addressof 出发来调用大量的 Win32 API 函数，另外也是为了能够为轻量对象建立 VTable。由于我们可以创建一个函数指针并调用它，所以本章中定义了 VB 上的一个转换 VTable 的轻量对象。函数指针为许多可能性提供了一个入口，例如：动态地装载（或卸载）DLL；编写任意的排序程序；在 VB 中使用显式堆栈分配；调用内联汇编代码。因为在调用标准函数指针方面仍然存在一些漏洞，所以我在本章中还介绍了如何将一个对象实例与函数指针（极大地简化了标准操作，例如子类化窗口）相联

系起来，并同时介绍了如何来调用 cdecl 函数，从而使得读者能够对 MSVCRT 中的入口项和其他的 cdeclDLL 进行调用。

12．重载函数

在完全没有提供重用的实现与提供了大量实现重用的聚合之间，还存在着一种部分实现重用，并且部分实现重用是继承性所提供的一个主要的优点。继承的优越性在很大程度上依赖于派生类为使对基类中函数的调用能够首先到达继承类而对某个函数的实现进行重载的能力。本章将说明如何来对函数进行重载，一是通过使用函数指针来重定向基类中的调用，二是使对单个对象的聚合定制化从而在无需与基类协作的情况下重载函数。为了在 VB 中使用继承，读者还必须编写一些附加的代码。不管怎样，这总比完全不能实现要好得多。

13．VB 中的线程

“VB 支持多线程组件”是一句简单的话，但它却意味着许多东西。本章试图弄明白多线程这一术语，并向读者展示如何充分利用 VB 的线程的能力。读者将看到 VB 如何在 COM 单元内运行以在线程层次上完成初始化的。读者还将会看到如何在一个 VB ActiveX EXE 工程中 spin 其他的线程。本书也将向读者展示如何在具有跨线程对象支持（提供优化的编程灵活性）或没有对象支持（提供优化的性能）的 DLL 中，启动一个工作线程。

14．VB 中的字符串

在 VB 中字符串操作是十分容易的。读者具有大量的操作符和字符串函数可供使用。但是，这并不意味着字符串操作是无代价的。每一个字符串是一块内存单元，越大的字符串需要越大的内存开销。自动字符缓冲可给读者有一种虚假的安全感，当字符串的大小大于缓冲区时这种感觉会马上消失。读者应结合 VB 字符串类型去了解处理字符串的真实开销以及如何使开销最小化，其中包括如何将一个字符串当作数值数组来处理以便提高处理速度。

15．类型库和 VB

尽管 VB 依赖类型库去编辑组件并向外部应用提供这些组件，但产生和使用类型库在日常工作中却不易碰到。读者和类型库的唯一直接联系是通过工程/参数对话框。产生完整的类型库几乎总是正确的，但在一些大的工程中，当读者迫切需要控制类型库时情况就不同了。本章将向读者展示如何创建 VB 使用类型库，以及读者编辑二进制兼容性文件和在 ActiveX 工程中创建的类型库的原因。本章没有详细介绍 PowerVB 类型库修改器、PowerVB 二进制兼容性编辑器、PowerVB 后期构建类型库修改器插件，这些都包括在本书的光盘中。相反，本章展示了让读者能使用这些工具来完成任务的步骤。

16. 控制 Windows

绝大多数客户端的 VB 程序都建立于窗口基础之上。事实上，VB 之所以能在众多的程序开发工具中占有一席之地，是因为它使得创建一个 Windows 应用变得十分简单，并免除了和各种复杂的窗口对象进行交互的麻烦。但是，如果试图在创建及窗口过程的层面上与窗口进行交互，这一保护层却带来了许多的麻烦。本章还将前几章中所讲到的函数指针、直接内存访问、轻量对象及函数重载等技术应用到一般的窗口对象和特殊的定制化控件上。读者还将了解到一种轻量的子类化方法，它用来控制即将到来的窗口消息。实现子类化之后，本书将着重于讲述定制化窗口的创建以及怎样来创建无窗口控件并使之如同一个带有窗口的控件一样发生作用。

附录：VBoost 参考

VBoost 对象是一个小的函数库，它为本章许多的技巧提供了基础。本书中包含了一个 VBoost 的 C++实现(VBoost6.DLL)和一个 VB 实现（VBoost.bas），这些实现使我们可以去除对外部 VBoost 依赖。并且它们提供了在执行聚合、定制化 IUnknown 钩入和函数重载中所需的任何东西。从这里还可以得到两个优化的内存管理器变量和在 VB 中大大简化指针操作的一系列赋值和算术函数。除了代码以外，VBoost 还包括一系列按功能分类的类型库。这些类型库将使得我们可以很容易地编译光盘中 Code 目录下的文件。

致　　谢

首先我想对我那才华横溢、美丽的妻子 Lynn 致以最深切的谢意，因为当这本书象蘑菇一样长成，开始呈现它的生命的时候，她只能忍耐我的日常时间的压缩。还要感谢我的女儿 Elizabeth。她与此书共同成长，把我从电脑旁拉开，告诉我是该休息的时候了。小孩总是知道什么是最重要的。

感谢那些审阅过本书的许多技术细节的人们：Troy Cambra、Dan Fergus、Michael Kaplan、Karl Peterson 和 Ben Wulfe，特别是 Brian Harris，他在这本书讲述的许多技术还没有修改完善时就已经使用过了。

特别要感谢为这一项目勾画雏形的 Bill Storage。他作为我的好朋友，一直热心地支持我的工作。Bill，我十分高兴和您一起探讨文章并且一起定义 VB black-belt 空间。

感谢 Glenn Hackney，他在这些年中忍受了许多不期的打扰，他是我新思想的知音和技术指导的源泉。。

感谢我的意大利朋友 Giovanni Librando 在两本书的项目中对我的信任。感谢 Francesco Balena，他为我写了绝妙的前言。与这两位绅士共事实在是我的荣幸。

感谢 Addison-Wesley 的 Kristin Erickson 、Rebecca Bence 和 Jacquelyn Doucette，当临近交稿时，他们表现得十分友好而有耐心，同样感谢 Gary Clarke，就是他在一年前用一本小书诱惑我进入到这个项目中来。现在那本小书变成了这本厚书，他知道一定会这样的。

感谢上个月黎明时提醒我睡觉的小鸟们。我会想念黎明的宁静，但是我不会忘掉我在深夜中对睡眠的渴望。

目　　录

第一章

构　建　块

众所周知，如果想要扩展一个系统的话，那么最好的方法便是从这个业已存在的系统本身开始进行构筑。本书介绍了大量新的编程技术，它们可以应用于各种使用 VB 开发的工程项目中。值得注意的是，所有这些都只是对已有的 VB 知识和技能的一个补充，而不是要取代它。就好比我们要扩建一幢早已存在的旧楼，新创建的部分仍然利用原有的地基和外墙一样，这些技术也都仍然建立在原有的 VB 系统结构之上。不过，在本书中，我们将会力图让我们的"新房间"和已有结构无缝的连接起来。

本章将对 VB 中部分我们可能用到的技术进行介绍，在书中的其他章节，我们将用它们来对 VB 进行扩展。很明显，了解一下我们工作是从哪儿开始的，对于学习本书中的其他章节来说是大有裨益的。这里，所有的扩展都源于一个最基本的概念，即 VB 是一种基于 COM 的语言，VB 的数据类型也都是 COM 数据类型。关于 COM 数据类型，已进行了很相近的描述，并且很容易进行声明。可以说，只要 VB 还是一种基于 COM 的产品，那么本书中所用到的技术就可以为我们所利用。但是一旦 VB 脱离了 COM 二进制标准，那么这些技术将必须作出相应的改变。正因为 VB 和 COM 就像同胞兄弟一样紧密联系在一起，所以在这里，我惟一敢肯定的是：假如想要使应用程序运行在未来的非基于 COM 标准的 VB 版本中，那么，本书中所列举的程序代码将不能直接进行使用，很多代码都必须进行重写。

1.1 虚　指　针

当考虑用 VB 进行编程时，指针可能是最容易被忽略的。实际上，如果想避免使用从 C++继承而来的复杂的指针操作，那么使用 VB 进行编程将是一个不错的选择。在 VB 中，甚至不允许定义一个指针类型的变量。我们知道，一个指针对应了内存中的一个地址，因此在 VB 中并不采用通过地址来操纵内存这一机理。然而，VB 不仅允许声明对象变量、数

组以及字符串类型，而且允许通过引用传递各种类型变量。而上述这些操作实际上都包括对指针的运用。对于这些操作，VB 和 C++惟一的不同之处在于 VB 实现了对这些指针的隐藏，使得它们对用户不可见。

下面，让我们来看一个简单例子，它们实现了为两个函数传递长整型值的功能。其中一个使用 ByVal 类型传递，另一个则采用 ByRef 类型传递。这两种情况所对应的 C++代码将清楚的表明在这个十分普通的例子中 VB 所做的工作（ByRef 类型是所有 VB 参数传递的缺省类型）。两种代码示例都做了同样的指针操作，但是 VB 自动的在调用代码和函数实现中完成了有关指针操作。我们所要做的只是在参数中指明为 ByRef 类型传递。

```
'Pass a parameter by value and the by reference
Sub ByValProc(ByVal Param1 As Long)
    Param1 =20
End Sub
Sub ByRefProc(ByRef Param1 As Long)
    Param1 =20
End Sub
Sub Main ( )
Dim lValue As Long
   lValue = 10

   ByValProc lValue
   'lValue is still 10

   ByRefProc lValue
   'lValue is now 20
End Sub

// Equivalent C/C++ code
void ByValProc (long Param1)
{
    Param1 = 20
}
void ByRefProc(long* pParam1)
{
    *pParam1 = 20;
}
void Main ()
{
    long lValue = 10 ;
    ByValProc(lValue);
    ByVaProc(&lvalue);
}
```

除使用 ByRef 类型来传递的参数之外，VB 中的一些固有类型同样也是基于指针类型的。实际上，参数类型可以分为两种：内联类型和指针类型。其中，内联类型包括字节型(Byte)、整型（Integer）、长整型（Long）、单精度型（Single）、双精度型（Double）、布尔型（Boolean）、货币型(Currency)、可变类型（Variant）、用户自定义型（UDT）、UDT 中的定长数组以及定长字符串，这些类型的数据存储在它们所定义的位置上并具有不同的长度。例如，一个字节型变量占据一个字节的存储空间，一个可变类型变量占据 16 字节的存储空间。指针类型（包括所有对象类型、变量、定长数组和字符串）是一种完全可变长度的结构，它所指向的数据被存储在另外的地方，而在定义变量的位置只是存储了一个指向这一块存有数据的内存空间的 32 位指针。尽管可变类型和用户自定义型都属于内联类型，但它们能包容指针类型，因此它们同样都占据了两块存储空间（即定义的位置和真正存储数据的位置）。

定义一个用户自定义型变量（在本书中也称为结构或记录）和一个具有相同数据成员的类模块，然后，通过调用 LenB 函数来察看它们的字节长度，我们可以清楚地看到内联类型和指针类型之间的不同。

```
'Type definition in a standard (.bas) module
Public Type TwoLongs
     Long1 As Long
     Lonb2 As Long
End Type
'Definition of CTwoLongs in a class file
Public Long1 As Long
Public Long2 As Long

'Calling code
Sub Main ()
Dim TL As TwoLongs
Dim CTL As New CTwoLongs
    Debug.Print "Structure Length:"; LenB(TL) 'Line 1
    Debug.Print "Class Length :"; LenB(CTL)   'Line 2
End Sub
```

正如我们所看到的一样，上面并不是一个十分让人满意的例程，因为 VB 将无法对上面例子中的第二行进行编译。VB 获知 CTwoLongs 为一对象类型，而且与该对象类型相联系的内存块长度没有进行定义。然而，我们可以另外再声明一个包含单个 CtwoLongs 元素的结构，并可验证该结构长度为 4 个字节。尽管 LenB 和 Len 函数适用子所有的内联函数，

但它们并不适用于除字符串变量之外的指针类型。在第 14 章中，我们将了解到获取一个字符串的长度是一件很容易的事情。

当 VB 拷贝一个内联类型的变量时，它拷贝的仅仅是数据。而当 VB 拷贝一个指针类型的变量时，它需要决定究竟是只拷贝一个指针值至新的位置，还是拷贝整个由该指针指向的数据并得到一个新的指针。仅仅拷贝指针值通常被称为“浅度拷贝”（shallow copy）；而拷贝实际的数据被称为“深度拷贝”（deep copy）。VB 根据 COM 的引用计算机理（我们将在第 3 章中对其进行讨论）总是对对象参数使用深度拷贝。而对于字符串和数组变量，没有引用计算机理，因此，VB 总是在指定一个字符串和数组时采用深度拷贝。深度拷贝在性能和内存上的开销将是很大的。但它不允许进入我们所不拥有的内存空间，从而保证了 VB 语言的安全性。我们将了解到一些有关字符串和数组的技术，这些技术通过直接引用我们所不拥有的内存空间来使深度拷贝最小化。当然，VB 没有办法得知该标准变量并不拥有内存空间，因此，下面将介绍在 VB 控制它之前如何清除这种自我管理引用。

传递一个 ByVal 对象可能经常会让一些 VB 开发人员感到困惑。通过值传递来传递一个对象并不表明我们是在做该对象的一个深度拷贝，相反，我们仅仅是传递了一个引用，同时增加了对象的引用次数而已。通过 ByVal 类型来传递一个对象和通过 ByRef 类型来传递一个对象主要的不同在于：当使用 ByRef 类型来进行传递时，我们能够修改该变量的实际内容，而使用 ByVal 类型来进行传递时，则无法对该变量进行控制。当然这仅仅是通过 ByVal 类型传递参数和通过 ByRef 类型传递内联类型参数在语义上的不同。除此之外，针对不同的对象类型，还有很多微妙的不同。我们将在第三章中的“参数和 IUnknown”部分对其进行讨论。

1.2 活 指 针

VB 在指针类型和 ByRef 参数的背后隐藏了所有的有关指针操作，但它也给予我们一个内建机制来访问内联类型和指针类型中的实际指针值。就像在任何其他的语言中一样，一旦拥有一个指针值，就可以像在其他语言中那样对它进行算术运算处理。当然，VB 本身并不彻底的支持指针在数学上的运算及其他操作。一旦我们开始使用指针，便意味着脱离了 VB 所保证的安全编程环境。并且在进行指针操作时得不到任何类型检查，编译器也不提供任何有关指针运算的帮助。也就是说，所有的一切便只能靠自己来完成。

指针允许任何个人以所偏爱的方式来操纵内存。基本指针操作运算可以做在 VB 中不可能做的事。例如，我们可以实现在两个类实例中共享变量而无需对它们进行拷贝，把一个字符串当作一个整型数组来处理，以及在任何内存块中创建一个数组。另外，除直接操作之外，本书同时注重利用 VB 中数组和对象类型的已有设计模式。一旦已经在内存中建立了一个对象或数组，便可以通过直接操作指针来将一个 VB 变量指定为数组或对象类型。不过，程序员的目标通常是尽可能使用标准 VB 方式来修改数据，从而在一个 VB 变量中得

到任何定制对象和数组。

在 VB 中，对内存进行操作的主要方法是使用 CopyMemory 函数。为使用该函数，用户应该在 VB 中或在一个类型库中为该函数提供声明。为了避免 ANSI/UNICODE 字符串转换问题（参见第 14 章）我们推荐对该函数使用类型库声明。CopyMemory 是自 16 位 VB 时代以来就有的被普遍使用的一个函数。在 32 位系统中，CopyMemory 是 Kernel.dll 中 RtlMoveMemory 的一个别名。CopyMemory 函数强制性的将一定数目的字节数从内存中的某个位置移至其他位置。例如，下面的代码段将 4 个字节从一个长整型变量移到另一个同样类型的变量上。注意，下一节我们将描述的 VBoostTypes 类型库包含了 CopyMemory 声明，并且 VBoost 对象扩展为许多本来需要使用 CopyMemory 函数的一般性操作提供了优化的函数。

```
'The VB declare statement for demonstration purposes
Declare sub CopyMemory Lib "kerne132" Alias "RtlMoveMemory"_
  (pDest As Any,pSource As Any,ByVal ByteLen As Long)

'Code to use it
Dim Long1 As Long
Dim Long2 As Long
    Long1 =10
    'Equivalent to Long1 = Long2
    CopyMemory Long2,long1,4
    'Long2 is now 10
```

在 VB 中有三种类型的指针。第一种是变量的内存地址；第二种是字符串、对象或数组指针的地址；（应该注意：指针类型的变量具有两个指针，一个指向变量，另一个则指向数据）第三种是函数指针。所有这些指针使用不同的方法进行返回。

可以使用 VarPtr 函数来得到任何非数组型变量的地址。VarPtr 函数在 VB 运行时被映射到一个简单函数上，它的功能是取出一块内存地址并返回该地址。Visual Basic 中的 VarPtr 函数在 VBA 类型库中进行了声明，并指向 VB 运行时 DLL 中的 VarPtr 项。通过映射不同的函数到该函数上，用户便可以得到不同变量类型的地址。例如，第二章中的“读取数组变量”部分介绍了 VarPtrArray 函数和 VarPtrStringArray 函数，它们都是通过不同类型的输入参数在运行时调用 VarPtr 函数。传递一个 ByVal 变量的地址等价于通过 ByRef 传递变量本身。因此，先前的 CopyMemory 调用也能够用 VarPtr 来实现。

```
CopyMemory ByVal VarPtr(Long2), ByVal VarPtr(Long1), 4
```

在 VBA 类型库中，还有两个内建的基于 VarPtr 函数的声明，即 ObjPtr 函数和 StrPtr 函数。其中 ObjPtr 可把指针赋给任何对象类型变量中的成员，而 StrPtr 则只能将指针赋给字符串变量中的字符。与 VarPtr 不同，ObjPtr 和 StrPtr 返回的是指向实际数据的指针而不是指向变量本身的指针。尽管我们可以通过 CopyMemory 和 VarPtr 函数来得到这些值，但看起来似乎直接调用 StrPtr 和 ObjPtr 函数要更简单一些。

```
'Copy the first seven characters from String1 into String2
Dim String1 As String
Dim String2 As String
Dim pString1 As long
    String1 = "PowerVB"
    String2 = String$(7,0)
    'These four operations all do the same thing

    'Try one: this requires a typelib CopyMemory declare
    CopyMemory String2,String1,14

    'Try two: Use strPtr to guarantee no ANSI/UNICODE munging
    CopyMemory ByVal StrPtr(String2),ByVal
      StrPtr(String1),14

    'Try three: Get the String's pointer from varPtr
    CopyMemory pString1,ByVal VarPtr(String1),4
    CopyMemory String2, ByVal pString1,14

    'Try four:Demonstrate VBoost.Assign instead
    VBoost.Assign pString1,ByVal VarPtr (String1)
    CopyMemory String2, ByVal pString1,14
```

通常，在 VB 程序代码中，VarPtr 和 StrPtr 的主要用途是激活对 UNICODE API 函数的调用。在 API 函数中并不需要用到 ObjPtr，但它能为任何对象提供一个惟一的数字标识符，因而显得十分有用。这样，无需通过使用一个对它的实际引用（参见第六章“弱引用和集合”部分），我们便可以产生一个对象的惟一标识。并且我们还可以使用 ObjPtr 来跟踪所有用 VB 建立对象的创建和销毁。例如，下面的程序代码便展示了一个类实例是如何创建和销毁的。

```
Private Sub Class_Initialize( )
    Debug.Print "Create: "; Hex$ (ObjPtr(Me))
End sub
Private sub Class_Terminate ( )
    Debug.print "Destroy: "; Hex$(ObjPtr (Me))
End sub
```

其实早在 Visual Basic 出现之前，VtrPtr 函数就已经是 Basic 语言的一部分了，当 VB 首次发行时，该函数被移植，作为 VB 的一个运行时函数。在 Visual Basic 1.0 直至 Visual Basic 3.0 中，这个虽然没有用文档进行说明但已为大家所共知的函数通过一个声明调用便可进入到 VB 运行时执行。而 32 位的 VB4 对作为参数传递的字符串类型数据进行了后台的 ANSI/UNICODE 转换，使大多数依赖于 VarPtr 的程序代码遭到破坏。这最终意味着 VarPtr 返回的是临时变量的地址，而不是实际变量所代表的数据。VB5 首先使自己能够调用类型库中的 Void 类型，并将 VtrPtr、StrPtr 和 ObjPtr 这些函数的声明加入到 VBA 类型库中（不过仍然对其进行隐藏），从而解决了这一问题。实际上，从 VB1.0 开始，VtrPtr 函数并没有发生任何变化，我们仍然可以在运行时调用它。只是，针对不同的 VB 版本，应该采用不同的函数声明方式而已。

在 VB 中，通过使用 AdressOf 工具，同样可以得到位于一个标准模块（.bas）中的任何函数对应的地址。AdressOf 主要是用来为 Win32 设定提供回调函数的。例如，如果我们想要获取一个系统的字符列表，或者想察看所有到达的消息而需要拆分窗口，都需要为系统提供函数指针。可是，VB 本身并没有提供任何可调用函数指针的方法，VB 只是能够提供它们。在本书中，对 AdressOf 的使用进行了扩展，使一些 Win32 的技术实现和创建定制 VTables 来构建 COM 对象（第八章将对其进行阐述）变得可能。另外，在第 11 章中，还将对如何直接调用一个函数指针进行阐述。

使用指针能够在一定程度上避免在程序运行中突然崩溃，在写这本书的同时，这种崩溃曾使我倍受煎熬。当然，可能并不经常会遇到这种崩溃现象，但有时为了做某种尝试，程序突然崩溃还是不可避免的。这里要提醒大家的是，为了避免不必要的麻烦，应该随时保存自己的工作成果。另外，还应该谨慎对待集成开发环境（IDE）中的“STOP”按钮和程序代码中的“END”字段。本书中涉及的许多技术要求在退出程序前先清除代码，再运行程序。因为一个突然出现的“END”有可能绕过许多可能导致终止程序的代码，并使 VB 潜在的处于一个不够良好的运行状态。当然，在点中 Stop 按钮后并不总是会出现崩溃，但足够多的崩溃现象常常使工作无法进行。通过在工具/选项对话框内的通用标签中关闭“按要求编译”选项将大大减少使用“Stop”按钮的必要性。

1.3 VBoost 对 象

本书中对 VB 所做扩展的核心部分被封装到一个我们称为 VBoost 的组件中。在本书的样例程序中，我们将看到许多地方包含了对 VBoost.AssignAddRef、VBoost.AggregateUnknown、VBoost.HookQI 以及其他函数的调用。所有这些样例都假定已经有了一个被称为 VBoost 的变量，其类型为在工程中实例化了的 VBoostRoot 下的一种。VBoost 包括三个核心部分以及一些卫星类型库（satellite typelibs）。读者可以通过从本书所附带的光盘中运行 Setup，从而在本地系统中安装这些文件。另外，在本书的附录部分中也可找到有关对 VBoost 对象和函数进行说明的整个文档。

VBoostTypes(VBoostTypes.Olb)类型库包含了对所有 VBoost 对象的定义。另外它还提供了本书中所要求的一些 API 函数的定义，例如 CopyMemory API 函数以及所有需要用来进行数组操作的 API 函数。VBoostTypes.OLB 被注册到“VBoost Object Types(6.0)”下。

VBoost6.DLL 是 VBoost 对象的一个 C++实现。该文件以“VBoostObject Implementation(6.0)”为名进行注册。编译该 DLL 中的对象时要求提供 VBoostTypes 库中包含的类型信息，不过，并不需要在已完成的软件产品中同时附带上“VBoostTypes6.Olb”。

VBoost.Bas 提供了所有 VBoost 对象的 VB 实现，并在 DLL 中包含了该实现的映像。这样，就可以轻松愉快的使用本书中提供的所有技术而无需在程序工程中添加额外的文件。并且，还可以通过改变 VBOOST_INTERNAL 这一条件编译值从而在其对应的 VB 以及 C++实现版本之间进行切换。

另外，还有其他一些类型库，它们均带有 VBoost 名字，因此在工程/引用对话框中很容易找到。这些类型库同样使用包含在主类型库中的类型，从而避免了对一些一般类型的重定义。这些文件包括：VBoost:Object Creation and Security(ObjCreate.Olb),VBoost:Ole Type Definitions(OleTypes.Olb),VBoost:Type Library Types and Interfaces(TlbTypes.Olb),VBoost:API Declares used for threading(ThreadAPI.Olb)和 VBoost:ROT Hook Types(ROTHookTypes.Olb)。当然，关于 VBoost 本身并不要求引用这些类型库。

应该注意的是：在运行本书中的样例程序之前，仍然需要初始化 VBoost 变量。其做法十分简单，下面给出操作步骤：

（1）用工程/引用对话框来增加对 VBoost Object Types(6.0)和 VBoost Object impliment-ation(6.0)的一个引用。

（2）从本书附带光盘中的 PowerVB\Code 目录下拷贝 VBoost.Bas 文件至本地系统，并将其加入到工程中。

（3）在程序代码中前面的某个部分调用 InitVBoost。这在主过程中很容易实现，但有必要引起注意的是：如果在程序中不只一次的调用 InitVBoost，可能会有一些小小的麻烦。

好了，可以开始了，从 PowerVB\Code 目录下找出这些文件，并针对 VBoost 扩展编写自己的代码。

第二章

使 用 数 组

对于许多态度严谨的程序员来说，数组是一个不可或缺的工具。数组允许程序实现从对单个数据项进行操作到对任意数量的数据项进行操作的转变，这是一个量的飞跃。另外，数组还能实现多个数据项的有序排列。因此，熟悉数组对我们来说是十分必要的。当然，我们并不一定需要了解在 VB 内部是如何控制数组的。之所以将这一章安排在本书的前面，是由于后续章节所讨论的许多技术都需要了解数组在 VB 中是如何使用的。

在最底层，一个数组仅仅是一个连续的内存块。该内存块被分割成一定数量的具有相同字节长度的元素单元。从该内存块的起始地址开始加入索引（索引应小于整个内存块的元素单元数目），我们便可以通过这一始于零的索引来对数组中的元素进行访问。一旦给定索引所对应的内存单元为数组所占有，最终的指针（resulting pointer）便可被视为一个与数组具有相同类型的非数组变量。

在这里，读者可能会认为，对于数组的定义没有包含多维数组以及索引非始于零的数组。事实上，这两种情况仅仅是对索引始于零的一维数组这一系统的扩展。如果索引并非从零开始的话（Dim MyArray(5 to 8) As Long），只需将该索引减去数组下界从而转变为一个始于零的索引，就可以正确对数组元素进行定位。而对于多维数组的情况（注意：维数也常常被称为数组的 rank），索引矢量可以通过计算转换成始于零的索引，并代入到公式<ArrayBase>+<Index>(Index<数组长度)中。编程语言支持非始于零的数组和多维数组从而避免了自己手工计算和转换的尴尬。由于这些复杂的数组类型实际上都基于一维数组，所以本章着重讨论这种始于零的一维数组。

元素数目	0				1				2				3				4				5				6			
字节编号	0				4				8				12				16				20				24			
十进制值	100000				1000				10				0				-10				-1000				-1000000			
内存	A0	86	01	00	E8	03	00	00	0A	00	00	00	00	00	00	00	F6	FF	FF	FF	18	FC	FF	FF	60	79	FE	FF

图 2.1 七个元素的长整型数组的内存设计

2.1 数组描述符

尽管任何数组的基本构建单元是包含有数组数据的内存块，但一个 VB 数组变量并不是一个指向数组数据块的指针。相反，数组变量指向的是一个描述了数组内容的结构，该结构被称为数组描述符，并且它具有 C++头文件中的 SAFEARRAY 类型（即用来描述数组的标准 OLE 自动化结构）。SAFEARRAY 在 VBoostType6.Tlb 中也被称作 SafeArray(SAFEARRAY 是 IDL/ODL 中的一个关键字，因此不能把它当作一个关键字来使用)。通过在 SafeArray 结构中描述数组，VB 可以使用各种由 OleAut32.Dll 提供的数组操纵规则。并且可以很方便的与外部组件进行交互。例如，VB 中的 ReDim 声明被映射到 SafeArrayCreate[Ex]API 上，并且它保护到 SafeArrayRedim 上的映射，消除到 SafeArrayDestroy 上的映射。

SafeArray 描述符在操纵数组方面占据了核心地位。该描述符中包含了数组边界、数组中包含数据的类型以及数组占有内存的原始分配等有关信息。这些信息不仅允许使用 SafeArray API 函数来对所有 VB 数组进行管理，它还允许当数组作为参数或函数返回值在不同线程和进程中的 COM 对象之间进行传递时对它们进行管理。另外，通过学习 SafeArray 结构以及 VB 和 SafeArray API 函数针对不同设置如何进行响应的有关内容，便可以做到对数组进行精确的控制。

下面给出了关于 SafeArray 和 SafeArrayBound 结构在 VBoostTypes 中定义的主要部分。它们实际上等同于通过 OLE 自动化来定义 SAFEARRAY 和 SAFEARRAYBOUND 结构。不过，这些结构对 VB 来说更为友好。实际上，在内存中一个 SafeArray 结构后面总是紧跟一个或多个 SafeArrayBound 结构，这里，SafeArrayBound 结构的个数由 SafeArray 中的 cDims 域来决定。另外，在 VBoostTypes 类型库中还包含了本章中所涉及到的所有标识符和 API 函数。

```
Type SafeArray
    cDims As Integer
    fFeatures As Integer
    cbElements As Long
    Clocks As Long
    pvData As Long
End Type

Type SafeArrayBound
     cElements As Long
     lLbound As Long
End Type
```

一般来说，读者通常可能倾向于使用一维数组，在 VBoostTypes 中也包含了对具有一个内联边界（single inline bound）的一维 SafeArray 结构的定义。

```
Type SafeArrayId
    cDims As Integer
    fFeatures As Integer
    cbElements As Long
    clocks As Long
    pvData As Long
    cElements As Long
    lLbound As Long
End Type
```

下面，让我们来单独的看一下 SafeArray 和 SafeArrayBound 中的每一个元素，以及我们可以应用于 fFeature 域中的不同标识。尽管对读者来说，这未免有些单调。但这个练习却有着重要的意义，因为我们将从用 VB 创建的数组中读取这些值，并且在我们自己创建的标识符中设置这些值。

- cDims 是数组的维数，对于一维数组而言，它必须设置为 1。
- cbElements 是数组中每个元素的字节数。数组中索引元素所占的内存地址通常由公式“pvData+Index*cbElements”来计算。
- cLocks 是对数组的锁定计数。当一个数组被锁定时，我们不能对其进行重新声明（ReDim，ReDim Preserve），也不能删除该数组。不过，仍然可以对数组中的元素进行修改，只是不能修改或销毁该数组的原有结构而已。当我们传递一个数组元素到一个 ByRef 参数或者使用“With”对一个结构类型的数组进行操作时，VB 会自动锁定该数组。如果在 cLocks>0 时试图销毁或重新声明保护（Redim Preserve），将会导致产生错误 10（“该数组已被固定或临时锁定”）。
- pvData 是包含在数组内的一个元素，它指向数组中数据所占有的内存。pvData 在 SafeArray 中扮演了一个重要角色：而所有其他域都只是起支持性的作用。pvData 被作为一个长整型值存储，在 OLE 自动化有关 SAFEARRAY 的定义中，它与 Void 类型的指针等长。
- cElements 说明了数组中所允许的元素的最大个数。
- lLbound 指明了数组的下界。通过 VB 中的 Lbound 函数可以直接从中读出其数量值，而 Ubound 则可通过计算 lLbound+cElements-1 得到。
- fFeature 是一个十分重要的域。其设置值使 SafeArray API 函数能够准确无误的销毁一个数组，并且使标准 COM 管理机（marshaling engine）能够成功的在不同线程或进程之间移动数组。所有这些标识可以分为两大类：其中一类负责说明数组中元素的类型；而另一类则被用来为描述符以及数组本身包含的数据分配内存。

第一组标识中包括三个内存标识—FADF_AUTO(&H1)、FADF_STATIC(&H2)、FADF_EMBEDED(&H4)和 FADF_FIXEDSIZE(&H10)。这些标识描述了数组是如何进行分配的，并且阻止了对其进行重新声明（ReDim[Preserve]）。如果设置了其中的任何一个内存标识，通过 Erase 能够清除数组中的元素，但并不能清空整个内存。如果 FADF_STATIC 被设置，Erase 对任何类型的内存清零。在 VB 中，变长数组的三个内存标识都没有被设置；定长数组对 FADF_STATIC、FADF_FIXEDSIZE 这两个标识进行了设置；而包含于结构中的定长数组则对所有这三个标识都进行了设置。VB 中并不产生设置了 FADF_AUTO 标识的数组。不过，在本章的后面部分将用到它，以表明该数组描述符在堆栈中被分配。

VB 在 fFature 标识以及局部定长数组变量上做了一定的手脚，（我们有充分的时间通过单步执行本地汇编代码来很好的证明这一点）。尽管，VB 为局部定长数组变量在堆栈中分配了描述符，但数组中的数据连同 SafeArrayAllocData 一起被分配在了堆中。这种分配方法通过在结构中传递大于 64K 这一界限的定长数组从而允许我们创建任意大的定长数组。不过，正如同设置这些标识的目的一样，这同时也意味着这些数组在某种意义上既不是定长的也非静态的。VB 只是在函数及类的生存期的范围内使用这些标识，从而防止其他类型的规则来修改数组。因此当一个局部数组变量超出其范围时，VB 将对定长的以及静态的数组（不包括 EMBEDDED 和 AUTO 类型数组）进行检查。VB 会在调用 SafeArrayDestoryData 之前清除这些标识，以释放这些被通过设置 fFeature 而锁定的内存块。

另外一组标识指明了数组中元素的类型。对于一个数组而言，只能设置其中的一个标识或不对其进行任何设置。FADF_BSTR(&H100)表明数组中包含的是字符串，因此在数组中调用 Erase 通过 SysFreeString API 清除了所有的字符串元素，然后销毁整个内存块或清零。与此相类似，FADF_VARIANT(H800)为每一个元素强行调用 VariantClear API 函数，而 FADF_UNKNOWN(&H200)和 FADF_DISPATCH(&H400)表明应该对每一个非零元素调用 IUnknown_Release 以释放对对象的引用。数组销毁规则使得仅通过查看这些标识便可和其他简单类型一样释放其占有的内存。

最后剩下的三个标识表明：在 SafeArray 结构之前，我们还可以从内存中直接获取关于数组类型的一些附加信息。这些值可以用来优化配置以及销毁结构中包含的数组。其中第一个标识，即 FADF_HAVEVARTYPE(&H80)表明了数组元素的变量类型占两个字节长度，并被存储在 SafeArray 结构起始点之前的四个字节的位置上。FADF_HAVEIID(&H40)表明描述符值之前的 16 个字节包含了数组类型的 IID。FADF_RECORD(&H20)表明了在描述符之前的四个字节的位置上包含了对 IRecordInfoFADF_RECORD 的一个引用。这和 VB6 中对于公共接口以及公有变量中结构的新的支持是相对应的。VBoostTypes 包含了 IRecordInfo 的一个可供 VB 调用的版本。

2.2 读取数组变量

本章的目的并不在于提供一种直接修改数组元素的方法，而在于为 VB 提供合适的描述符，从而使之能够以正常的模式访问各数组元素。不过，为了能够利用一个数组描述符，首先需要能够读出和写入一个 VB 数组变量。在能够对数组变量进行读写之前，需要拥有数组变量的地址。决定该地址实际上要比听起来困难得多，这是因为并没有内建（built-in）函数用以获取一个数组变量的地址。

VB 支持三种在函数声明中没有进行类型定义的变量。第一种类型即 Any，仅仅在 VB 的声明陈述中被允许，它承认任何非数组类型的变量。并且数据被传递到任何类型的参数中并自动的经过 UNICODE/ANSI/UNICODE 转换。而且，VB 还能够识别出在类型库中声明了的 void*类型。除了在参数被传递到这种类型时不会进行字符串转换之外，void*类型都可以等同于 Any 一样来看待。从 VB5 开始，void* 支持能够作用于任何类型的 VarPtr 声明。关于 Any 和 void*，它们存在着同样一个缺点，即编译器并不支持传递一个数组到这两种类型上。

不幸的是，VB 在类型库中并不支持 SAFEARRAY(void*)。因为一个 SafeArray 实际上起始于一个类型不可知的结构，这对用户来说同样是一个不幸：它导致无法编写一个在类型库中定义的能够允许任意类型的数组的 VarPtrArray 函数。这样，为了适应编译器，就必须包含对 VarPtrLongArray、VarPtrIntegerArray、VarPtrSingleArray 等一些诸如此类的函数的定义。当然，即便是把整个固有类型的列表都包括进去，也不能完全解决这一问题，因为仍然不能取得结构中的一个数组的地址。不过幸运的是存在一个 Any 和数组等同，这也就基本上解决了这一问题。下面的 VarPtrArray 函数对除字符串之外的所有类型的数组来说都是适用的。对该函数的定义包含在 VBoost.Bas 中。

```
Public Declare Function VarPtrArray _
  Lib "msvbvm60.DLL" Alias "VarPtr" (Ptr ( ) As Any ) As Long
```

对一个数组变量可以使用 VarPtrArray 函数来返回该变量的地址，但是这对取得数组描述符本身来说仅仅完成了一半的工作。例如对字符串类型以及对象类型而言，数组都是指针类型，变量的地址是一个指向一个数组描述符指针的指针，而不是该数组描述符本身（在 C++中语法为 SAFEARRAY**）。为了取得实际的描述符，必须废弃 VarPtrArray，然后拷贝该值到一个数组描述符中。以下的函数用来取得数组描述符。

```
Function ArrayDescriptor (ByVal ppSA As Long) As SafeArray
Dim pSA As Long
    'psa = *ppSA
    pSA = VBoost.Deref (ppSA)
```

```
    'Make sure we have a descriptor
    If pSA Then
      'Copy the safeArray descriptor
      CopyMemory ArrayDescriptor
        ByVal pSA, LenB (ArrayDescriptor)
    End If
End Function
'Calling code snippet
ReDim VariableArray (0) As Long
Dim FixedArray (0) As Long
Debug.print Hex$ (ArrayDescriptor ( _
  VarPtrArray (VariableArray) ).fFeatures)
Debug.print Hex$ (ArrayDescriptor ( _
  VarPtrArray (FixedArray ) ).fFeatures)

'Output.Both arrays have FADF_HAVEVARTYPE features set.
'FixedArray also has FADF_FIXEDSIZE and FADF_STATIC set.
80
92
```

另外，也可以使用 VarPtrArray 函数来查看存储于整个数组开始部分的附加信息。一般来说，在程序代码中并不需要这样做，但对于查看整个数组描述符设计来说，这确实是一个很有用的练习。对于这里出现的所有支持性的函数都提供了 VBoostTypes。VBoost.Deref 取回了存储于数组变量中的指针值，并且 VBoost.Udif 通过采用无符号运算的方法执行了指针计算。

```
Dim ObjArray () As VBA.Collection
Dim Guid As VBGUID
Dim GuidString As String *38
ReDim ObjArray (0)
'Read the IID from the memory stored at 16 bytes before
'the beginning of the normal SAFEARRAY descriptor.
CopyMemory _
  Guid, _
  ByVal VBoost.UDif (VBoost.Deref (VarPtrArray (ObjArray) ), _
   LenB (Guid) ),LenB (Guid)
StringFromGUID2 Guid,GuidString
Debug.print GuidString

'Output
{A4C46780-499F-101B-BB78-00AA00383CBB}
```

通过使用 TLI(TlbInf32.Dll)对象库我们可以验证，这对于一个集合对象来说是一个真正的 IID。加入一个工程引用到类型库信息并在立即窗口运行该代码，便产生了和前面程序段相同的输出。

```
?TLI.TypeLibInfoFromFile ("msvbvm60.Dll")._
  TypeInfos.NamedItem ("Collection").DefaultInterface.Guid
```

VarPtrArray 对于除字符串类型之外的每一种数组类型都适用。由于一些我一直都不能确定的比较模糊的因素，当我们在一个声明了的函数中采用 As Any 或 As String 传递字符串数组到 Ptr（）上时，VB 实际上执行了 ANSI/UNICODE 转换。这种行为非常让人感到惊奇，因为 SafeArray 是一个 UNICODE 中的“猛兽”，而不是一个 ANSI 中的“猛兽”，因此分配一个包含 ANSI 字符的 BSTR 实际上违反了该类型规则。至少 VB 并不对包含于结构中的数组执行字符转换，因此，与一个纯字符串数组中的字符串不同，在一个 API 函数调用期间并没有接触到 UDT 中的字符串。为了查看一个针对字符串数组的数组描述符，需要用到一个在类型库中进行了声明的 VarPtrStringArray 函数。如果使用 VarPtrArray，实际上得到的是一个临时结构的地址，并且如果废弃它将会迅速导致崩溃。如下面所示，类型库声明包含了对 VBoost 类型的定义。通过使用 VarPtrStringArray 函数和 VarPtrArray 函数，我们便可以访问一个任意类型的数组变量。

```
[dllname ("msvbvm60.DLL")]
module StringArray
{
    [entry("VarPtr")]
    long_stdcall VarPtrStringArray (
       [in] SAFEARRAY (BSTR) *ptr);
};
```

2.3 写入到数组变量

读取有关 VB 已经创建好的数组的信息实际上并没有给我们的编程技术添加更多的实质性的内容。但是能够通过数组变量的地址来读取有关信息，同时也就使我们能够在同样的地址中写入信息或者修改已经存在于所给定的描述符中的数据。通过填入自己的描述符以及把它们分配到一个数组变量，不仅可以给数组赋予任何自己所喜欢的性质，还可以在任何的内存位置中指向数组。下面的代码对这一基本的技术进行了说明，这些代码通过使用一个包含四个元素的 Byte 类型的数组来写入到一个长整型变量。

```
Sub FillLongViaBytes ( )
Dim lVar As Long
Dim Bytes ( ) As Byte
Dim SABytes As SafeArray1d
    With SABytes
        .cDims = 1
        .cbElements =1
        .cElements =4
        .fFeatures = FADF_AUTO or FADF_FIXEDSIZE
        .pvData = VarPtr (lVar)
    End With
    'Assign the address of the safeArray structure to the
    'array variable.You should remember this code.
    'VBoost.Assign is equivalent to CopyMemory xxx,yyy,4
     VBoost.Assign ByVal VarPtrArray (Bytes), VarPtr(SABytes)
    Bytes(0) = &H12
    Bytes(1) = &H34
    Bytes(2) = &H56
    Bytes(3) = &H78
    Debug.Print Hex$ (lVar)
End Sub

'Output (Intel uses big-endian integer layout)
78563412
```

在这些代码中，最重要、最值得引起注意的是它如何取出一个处于任何位置的内存块并应用于任何类型。我们本来在前面已经说过“把一个长整型变量的地址视作四个字节。”在这里，我们也可以让VB把同样的内存块当作一个具有两个整型变量的数组，（Integers()As Integer,cbElements=2,cElements=2）或者一个只包含一个长整型变量的数组（Long()As Long,cbElements=4,cElements=1）。我们可以通过改变变量来修改数组元素，也可以通过改变数组元素来修改变量，这是因为数组和变量指针都指向同样的内存块。如果数组和变量具有相同的类型，那么数组和变量就等值。

通过调整数组声明的类型及与目标类型的描述符范围的匹配，可以创建一个机制用来查看或修改任意的一个内存块为一个特定的类型。这种查看内存的方法被大量的应用到了本书中的其他部分。例如，通过数组描述符对字符串进行数量上的修改，以及在类实例和跨线程之间共享数据。另外，其他的应用还包括修改任意的内存块，例如一个内存中的位

图，不需要拷贝这块内存到一个 VB 分配的字节数组。

当使 VB 数组指向任意的内存块时，总是需要考虑其范围。一旦数组变量超出了范围，VB 便会清除该数组，就好像完成了所有的填充（populating）工作一样。如果一个数组变量超出了范围且没有设置 fFeature 标识的话， VB 把数据和描述符所用的内存传递到将被释放的堆中。对于堆来说，它并不喜欢释放它没有进行分配的内存，并且这将导致 VB 产生崩溃。正如我先前所提及过的一样，只要已经设置了 FADF_STATIC 和 FADF_FIXEDSIZE 标识，VB 便将其作为由 VB 管理的定长数组并且将该数据传递到堆中。这也意味着在指针指向一个位于任意内存块中的数组时，不能使用这些标识。用起来最为安全和保险的标识是 FADF_AUTO 和 FADF_FIXEDSIZE。当这些标识被设置时，VB 仅仅执行最少量的清除工作。

如果读者想通过 Erase 来对数组进行清零，可以添加 FADF_STATIC 标识。不过应该注意的是，于万不要在没有设置 FADF_AUTO 标识的情况下使用 FADF_STATIC 标识。通过以下对第一个例程进行的修改，读者可以从中看到使用 FADF_STATIC 标识产生的效果。当加入 FADF_STATIC 标识时，该例程输出为零。如果去掉 FADF_STATIC 标识的话，便可以得到预期的结果。

```
Sub CallFillLongViaBytes ( )
Dim lVar As Long
    FillLongViaBytes lVar
    Debug.print Hex$ (lVar)
End Sub
Sub FillLongViaBytes (lVar As Long)
Dim Bytes ( ) As Byte
Dim SABytes As SafeArrayld
    With SABytes
        .cDims =1              '1 dimensional array
        .cdElements =1         '1 byte per element
        .cElements =4          '4 bytes to a long
        .fFeatures = FADF_AUTO or FADF_STATIC or
         FADF_FIXEDSIZE
        .pvData = VarPtr(lVar) 'Point at the data
    End With
    VBoost.Assign ByVal VarPtrArray (Bytes), VarPtr (SABytes)
    Bytes(0) = &H12
    Bytes(1) = &H34
    Bytes(2) = &H56
```

```
    Bytes(3) = &H78
End Sub

'Output
.0
```

当然，当数组变量超出范围时，VB 会清空该数组中数据的现象意味着 VB 实际上做了并非必须的工作。毕竟，实际上并没有可供清除的东西。尽管数组变量超出范围是安全的，因为这时没有内存被释放，但是，为了做得更加完美一些，我们可以在变量尚未超出范围之前清除数据指针或数组变量本身。彻底清除该变量后，剩下来需要 VB 来完成的工作便所剩无几。此外，清除 pvData 域也是一个可行的选择。另外可以通过设置 pvData、cDims 或者 cElements 为零来防止当数组产生溢出时被清空。不过，在修改数组或描述符时，应该使其能够防止 VB 释放数组所占据的内存或对数组进行清零。毕竟数组只是借用一个内存块，而并不真正拥有它。

```
'Clear the array variable
ZeroMemory ByVal VarPtrArray (ByTes ( ) ), 4
'or
VBoost.AssignZero ByVal VarPtrArray (Bytes ( ))

'Change the array descriptor so that it has no data.
'Setting cDims or cElements to 0 is also sufficient.
SABytes.pvData = 0
```

2.3.1 模块级数组

修改局部变量可以提供对数组变量的精确控制，然而当我们在处理模块级的变量时，会产生与处理局部变量时完全不同的问题。第一个问题仍然是关于范围方面的。对于一个标准模块来说，它并不存在一个 Terminate 事件，因此当数组将要产生溢出时，我们并不能得到通知。一般来说，这将会是一个巨大的问题，不过 VB 中模块级拆分（modulate-teardown）代码并不遵从 FADF 内存标识。VB 假定它填充它本身的可变长数组变量，因此它总是应该能够知道怎样拆分它们。这样，即使描述符被标识为禁止的，VB 也总是调用 SafeArrayDestory Descriptor，这样就导致产生典型的释放未被分配的内存的问题。

读者也许会感到疑惑，为什么在第一个地方会想要使用一个模块级的数组/描述符对。事实上，如果我们调用一个多次用到数组描述符的函数时，一个模块级的描述符将使得我

们没有必要在每次调用时都去填充和分配描述符。可以在模块中设定一个 EnsureInit 规则，或者在模块级的描述符中 key off 一个域从而来测试一下是否需要对该数组描述符进行初始化。（当声明一个未进行初始化的数组时让 cDim=0 是一个不错的选择。）

在进行模块的拆分时，VB 不仅忽略 fFeature 的设置，而且同时忽略从 SafeArrayDestory Descriptor 返回的错误值。这样，对于拆分问题便会有一些解决办法。通过初始化标识符使 cLocks =1 来锁定数组可以防止不希望有的内存销毁。这种方法的缺点在于 VB 在拆分时仍然会关注 pvData 域的值，因此，无论何时产生局部溢出都应该确保 pvData 值为零。下面列出了解决该问题的三种方法。

- 将数组放入一个结构中，并在使用 light_weight 对象时加入终止代码。关于具体做法，第八章中将会对其进行说明。
- 在离开函数作用范围时清除模块级的数组变量，但对于这种情况（leave the structure in place），可能还需要用到局部描述符，因为所做的保存只是少量的。
- 通过创建一个类模块以避免标准模块引起的麻烦。这样，便可以自动的得到一个 terminate 事件。并且经常注意在类模块的 terminate 事件过程中清除数组变量。如果需要的话，还可以调用 Erase 来释放引用的指针类型。

这里，ClearArrayData 的例子表明了如何在不释放内存的情况下来清除数组中的数据。该例程依赖于一个简单的技巧，即通过临时性的设置数组描述符的 FADF_STATIC 标识来使得对 SafeArrayDestroyData 的调用实际上并不释放数组所占有的内存块。为了修改 fFeature 域，我们在这里使用了一个模块级的描述符/数组对，它能够允许直接对描述符进行读取和修改。另外一种方法是使用 CopyMemory 函数来加入数据到数组描述符或从中取出数据。在一个 FADF_STATIC 数组中，ClearArrayData 等同于一个 Erase 语句。下面列出了分别采用这两种方法的例程。其中第一个例程在数组中使用了锁定计数来用于块销毁，而第二个例程则通过使用轻量 ArrayOwner 对象来在模块拆分时清除数组变量。

```
'Version 1 of ClearArrayData. Uses a lock count
'to stop module teardown from freeing memory it shouldn't
Private m_SA As SafeArrayId
Private m_pSA ( ) As SafeArray

Public Sub ClearArrayData (ByVal ppSA As Long)
Dim fOldFeatures As Integer
    With m_SA
        'Initialize the module level variables once
        If .cDims = 0 Then
            .cDims = 1
            .fFeatures = FADF_AUTO Or FADF_FIXEDSIZE
            .cElements = 1
```

```
            .cLocks =1
            VBoost.Assign ByVal VarPtrArray(m_pSA),
            VarPtr(m_SA)
        End If
        'Change the data pointed to by the helper array
        .pvData = VBoost.Deref(ppSA)
        If .pvData = 0 Then Exit Sub
        'm_pSA (0) is a SafeArray-typed variable that can be
        'used to modify the memory of the SafeArray
        'structure.
        With m_pSA (0)
            'Save the old fFeatures value
            fOldFeatures = .fFeatures
            'Turn on the static flag
            .fFeatures = fOldFeatures Or FADF_STATIC
            'Clear the data
            SafeArrayDestroyData .pvData
            'Restore the old value
            .fFeatures = fOldFeatures
        End With
        'Make teardown safe again
        .pvData = 0
    End With
End Sub

'Version 2 of ClearArrayData.Runs code immediately before
'the array is released, giving us a chance to zero the array
'variable.The ArrayOwner structure needs to be included
'following it.
Private Type SAOwner
    Owner  As ArrayOwner
    pSA( ) As SafeArray
End Type
Private m_SAOwner As SAOwner

Public Sub ClearArrayData (ByVal ppSA As Long)
Dim pSA As Long
Dim fOldFeatures As Integer
    'Initialize the first time we call this
```

```
    If m_SAOwner. Owner.SA.cDims = 0 Then
       'Pass the element size and features.
       'The final False parameter means that
       'pvData is ignored on teardown.Setting
       'the features to zero is OK because this
       'array never makes it to teardown.
       InitArrayOwner m_SAOwner.Owner, _
        LenB(m_SAOwner.pSA(0)), 0, False
    End If
    'We treat m_SAOwner.Owner.SA the same way as
    'm_SA in the previous example
    With m_SAOwner.Owner.SA
        pSA = VBoost.Deref(ppSA)
        If pSA = 0 Then Exit Sub
        .pvData = pSA
        With m_SAOwner.pSA(0)
            fOldFeatures = .fFeatures
            .fFeatures = fOldFeatures Or FADF_STATIC
            SafeArrayDestroyData pSA
            .fFeatures = fOldFeatures
      End With
      '.pvData = 0 not needed because all cleanup is
      'done by the ArrayOwner object.
    End With
End Sub

'Calling code
Dim strArray() As String
. . .
ReDim strArray (SomeSize)
. . .
'Clear the array
ClearArrayData VarPtrStringArray (strArray)
```

在这里给出了一个一般性的帮助例程，该例程把一个内存块指针与一个仅包含一个元素的一维数组相连结并在完成工作之后清除了该数组变量。这样便避免了以往每次为一个数组变量连结一个标识符时都必须写入数组描述符代码的操作。在这个例程中，MemPtr 是可选的；而 pvData 则可以先将其置为零，待以后再对其进行填充。ElemByteLen 参数也是一个可选参数，这是因为该域对只包含一个元素的数组来说并不需要。另外，只有在想使

用 Erase 来为所有自己所拥有的内存置零的时候（as opposed to 0bytes of），才需要说明元素的长度。这一函数被包含在 SafeArray.Bas 中。

```
Public Sub ShareMemoryViaArray ( _
  ByVal ArrayPtr As Long,SAlD As SafeArrayld, _
  Optional ByVal MemPtr As Long, Optional ByVal ElemByteLen
    As Long)
    With SAlD
        .cbElements = ElemByteLen
        .cDims = 1
        .fFeatures = FADF_AUTO Or FADF_FIXEDSIZE
        .pvData = MemPtr
        .cElements = 1
    End With
    VBoost.Assign ByVal ArrayPtr,VarPtr(SAlD)
End Sub
Public Sub UnshareMemory(ByVal ArrayPtr As Long)
   VBoost.Assign ByVal ArrayPtr, 0&
End Sub
```

2.3.2 With 语句与重入

任何时候如果想要使用模块级的变量来替代局部变量，便可能存在重入(reentrancy)的风险，从而导致产生问题。例如，在 ClearArrayData 例程中，对 SafeArrayDestroyData API 函数的调用实际上能够释放一个 VB 对象，并运行 Class_Terminate 事件过程，从而最终导致调用 ClearArrayData 函数。这看起来似乎很危险，因为这时在堆栈中便触发了两个对 ClearArrayData 的调用，并且它们都使用模块级变量 m_SA 和 m_pSA。尽管这看起来是一件糟糕的事情，不过 With m_pSA(0)语句会保证其安全性。

之所以通过 With 语句来建立对一个数组的锁定的原因在于它只使用一次数组描述符来取回元素所占内存，便可以锁定整个数组以保证它所取得的指针总是有效的。在 End With 语句出现时，数组会被解锁。当然，这与我们的数组代码有关，因为数组描述符中的 pvData 域并不是紧跟在 With 行之后使用，我们可以任意的改变它。甚至，我们可以指定一个锁定了的数组变量至另外一个内存位置。对于 With 语句惟一的要求是当 End With 试图对数组进行解锁时，原始数组描述符所占的内存必须保持有效。在通常的情况下，锁定数组能够防止由 With 上下文变量所引用的内存块被释放。不过，以上我们只是讲述了锁定操作好的一

面。包含于 With 语句之内的任何操作实际上都引用了所占用的内存块，这可以从使用 With 语句时 SafeArray 的状态得到证明。VB 并不会从 With 语句块内查看 SafeArray 的状态，因此它也就看不到 PSA(0)值发生的改变。这样，就允许我们修改 PSA(0)的值而不受惩罚。

最后一点是，只要保证数组描述符在访问数组元素之前保持有效，便可以在重入（reentrant）的情况下使用模块级的描述符/数组对。如果没有 With 语句的话，ClearArrayData 将要求添加一些额外的代码来保证其安全性。正如我们从这里所列出的不带 With 的代码中所看到的一样，使用 With 语句要容易的多。

```
With m_SA
    'Change the data pointed to by the helper array
    .pvData = VBoost.Deref(ppSA)
    If.pvData = 0 Then Exit Sub
    fOldFeatures = m_pSA (0).fFeatures
    m_pSA(0). fFeatures = fOldFeatures Or FADF_STATIC
    SafeArrayDestroyData m_pSA(0).pvData
    'Guard against reentrancy, reset .pvData
    .pvData = VBoost.Deref(ppSA)
    m_pSA (0) .fFeatures = fOldFeatures
    .pvData =0
End With
```

2.4 数组选项：超出固定或可变字长

我们都知道，在 VB 中有两种类型的数组可供选择：即定长类型和可变长度类型。如果能够对数组描述符进行访问，便可以充分利用每种类型的优点从而在数组的操作上获得更多的灵活性。首先，让我们来看看使用定长数组的优越性，然后再说明怎样增强可变长度数组，以使其具有同样的性能。使用定长数组共有三大好处：首先，当数组作为一个局部变量时，数组描述符被分配在堆栈（stack）中。一般来讲，在堆栈中分配要比在堆（heap）中分配执行起来更好一些，这也正是使用定长数组优越性的一个体现。注意，这里的数组必须通过 Dim 关键字来定义，并且在声明中需要说明为固定字长。这里，一个普遍产生的错误是使用 ReDim 关键字来作为数组变量的主声明。

```
Snippet 1, compile error
Dim lArr (0, 0) As Long
lArr(0) = 10 'Wrong number of dimensions
```

```
'Snippet 2,no compile error, not a fixed-size array
ReDim lArr(0, 0) As Long
lArr(0) = 10 'Runtime error 9
```

其次，当数组被传递到一个外部函数时，定长数组的形状便不能再改变。FADF_STATIC 和 FADF_FIXEDSIZE 标识锁定了数组的长度和形状。

最后，编译器对当前范围内被声明的定长数组元素的访问进行了优化。由于编译器知道数组描述符存在于堆栈中，并且它还确切的知道到哪里去找 pvData。这样，便省去了怎样基于数组指针对 pvData 进行定位的代码（这对于可变长度数组而言是需要的）。不过，由于 VB 并不支持参数列表中的定长数组，所以当数组被传递到另一个函数时，这种优化不能被执行。

与定长数组不同，编译器不能在编译时对可变长度数组进行排列检查或一些其他优化操作。不过，使用一个简单的操作，便可以锁定数组的布局。这样，外部代码便不能修改该数组的排列和长度。除锁定数组长度之外，我们还可以把用来填充一个可变长度数组变量的堆分配的数目从两个减少到一个甚至为零个。

2.4.1 数组锁定

首先，让我们来看看一个可变长度数组是怎样防止其他代码来修改其形状的。在我们所写的大部分代码中，都没有必要对锁定数组太过在意。通常，我们是传递自己的数组至自己的代码中，并且我们知道调用代码不会任意去删改所传递过来数组的结构。然而，当我们传递一个数组到外部代码时，就像我们对 ActiveX control 触发一个事件一样，便再也不能指望返回时数组的状态仍然保持与开始传递过去的一样了。一般来说，当数据被传递到不可确信的代码时，应该想办法从外部进行控制，使之尽可能的减少对自己所传递过去的数据进行破坏。不管怎样，总应该确信自己的代码能够决定一个可信的边界。

通过修改 fFeature 标识，能够对自己所创建的数组结构进行合理的保护。设置 FADF_FIXEDSIZE 标识（它将导致对 ReDim Preserve 的调用，从而产生一个错误）能够锁定数组的长度。不过为防止调用 ReDim 同时也要求设置 FADF_STATIC 标识，因此设置 FADF_FIXEDSIZE 标识只是对数组进行部分保护。而使用 FADF_STATIC 标识产生的问题是：如果在数组被销毁之前没有清除该标识，整个数据所占的内存将得不到释放。为了使用 fFeatures 来锁定自己所创建数组的形状，必须在数组被第一次分配的时候并在其被销毁之前修改 fFeatures。尽管这对读者来说，是十分可行的（只需修改 ClearArrayData 规则来适当的对数组结构进行修改），但同时也存在着争论，因为另外还存在着一种内建的锁定机制可以帮助我们实现目标而无需对标识符进行修改。

锁定数组对于防止下降流（downstream）对所定义的数组结构进行修改来说，是一种较好的机制。实际上，对比于 STATIC 标识和 FIXEDSIZE 标识，锁定数组提供了更多的保

护，因为一个锁定完全性的封锁了 Erase 语句。下降流（downstream）代码只能对数组中的单个元素进行操作，而不能对整个数组进行操作。如果不希望自己数组的形状被修改，那么便必须在传递该数组之前增加数组的锁定计数，传递完毕后再减去所增加的锁定计数。如果不在调用之后对数组进行解锁，那么在数组被释放后便会产生错误并导致内存泄漏。这样，我们就需要提供错误处理过程以对可能产生的错误进行处理。正如先前所提到的，一个数组通常在数组的某一个元素（而不是整个数组）被传递到一个 ByRef 参数上或者当元素为结构类型并且被用于 With 语句之中的时候被锁定。采用这种方法，With 语句提供了一种用来锁定数组并能自动的对数组进行解锁的简单机制。然而，我们并不能对简单类型或对象类型的数组使用 With 语句。对于那些非结构数组，不能通过调用 SafeArrayLock 和 SafeArrayUnlock API 函数来直接进行锁定和解锁。

```
'Lock a Long array during an external call.
Dim lArr ( ) As Long
ReDim lArr (0)
    On Error GoTo Error
    SafeArrayLock VBoost.Deref(VarPtrArray (lArr))
    PassLongArray lArr
Error:
   SafeArrayUnLock VBoost.Deref(VarPtrArray (lArr))
   With Err
      If .Number Then .Raise .Number, _
        .Source, .Description, HelpFile, .HelpContext
   End With

Sub PassLongArray (lArr ( ) As Long
  Erase lArr 'Error 10
End Sub

'Lock an object type array during a call
Dim GuidArr ( ) As VBGUID
ReDim GuidArr (0)
   With GuidArr (0)
     PassGuidArray GuidArr
   End With

Sub PassObjectArray(GuidArr () As VBGUID)
   Erase GuidArr 'Error 10
End Sub
```

2.4.2 交替数组分配技术

对于一个活数组（viable），在数组中应同时具有描述符和数据存在。VB 中的 ReDim 语句对描述符和数组分别采用了堆分配。有两种方法用来减少一个可变长度数组所需要的堆分配的数量。第一种方法是通过调用 SafeArrayCreateVectorEx API 函数来实现，这种方法在单个堆中对描述符和数组进行分配。而第二种机制只对局部数组适用，该机制使用了一个局部描述符以及显式堆栈分配。

SafeArrayCreateVectorEx API 函数采取了一种可变类型（例如 vbLong 和 vbInteger）并且返回了一个分配好了的数组。下面，我们将说明该 API 函数的 Ex 版本以支持通过设置 FADF_RECORD、FADF_HAVEIID 和 FADF_HAVEVARTYPE 标识创建数组。读者可以尝试着在自己的程序中创建 VT_RECORD 和 FADF_HAVEIID 数组。下面，作为一个范例，我们使用单个的堆分配来创建一个包含 10 个元素的长整型数组。大家可以看到，当采用这种方法对数组分配完毕之后，它看起来就好像是采用 ReDim 语句分配的一样。

```
Dim lArr () As Long
VBoost.Assign ByVal VarPtrArray (lArr), _
  SafeArrayCreateVectorEx(vbLong, 0, 10)
Debug.print Hex$ (ArrayDescriptor (VarPtrArray(lArr)). fFeatures)
```

SafeArrayCreatVectorEx 函数使我们可以只在单个的堆而不是两个堆中分配一个数组。当然，我们还可以通过使用堆栈分配和自己的数组描述符来处理零堆分配。尽管这使用了更多一些代码，但这种可变长度数组分配机制却是开销最小的一种机制。堆栈分配仅仅在当前函数的生存期内有效，因此，只有在当前函数以及由当前函数直接调用的任何函数内，才能使用这种类型的数组。第 11 章中对堆栈分配规则进行了详细描述。

```
Sub UseStackAlloc ()
Dim lArr () As Long
Dim SAlArr As SafeArrayId
Dim l As Long
Dim cbExtraStack As Long
   With SAlArr
        .cDims = 1
        .cdElement = 4
        .fFeatures = FADF_AUTO Or FADF_FIXEDSIZE
        .cElements = 500
        cdExtraStack = .cdElements * .cElements
        .pvData = StackAllocZero.Call (cbExtraStack)
```

```
        VBoost.Assign ByVal VarPtrArray (lArr), VarPtr (SAlArr)
    End With
    '  .  .  .
    If cbExtraStack Then StackFree. Call cbExtraStack
End Sub
```

我们先前曾经提到，一个局部定长数组实际上是使用堆分配来支持任意大的数据块的。如果在编译时知道数组的长度并且想消除额外的堆分配，便可以对单个的数组元素定义自己的类型。这样便使我们的局部数组和模块级定长数组性能大大增强。并且，如果一个模块级的数组必须始终绝对保持有效直到每一模块中的每一拆分代码结束之后的话，那么，我们便必须对自己的数组使用这种技术。对于 VB 中的拆分过程，一个模块级的定长数组相当早的就丧失了数据，但模块级的结构变量中的定长数组却能一直保持到最后。

```
'One heap allocation
Sub HalfStacked ()
Dim Slots (255) As Long
End Sub

'No heap allocation
Private Type Long256
    lArr (255) As Long
End Type
Sub FullyStacked()
Dim Slots As Long256
End Sub
```

2.4.3 多维数组

到目前为止，我们都只是在处理一维数组。而要处理多维数组，我们还有很多新的工作要做。COM 并不能在编译时指定一个数组所要求的维数。这就意味着一个数组在运行时总是可以拥有任意的维数。另外，程序代码中的数组索引的数量必须与数组的维数相匹配，这样，便会存在问题。如果想要自己创建一个统计求和的程序来将一个数组中的所有元素结合起来的话，那么便需要为所能支持的不同维数的数组单独的编写程序。

对于高阶（high rank）数组而言，另外需要单独编写程序以支持不同维数数组并不是惟一的问题，多维数组中的索引项也是一个沉重的负担。下面给出了计算索引的程序代码，我们可以看到，整个 VB 代码等同于要访问一个多维数组中的单个项所做的工作。这种计算对于访问一个数组元素总是必要的。

```
'Calculate the zero-based index for a given item based on
'the array bounds and a set of indices. Both arrays are
'listed with the least-significant (right-most) index first,
'so the first index is in position cBounds - 1
Private Function CalcIndex(Bounds() As SafeArrayBound, _
  Indices() As Long, ByVal cBounds As Long) As Long
Dim i As Long
   CalcIndex = Indices(0) - Bounds (0). 1Lbound
   For i = 0 To cBounds - 2
      CalcIndex = CalcIndex * (Bounds(i).cElements - 1) + _
             (Indices (i + 1) - Bounds(i + 1).1Lbound)
  Next i
End Function
```

如果读者编写一个程序，只是想要得到一个数组而不考虑它的阶数（rank），则应该使得所新建的数组看起来仍然好像是下边界从零开始的一维数组一样。然后确保该例程的大小和归一化后的数组一致。这可以通过临时性的修改当前的数组描述符或者另外再创建一个数组描述符并使之指向同一数据。下面我们将对第一种实现方法进行讲述，第二种方法则留给读者作为练习。

为了对新建的数组进行归一化，需要把该数组转化成一个起始于零的在第一维中保存整个元素计数的一维数组。换句话说，即改变现在我们都已熟知的 SafeArray1d 结构中的 cDims、cElements 和 iLbound 项。我们可以先隐藏这些项中已存在的值，然后再修改这些值，等到对数组处理完毕之后再恢复其原来的值。

```
Private m_SADesc As SafeArray1d
Private m_pSADesc() As SafeArray1d
Private m_SABounds As SafeArray1d
Private m_pSABounds () As SafeArrayBound

Private Sub InitHelperArrays()
    With m_SADesc
        .cDims = 1
        .fFeatures = FADF_AUTO Or FADF_FIXEDSIZE
        .cLocks = 1
        .cElements = 1
        VBoost.Assign ByVal VarPtrArray (m_pSADesc), _
```

```
                VarPtr(m_SADesc)
    End With
    With m_SABounds
        .cDims = 1
        .fFeatures = FADF_AUTO Or FADF_FIXEDSIZE
        .cLocks = 1
        .cbElements = LenB(m_pSABounds(0))
        VBoost.Assign ByVal VarPtrArray(m_pSABounds), _
                VarPtr (m_SABounds)
    End With
End Sub

'Modify the array descriptor to be one dimensional and zero
'based to enable processing any array as a one-dimensional
Public Function NormalizeArray (ByVal ppSA As Long, _
  Bound0Start As SafeArrayBound, cDimsStart As Integer) _
  As Boolean
Dim i As Long
Dim lSize As Long
    With m_SADesc
      If .cDims = 0 Then InitHelperArrays
      .pvData = VBoost.Deref (ppSA)
      If .pvData = 0 Then Exit Function
      With m_pSADesc(0)
          m_SADesc.pvData = 0
          If .cDims = 1 And .lLbound = 0 Then Exit Function
          cDimsStart = .cDims
          Bound0Start.cElements =.cElements
          Bound0Start.lLbound = .lLbound
          m_SABounds.pvData = VarPtr (.cElements)
          m_SABounds.cElements = .cDims
          lSize = 1
          For i = 0 To .cDims - 1
              lSize = lSize * m_pSABounds (i).cElements
          Next i
          .cDims = 1
          .lLbound = 0
          .cElements = lSize
          m_SABounds.pvData = 0
```

```
    End With
End With
    NormalizeArray = True
End Function

'Undo the temporary damage done by NormalizeArray
Public Sub UnNormalizeArray(ByVal ppSA As Long, _
  Bound0Start As SafeArrayBound, ByVal cDimsStart As Integer)
Dim i As Long
Dim lSize As Long
  With m_SADesc
     Debug.Assert .cDims `Should call NormalizeArray first
     .pvData = VBoost.Deref (ppSA)
     If .pvData = 0 Then Exit Sub
     With m_pSADesc(0)
        .cDims = cDisStart
        .cElements = Bound0Start.cElements
       .lLbound = Bound0Start.lLbound
     End With
     .pvData = 0
  End With
End Sub

'Calling code
Public Function TotalItems (Values() As Currency) As Currency
Dim fNormalized As Boolean
Dim Bound0Start As SafeArrayBound
Dim cDimsStart As Integer
Dim i As Long
    fNormalized = NormalizeArray( _
     VarPtrArray (Values),Bound0Start, cDimsStart)
    On Error GoTo Error
    For i = 0 To UBound (Values)
      TotalItems = TotalItems + Values(i)
    Next i
Error:
  If fNormalized Then
    UnNormalizeArray _
      VarPtrArray (Values), Bound0Start, cDimsStart
```

```
    End If
    With Err
       If .Number Then .Raise .Number, _
       .Source, .Description, .HelpFile, .HelpContext
    End With
End Function
```

对于归一化后的数组来说，还存在着另外一个问题：即如果关闭了边界检查的话，那么下面所给出的所有归一化代码在本地进行编译和执行时都是没有必要的。这样，在编译了的代码中，如果我们指定了一个无效的索引值，也不会返回一个数组溢出的错误。而且即便我们指定了错误的索引值，也同样不会返回任何错误，尽管在我们指定了太多的索引值时，会导致程序产生崩溃。在边界检查被关闭的情况下，运行时进行查看的惟一的域是每一个 SafeArrayBound 中的 iLbound，而 cDim 和 cElements 则被忽略。所以，对于编译后可执行的代码而言，在代码中可以简单的忽略所创建的是一个多维数组这一事实。我们仅仅需要知道的是第一维中的下边界和数组中整个的数据项的数目。这样，就没有必要修改数组描述符。下面，我们将用一个被称作 CountArrayElements 的 NormalizeArray 的简化版本来决定元素数目。

```
Public Function CountArrayElements(ByVal ppSA As Long) As Long
Dim i As Long
Dim lSize As Long
    With m_SADesc
       If .cDims = 0 Then InitHelperArrays
       .pvData = VBoost.Deref(ppSA)
       If .pvData = 0 Then Exit Function
       With m_pSADesc(0)
          m_SADesc.pvData = 0
          m_SABounds.pvData = VarPtr(.cElemnts)
          m_SABounds.cElements = .cDims
          lSize = 1
          For i = 0 To .cDims -1
             lSize = lSize * m_pSABounds(i) .cElements
          Next i
          m_SABounds.pvData = 0
       End With
    End With
    CountArrayElements = lSize
```

```
End Function
'New, simplified calling code. Note that the error
'trapping is no longer necessary as the error handler
'Was needed only to restore the array descriptor.
'This works for multi-dimensional arrays only
'When bounds checking is turned off.
Public Function TotalItems(Values() As Currency) As Currency
Dim i As Long
Dim LowerBound As Long
    LowerBound = LBound(Values)
    For i = LowerBound To _
         CountArrayElems(VarPtrArray(Values)) +
                        LowerBound - 1
      TotalItems = TotalItems + Values(i)
    Next i
End Function
```

2.5 使用数组的一些小提示

除上面所述专题外，下面是一些更多的关于数组的一般性的评述，为了方便读者学习和总结，我们把它安排在本章的最后部分。

2.5.1 从函数返回数组

在 VB 中，函数名和 Property Get 过程名同样是位于过程内部的一个局部变量名。作为防止数据被复制的一种有效方法，我们建议读者在程序的主体部分使用该隐含变量名，而非仅仅在函数末尾使用一次。例如，如果我们有一个返回结构的函数，那么在下面所列出的两个程序片段中，应该使用第一个程序片段而不是第二个。

```
'Good code
Function FillStruct() As Struct
    With FillStruct
        'Fill in data
    End With
End Function
```

```
'Wasteful code, unnecessary copy
Function FillStruct() As Struct
Dim tmp As Struct
    With tmp
        'Fill in data
    End With
    FillStruct = tmp
End Function
```

如果我们能够将这一原理应用到返回数组的函数中，将会是一个不错的选择。然而在VB中对括弧（parenthese）大量进行重载，将导致这种方法在实际上是不可行的。

对于数组，我们通常能做三件事：即将其传递至另一过程、对其进行 ReDim 或访问某个数组变量。在这里，我们可以传递一个隐含函数名变量，但是我们不能使用由 ReDim 进行声明的变量，也不能对一个数组元素进行访问。这两项操作都需要用到括弧。VB 将括弧解释成一个对函数的调用，而不是对同名的局部数组变量中的一个索引。这样，我们就必须在程序返回之前使用一个分配给函数名的临时性的数组变量。

为函数名分配临时数组变量容易产生的问题是它可能会导致最终拷贝了整个数组。有两种方法可以避免这个问题。其中一种方法由 VB 本身所提供，而另一种方法是通过 VarPtrArray 来实现。如果在紧随 Exit Function/Property 或 End Function/Property 语句之前将一个局部数组赋值给一个隐含函数名变量，VB 会自动的传递该数组而不是对它进行拷贝。但是如果有一个插入性指令（包括 End If，不包括注释）的话，则该数组被拷贝而不是被传递。

在函数中的任何地方使用如下代码，就可以防止整个数组被拷贝，因为这时数组的所有权被明确无误的传递到了函数名变量。

```
'tmp must be a variable-size array to ensure that the
'descriptor and data are on the heap instead of on the stack.
VBoost.AssignSwap _
  ByVal VarPtrArray(FuncName), ByVal VarPtrArray(tmp)
```

2.5.2 关于上下边界的开销

正如我们前面所提到的一样，当对定长数组进行操作时，VB 编译器会执行一些优化工作。然而，这些优化工作并不包括对 UBound 值和 LBound 值进行编译。编译器会在 VB 运行时产生对这些函数的调用。这是一个可以用来执行自己的优化工作的好位置。我们可以

对边界使用常量值而不采用 UBound 和 LBound。

2.5.3 不进行边界检查的重要性

为了使对数组的访问能够更快一些，我们总是在高级优化对话框中选择“移除数组边界检查”这一项。实际上，如果在每一次访问时都对数组边界进行检查将不可避免的增大开销。并且，在最终的代码中，我们应该杜绝发生错误 9（下标超出范围）。虽然在调试过程中，它是一个极为有用的错误，但是对于最终形成产品的代码来说，绝对没有理由去访问超出边界范围的数组元素。如果不想关掉边界检查，那么意味着我们将要浪费大量的循环用来检查那些实际上从来不会发生的错误。另外，选择“移除整数溢出检查”和“使用长整型变量对数组进行索引”项也会改善对数组的访问速度。

2.5.4 决定元素长度

尽管读者可能会认为，在查看了一个数组的元素类型之后，那么该数组中的元素长度将是明显的，其实实际情况并不如此。使用 LenB 函数可以返回一种类型所要求的字节数，但是这个值并不包含数组元素间可能用来保持元素以 4 个字节为界进行排列所用到的填充字节。当然，如果所有类型都是按 4 个字节进行排列的话，将不会有任何问题。但实际上只有当元素在结构中进行排列时，才会使用填充字节。而且它并不是强制性的应用于任何结构。例如，一个格式为{ Long，Integer}的结构，其 LenB 返回 6，cbElements 返回 8。而一个格式为{Integer，Integer，Integer}的数组，其 LenB 和 cbElements 均返回 6。决定元素数量的最简单的方法是从数组本身寻找答案：查看 cbElements 域或者使用 SafeArrayGetElemsize API 函数。另外，还可以通过公式 VBoost.UDif(VarPtr(1)，VarPtr(0))来计算元素数量。这里需要注意的是，除一些简单类型外，并不能假定元素长度与 LenB 返回值是相等的。

2.5.5 多维数组的内存布局

与 C++使用列主（column-major）布局不同，SafeArrays 采用的是行主（row-major）布局。这意味着是在最近的数组元素基础上增加最后的索引（）。我们可以采用下面的代码段来查看内存布局。该代码段输出一个关于内存地址的顺序列表。

```
Dim Bytes() As Byte
Dim i As Long, j As Long, k As Long
ReDim Bytes (1,2,3)
   For i = 0 To 3
      For j = 0 To 2
```

```
            For k = 0 To 1
               Debug.Print Hex$ (VarPtr (Bytes (k,j,i)))
            Next k
         Next j
      Next i
```

SafeArray API 函数的设置以及描述符的布局对清除内存布局的混乱并没有太大的帮助。SafeArrayCreate 采用了对 SafeArrayBound 元素以行主（row-major）（VB）顺序进行排列，但是描述符本身以相反顺序存储该列表，所以实际上是按列主（column-major）顺序排列。同样，SafeArrayGetElement 以及其他 API 函数中也都将这些索引进行了相反顺序的排列。因此，为了得到 VB 中为字节{1，2，3}的元素，应该传递{3，2，1}的索引到 SafeArrayGetElement，并在 C++中使用（（unsigned char*）psa->pvData）[3][2][1]。

2.5.6 C++与 SafeArrays

我曾经与许多并不喜欢使用 SafeArrays 的 C++程序开发人员交流过。每一次听到他们这样说，我都会问他们使用哪一种 SafeArray API 函数，并且回答都会是一长串列表。例如 SafeArrayCreate、SafeArrayAllocDescriptor、SafeArrayAllocData、SaeArrayPutElement、SafeArrayGetElement、SafeArrayAccessData 等等诸如此类的函数。实际上这些调用在 Win32 中都是不需要的。在 C++中，可以简单的使用基于堆栈的 SAFEARRAY 描述符。将其中各个域填充好，并且让 pvData 指向适当的内存块，如果还没有一个适当的内存块的话，可以使用_alloca 在堆栈中分配一块。在 Win32 中，无需锁定数组或者使用 SafeArrayGetElement，pvData 也同样是有效的。只需简单的将 pvData 置为适当的类型并将其作为一个标准的 C 类型的数组来使用。这里应说明的是，如果读者脱离 API 函数设置来填充自己所创建的结构的话，那么相信你一定会诅咒 SAFEARRAY 集成竟是如此的复杂了。

第三章

IUnknown 接口:一个未知量

VB 创建并使用 COM 对象。所有的 COM 对象都是由 IUnknown 接口派生出来的。但是，IUnknown 接口并不等同于 VB 中的对象（Object）这一关键字。在 VB 中，对象实际上与 IDispatch 同义。IDispatch 是 VB 中的一个多功能接口，VB 支持 IDispatch 接口，但我们并不会经常使用它（读者经常使用 VTable 绑定，对否？）。IUnknown 接口在 VB 中没有对应的原始数据类型。VB 不允许声明一个 IUnknown 接口类型的变量，这个规定确实给读者做底层的对象操作带来了许多麻烦，比如在本书中要介绍的脑引用和聚合技术。在本章中，我们将详细介绍 IUnknown 接口在 VB 中的用法以及如何在 VB 中传递、声明、调用 IUnknown 接口。

讨论接口必须首先定义接口的成员函数的功能。IUnknown 接口包含三种函数：QaeryInterface 函数、AddRef 函数和 Realease 函数。首先让我们看一下 AddRef 函数和 Release 函数。在获得了一个对象的引用时调用 AddRef 函数，而不再需要该引用时就应调用 Release 函数。AddRef 函数增加内部引用计数，Release 函数则减小内部引用计数。当内部引用计数减为零时，Release 函数调用对象的 Terminate 事件并释放对象所占的内存。AddRef 函数和 Release 函数都不带任何参数，它们的返回值都是一个对象的当前引用计数。尽管它们的返回值通常是内部引用计数，但是这并不适用于所有对象的实现。如果 Release 函数返回 0，这表明对象已被销毁。如果返回别的值，表明对象依然存在。

QueryInterface(QI)函数可用来返回 COM 对象的某一个接口引用。在这种函数中，通常要传递一个接口标识符 （IID）。当全球惟一标识符 （GUID）用来识别接口时，就称为接口标识符。如果 COM 对象支持该接口，该函数将返回一个指向接口实现的指针。如果 COM 对象不支持该 IID，函数将返回一个错误。如果成功的返回了一个指针，QueryInterface 在返回之前调用 AddRef 函数或起同样作用的内部函数。COM 对象的用户通过调用 Release 函数来平衡 QueryInterface 调用和 AddRef 调用，并依此管理对象的生命期。如果 COM 的所有用户都遵从这一规则，那一切都可以正常进行。但如果 Release 调用的次数太少，不足以平衡 QI 调用和 AddRef 调用时，就会发生内存泄漏。反之，如果过于频繁的调用 Release

函数，VB 就很有可能会崩溃。除了在循环引用的情况之外（见第六章），VB 都会自动的平衡对 IUnknown 接口的调用。

VB 为标准类提供的 QueryInterface 函数支持 IUnknown 接口、IDispatch 接口和 VB 为接口生成的 IID，这个 IID 是由类中公共过程隐含的进行定义的。如果添加一些实现声明，QI 将会变得更加灵活，能够支持其他的接口实现。如给对象添加一些新特征（例如使其公有化或支持持久性）VB 就会支持别的接口，例如 IExternalConnection、IProvideClassInfo、IPersistStream、IPersistStreamInit、IPersistPropertyBag 等接口。如果创建一个 UserControl 类，VB 会添加对 IOleObject、IOleCotrol、IOleInPlaceObject、IOleInPlaceActiveObject、IQuickActivate 等接口的支持。反过来也可以说，VB 通过增加对别的接口的支持，使新创建的对象为一个持久类、ActiveX 控件或 ActiveX 文档。QueryInterface 函数是整个过程的关键。

所有的引用都会被返回，并且对不同的 IID 对 QI 调用返回不同的指针，那如何标识这些引用是属于同一个 COM 对象呢？COM 运用两种规则来在多个引用中建立对象标识。首先，当使用 QI 获取 ID_IUnknown 接口的引用时，返回值为同一指针。其次，VB 支持接口 ID，一个对象返回的所有引用必须支持同一个 IID。遵从上面两个 COM 识别规则就意味着实现一个“controlling（控制） IUnknown”对象。返回的任何引用都包含一个对控制 IUnknown 接口的内部引用，并且所有 QI 调用也只是简单的提交给控制 IUnknown 接口。然后，控制 IUnknown 接口对象将这个要查询的接口映射到特定的对象上去。

3.1 VB 和 IUnknown 接口

所有的 VB 变量类型能与 OLE Automation 中定义的变体（Variant）类型相兼容。VB 能处理三种变体对象类型：VT_DISPATCH、VT_UNKNOWN、VT_USERDEFINED。VT_DISPATCH 类型对应着 VB 的 Object 类型。这种类型的对象支持 IDispatch(也就是支持后期绑定)。VT_UNKNOWN 在 VB 中没有对应的类型，VB 可使用这种类型的参数来调用函数。VT_USERDEFINED 包含了其他的所有对象类型，例如对象的接口 ID 和有关接口 ID 的基础接口的信息。

在使用 Set 语句时、在函数调用中传递一个对象变量到一个参数时、使用 IS 或 TypeOf...Is 操作符时，VB 都是在与 IUnknown 接口发生作用。让我们首先看一下 Set 语句。

3.1.1 Set 语句和 IUnknown 接口

VB 的对象变量有两种状态：IS Nothing(无当前引用)状态和 Not Is Nothing（有当前引用）状态。一点都不奇怪，变量在 IS Nothing 状态时，其值为零，该值可通过检查 ObjPtr(Nothing) 来验证。Set 语句的最简单的应用是 Set ObjVar=Nothing 语句。当 ObjVar 为空时，Set

ObjVar=Nothing 语句不会做任何事情。如果 ObjVar 非空时，执行这一语句会对 ObjVar 调用 IUnknown 接口中的 Release 函数并将变量的值设为零。当对象变量的值超出范围时，VB 自动的对所有的对象变量调用 Release 函数。

使用 Set 语句将一个当前引用赋给一个对象变量绝对会更有意思。读者可能认为 Set 语句通常会调用 QueryInterface 函数，但事实上并非如此。实际上，许多 Set 语句只是简单的调用 AddRef 函数，而不是代价高的 QueryInterface 函数。AddRef 函数比 QueryInterface 函数更加简单，执行起来也更加迅速。当 VB 编译器遇到一个 Set 语句时，它会检查表达式两边的变量类型。（尽管读者在 VB 中不能声明 VT_UNKNOWN，我也将这一类型包含在下面的表中，以便于讨论参数传递）

在这张表中，变量的类型从上到下的排列在等号的左边，右边表达式的类型是从左到右的排列的。右边表达式所代表的类型可分为以下五类：

- UNK=VT_UNKNOWN 类型
- DISP=VT_DISPATCH
- USER:DISP=VT_USERDEFINED 有 IDispatch 支持
- USER:DISP=VT_USERDEFINED 无 IDispatch 支持
- RHLS=LHS 表示左边的变量类型与右边的表达式所代表的类型相同

要注意：赋值给 VT_UNKNOWN 并不执行 QI 调用，赋值给同样的用户自定义类型也是如此。当赋值给 AS Object 变量（VT_DISPATCH）时，如果右边的表达式的类型不是由 IDispatch 派生而来的，才会执行一次 QI 函数。如果 VB 定义的对象类型之间相互赋值时，由于这些对象类型是由 IDispatch 派生而来的，VB 在赋值给 AS Object 变量时，不会执行 QI 函数；在同一类型的变量之间相互赋值时，也不执行 QI 函数。VB 只在不同类型的变量之间相互赋值时才执行 QI 函数，从一个对象中得到一个已经实现的接口时通常要用到这种赋值。

表 3.1 确定 Set 语句何时使用 QI 或 AddRef

LHS⇓，RHS⇒	RHS=LHS	UNK	DISP	USER：DISP	USER：UNK
UNK	A	A	A	A	A
DISP	A	QI	A	A	QI
USER：DISP	A	QI	QI	QI	QI
USEP：UNK	A	QI	QI	QI	QI

为了讨论的方便，让我们看一下如果在赋值给 AS Object 变量时，VB 执行 QI 函数，多态对象（有多接口支持的对象）将作何改变。在下面的例程中，Class1 实现 Class2，Class2 有一个 Test 方法。

```
Dim Cls2 As Class2
Dim Obj As Object
Set Cls2 = New Class1 'QI for IID_Class2
Set Obj = Cls2 'Hypothetical QI for IID_Idispatch
Obj.Test 'Late bound call to Test fails
```

为何在上面的程序中后期绑定 test 调用失败？原因在于对 IDispatch 的 QI 调用被传回到控制 IUnknown 接口，控制 IUnknown 接口能返回支持 IDispatch 的多个接口。QI 忽视在接口 ID_IDispatch 请求中内含的二义性，返回它的主接口：Class1 而非 Class2。Class1 并不含有一个 Test 函数，因此 IDispatch 调用失败。这个例子表明，如果 VB 执行对 IDispatch 的任意 QI 调用，对第二个接口上的后期调用将得不到支持。如果试图从第二个接口中获取 IDispatch，实际上得到的将是第一个接口的 IDispatch。

3.1.2 参数和 IUnknown 接口

将一个变量或表达式传递给一个参数很像将变量或表达式赋给与参数同类型的变量。将两种 Set 赋值语句赋值的类型（AddRef 和 QI）和两种参数修改符（ByVal 和 ByRef）结合起来，就会产生四种组合，VB 在将它们传递给对象参数时要能对它们进行处理。我们将根据参数传递时调用的 IUnknown 接口函数，逐次介绍一下这四种组合。

组合 1：ByVal，需要用到 QI

调用者：对传递来的变量执行 QI 函数，得到参数的类型并将结果暂存于一个临时对象变量中。如果传递来的变量为 Nothing，则将该临时变量初始化为 Nothing。然后，传递这个临时变量。

被调用者：对传递来的函数顶部的 ByVal 参数执行 AddRef 函数（VB 产生的代码通常这样做，用其他语言创建的 COM 对象可能不会这样执行）。

被调用者：参数超出范围时，Release 这一参数。

调用者：对临时变量执行 Release 函数。

组合 2：ByVal，无需 QI

调用者：对最初的变量执行 AddRef 函数并传递该变量。

被调用者:进入并执行 AddRef 函数。

被调用者：离开时执行 Release 函数。

调用者：对原来的变量执行 Release 函数。

组合 3：ByRef，需要用到 QI

调用者：执行 QI，得到正确的类型，并将结果存储在一个临时变量中。然后传递该临时变量的地址。

被调用者：不执行任何操作。

调用者：如果此临时变量不是 Nothing，则对该临时变量的当前引用执行 QI 函数，得到传递来的参数的类型。将返回的引用置于另一个临时变量中。

调用者：对第一个临时引用执行 Release 函数。

调用者：对原来的变量的引用执行 Release 函数。

调用者：将第二个临时变量的值赋给原先的变量。

组合 4：ByRef，无需 QI

调用者：传递变量的地址（这就是全部操作）。

很明显，如果传递同种类型的数据，使用 ByRef 传递会最有效。特别在创建自己的工程时更是如此。尽管使用 IUnknown 接口的开销并不大，读者也必须认识到 ByRef 对象参数的其他含义。如果使用 ByRef 传递给不安全的代码，被调用者就有使传递来的数据无效的机会。这通常不是我们所希望发生的。并且，如果将对象传递给另一个线程里的一个对象的方法，不要使用 ByRef 传递，因为这会迫使管理器向两个方向移动对象。当然，如果要获取一个对象而不是传送对象，就需要 ByRef 参数。

很明显类型不相匹配的 ByRef 调用将是灾难性的。我很少看到有开发人员确实需要这种能力。在绝大多数情况下，被调用者并不实际改变传送来的引用，所以类型不相匹配的 ByRef 调用只是运行了许多无用的代码。在这个函数调用之后执行的 QI 函数看起来有些奇怪，因为它在函数调用之后执行。如果说进行调用时加上的这个 QI 调用还不太可怕的话，处理这些调用所需的大量代码可能让读者对这些调用便望而生畏。在做 ByVal 调用时，因为 AddRef/Release 调用发生在被调用者，所以调用者不需要对 IUnknown 做额外的处理。但是，对于 ByRef 调用，对 IUnknown 的处理工作都是由调用者来完成，所以当用类型不匹配的数据调用函数时，就会产生大量的代码。处理不匹配的 ByRef 调用要生成的代码大约是处理不匹配的 ByVal 调用产生的代码的两倍长。除非确实在使用 ByRef 参数，否则读者都应尽量避免这种结构。

我并没有解释表中的 As Any 和 void*参数类型，因为它们不是太有意思。尽管 As Any(用在 VB 的声明语句中)和 void*(用在类型库声明中)参数类型对于字符串来说表现不同，但它们对于对象类型来说表现相同，所以我在这里一并介绍。当传递一个对象到 ByRef As Any 参数时，VB 传递的是对象变量的地址。当传递一个对象到 ByVal As Any 参数时，VB 传递的是对象变量的值。在此种情况下，调用者不作 IUnknown 调用（我已经告诉读者这些没有多大意思。As Any 类型没有 QI 函数是一个便利条件，我将在介绍弱引用和循环引用时利用这一好处。

3.1.3 Is 和 TypeOf...Is 操作符

因为一个 COM 对象可以使用不同的指针值返回多个引用，所以仅仅靠检查 COM 对象的指针值来判断两个引用是否属于同一个 COM 对象是不可行的。如果 ObjPtr(x)=ObjPtr(y)，那很明显 x 和 y 属于同一 COM 对象。但是，如果 ObjPtr(x)<>ObjPtr(y)，这并不表明 x 和 y 不属于同一个 COM 对象。

VB 提供了 Is 操作符以对两个对象引用作明确的标识同一性测试。Is 操作符使用 COM 识别规则，该规则规定：对同一 COM 对象的所有引用返回同样的控制 IUnknown 指针。IS 操作符对 ID_IUnknown 的每一个引用执行 QI 函数并比较由 QI 返回的 IUnknown 指针，然后对所有的 IUnknown 指针调用 Release 函数以平衡由 QI 函数产生的引用。如果 IUnknown 指针相同，IS 返回 True。如果两个引用都为 Nothing 的话，Is 操作符同样返回 True。

VB 还提供了 TypeOf...IS 操作符来测试一个对象是否支持一个给定的接口。“TypeOf x Is y”在对象 x 上对 y 执行 QI 函数。如果 QI 成功执行，则对 QI 返回的引用执行 Release 函数，这时 Type...Of 返回 True。我并不建议读者使用 TypeOf...Is 操作符，因为 QI 是一个费时的操作，特别是对象是响应 QI 函数而建立的。在下面的代码中，*QI/Release* 将运行两次。

```
Dim x As SomeClass
Dim pIFoo As IFoo
If Typeof x Is pIFoo Then 'QI now/Release now
    Set pIFoo = x 'QI now/Release later
End If
```

读者也可以使用一个带有错误陷阱的 Set 语句来替换上面的代码。尽管这一替换后的代码不是很好，但它执行起来更快，因为在这段代码中对 IFoo 只执行了一次 QI 函数。即使加上执行错误陷阱的时间花费，总的时间花费要比第一段代码低百分之十五；可能一次执行节约的时间不是太多，但是如果该对象支持多个接口的话，节省的总时间将相当可观。

```
Dim x As SomeClass
Dim pIFoo As IFoo
On Error Resume Next
Set pIFoo = x
On Error GoTo 0
If Not pIFoo Is Nothing then
    'Process against IFoo
End If
```

3.2 声明 IUnknown 并调用它的函数

在本章开始，我曾经强调：在 VB 中不能声明一个 IUnknown 类型的变量。读者可能对此表示怀疑，因为曾经在自己的代码中声明过一个 IUnknown 类型的变量，或者在公开发行的代码中看到过这样的声明。这其中的微妙的区别是 VB 将声明为 As IUnknown 的变量当作带有 IID_IUnknown 的 IID 的 VT_USERDEFINED 类型，而不是真正的 VT_UNKNOWN 类型。

如果不声明变量为 IUnknown 类型并编译通过的话，整个工程将不包含对 stdole(OLE Automation)类型库的引用，这一类型库通常包含在 stdole2.dlb 库文件中。如果工程中不包含这一类库，就要使用 Project/Reference 对话框将类库添加到文件中。在这时，编译器能识别出 IUnknown 类型，但在语句完整性列表中并不能看到它，因为 IUnknown 是一个隐含类型。若读者确实需要看到它，就应打开对象浏览器（Object Browser）的内容（context）菜单，选中其中的 Show Hidden Members 项。

可以很容易的看出，从 stdole.IUnknown 中得到的 IUnknown 引用不是 VT_UNKNOWN 类型，这是因为赋值给 VT_UNKNOWN 并不产生 QueryInterface 调用。当读者将一个 VB 定义的类实例赋给一个 IUnknown 变量时，可以检查对象的指针，会看到一个不同的接口。这清楚的表明曾经做过 QI 调用。这也表明 VB 类的控制 IUnknown 接口和它的主接口并不相同。

```
'In Class1
Private Sub Class_Initialize()
Dim pUnk As IUnknown
    Set pUnk = Me
    Debug.print Hex$ (ObjPtr (Me)), Hex$ (ObjPtr(pUnk))
End Sub

'Output
19D988      19D9A4
```

如果在产生类型库（将在第六章介绍）时使用 IUnknown 类型，就可以利用到 VB 对 VT_UNKNOWN 和 stdole.IUnknown 的不同处理方法。下面举一个例子来说明这一点。VBoost 提供了一个叫做 CreateDelegator 的函数（见.第五章“IUnknown 钩入”）。该函数的第一个参数为 stdole.IUnknown 类型（在对象浏览器中显示为 **IUnknown**），第二个参数为 VT_UNKNOWN 类型(在对象浏览器中显示为 Unknown)。CreateDelegator 的第一个参数需要 VB 中的对象的控制 IUnknown，第二个参数则不需要。通过在类库定义中使用 stdole.IUnknown，就能在程序 CreateDelegator 中省略对 IID_IUnknown 的 QI，因为 VB 已

自动的做了这种调用。

如果要赋值给一个 IUnknown 变量时不执行 QI，可以利用 VBoost 中定义的两个函数。第一个为 AssignAddRef 函数，它类似于 Assign 函数，它将 pSrc 参数复制给 pDst 参数。复制完成之后，AssignAddRef 调用 AddRef 增加引用计数。由于参数被声明为 As Any，这里不需要 QI 调用。第二个函数为 SafeUnknown，它有一个 VT_UNKNOWN 类型的参数，返回的参数类型为 stdole.IUnknown。与 AssignAddRef 不同的是，SafeUnknown 可被用为内联函数。

```
'In Class1
Private Sub Class_Initialize()
Dim pUnk As IUnknown
    VBoost.AssignAddRef pUnk, Me
    Debug.print Hex$ (ObjPtr (me)), Hex$ (ObjPtr(pUnk)), _
      Hex$ (ObjPtr(VBoost.SafeUnknown (Me)))
End Sub

'Output
19D988     19D988     19D988
```

现在有一个 IUnknown 类型的变量，读者可能迫切的希望调用 QueryInterface 函数、AddRef 函数和 Release 函数。但当键入“punk.”时，得不到该变量的语句完整性列表。这是因为这三个 IUnknown 函数都为限制性的函数，VB 编译器不允许调用它们。OLE Automation/VB 的设计者有意将 IUnknown 函数设为限制性的，他们的决定基于如下的考虑：VB 为读者管理所有的 IUnknown 操作，以免读者在程序中发生误操作，而将 IUnknown 的函数设成限制性的函数可以十分有效的实现这一想法。在我介绍如何突破这种限制之前，必须指出在实际中很少有必要突破这一限制。VB 的设计者将 IUnknown 封装起来的做法是完全正确的，因为调用 AddRef 函数和 Release 函数通常会导致系统的不稳定，尤其是在调试时更是如此。

为了调用 IUnknown 函数，需要定义另一个与 stdole.IUnknown 有相同 IID 的新类型。我在文件 VBoostTypes6.olb 中提供了一个叫做 IUnknownUnrestricted 的类型。这一在 IUnknown 的改进版本中定义的类型可以很好的适用于封装的对象。例如下面的 Unknown 钩入对象和 ROT 钩入对象将使用 IUnknownUnrestricted 类型在封装的内部对象上调用 IUnknown 函数。

这一改进版本中的 AddRef 函数和 Release 函数定义，除了返回长整型(Long)而不是无符号长整型(unsigned Long)的值以外，与 VB 中的函数定义完全相同。但是 QueryInterface 不再返回 HRESULT 类型数据，而是返回一个常整型数据。这就意味着即使调用失败，VB 也不会自动引发错误。因为 VB 不能很好的处理 void**类型的数据，输出参数的类型也由

void**变为 ByRef Long。下面是两段使用 IUnknownUnrestricted 类型的小程序。我再次强调一下，不要认为我支持读者经常使用 IUnknownUnrestricted 接口。

```
'Check the current refcount of an object by reading
'the return values from IUnknownRelease. This is a
'handy debugging function, but not one you should
'rely on in production code.
Function RefCount (ByVal pUnkUR As IUnknownUnrestricted ) As Long
  'Balance the pending Release
  pUnkUR.AddRef
  'Reduce the value returned by Release down 2
  'because pUnkUR is itself holding a reference, and
  'the implicit temporary variable in the calling
  'function also holds a reference.
  RefCount = pUnkUR.Release -2
End Function

'Calling code: output is 1 for a VB class module
Dim cls1 As New Class1
Debug.Print RefCount (cls1)

'Call QueryInterface to get an IDispatch object. This is
'equivalent to setting an As Object variable to a type that is
'not derived from IDispatch
Function GetDispatch(ByVal pUnkUR As IUnknownUnrestricted) As
Object
Dim IID_IDispatch As VBGUID
Dim pvObj As Long
Dim hr As Long
  IID_IDispatch = _
    IIDFromString ("{00020400-0000-0000-C000-000000000046}")
  hr = pUnkUR.QueryInterface (IID_IDispatch, pvObj)
  If hr Then Err.Raise hr
  VBoost.Assign GetDispatch,pvObj
End Function
```

GetDispatch 函数使用的原则是：如果在编译时知道 IID(通常会如此)，就应使用 Set 语句。

第四章

绑定函数到对象上

当读者使用 New、CreateObject 或者 GetObject 这些关键字来创建一个对象时，当然希望能够调用该对象的方法或者读取它的属性。毕竟，如果我们有一个对象而不能利用它来做任何事情的话，那么该对象将毫无用处。在 VB 中我们是通过执行一种被称之为“绑定”的操作来实现这一功能的。在这里，我们将绑定定义为“根据指向对象的指针定位某一特定函数的过程”。

任何类型的绑定都使用静态的编译时数据以及对象指针的动态运行时值来寻找要被调用的函数。所有的绑定可以分为两种类型。第一类通常称为 VTable 绑定，编译器会产生一段应用一个简单公式来定位函数的代码，该公式根据对象指针和静态函数的偏移量来定位函数。第二类则被称为“IDispatch”绑定。使用这一方式时没有足够的编译时信息供利用以直接定位函数，因此编译器将控制权转交给对象的一个帮助函数，然后由这个名为“Invoke”的帮助函数来定位函数。

这个用于定位 VTable 绑定函数的公式将依赖于运行时对象虚函数表（我们常称之为 VTable）。VTable 是函数指针的一个有序数组，通常 VTable 的指针被存储在对象的第一项中。正如我们将在关于轻量对象(lightweight object)的讨论中所看到的一样，位于第一项的 VTable 指针实际上是一个 COM 对象和普通结构之间的惟一差别。下面便是给定函数偏移量，据此来确定函数指针的例子，这里的函数偏移量是由函数指针在 VTable 数组中的位置（起始位为零）与函数指针的长度（在 Win 32 中为 4 字节）相乘而得到的。

```
Function VTableEntry ( _
  ByVal pObj As Long, ByVal Offset As Long ) As Long
    With VBoost
        .Assign VTableEntry, ByVal .UAdd(.Deref(pObj), Offset)
    End With
End Function
```

```
'Calling code. 24 is the 7th function, which is
'the standard IDispatch::Invoke entry for any
'object that supports IDispatch
Debug.Print VTalbeEntry(ObjPtr(SomeObj), 24)
```

编译器从描述这一类型的类型库中提取函数偏移量。然后，偏移量与向函数提供参数和读取返回值的信息一起被编译为代码。如果去掉其中的 VTableEntry 公式，使用固定偏移量来汇编代码时，读者将会看到 VTable 绑定执行起来是十分迅速的。特别是在同下面将要介绍的 IDispatch 绑定相比较时，其快捷程度更为明显。

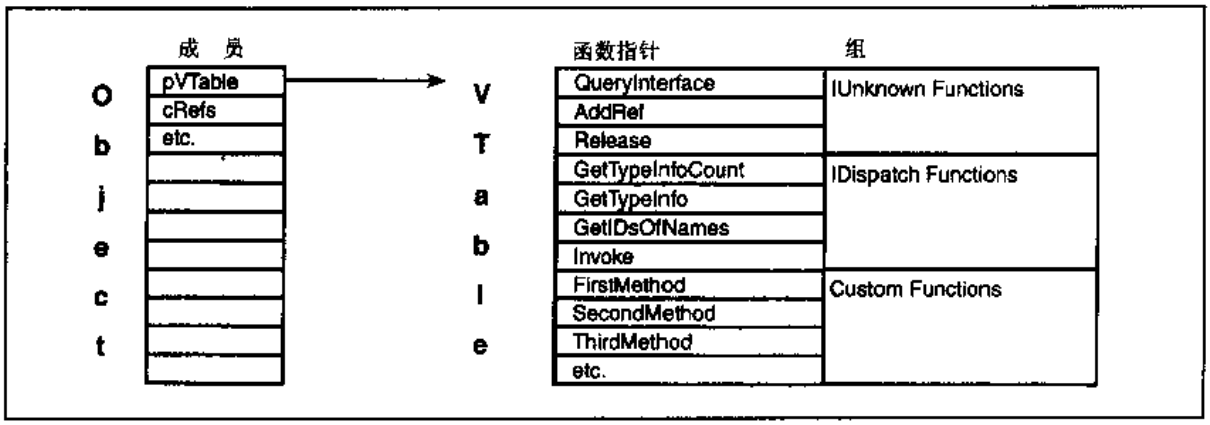

图 4.1 COM 对象的第一个成员是指向该对象的类的 VTable 的指针
VTable 本身就是一组函数指针

IDispatch 绑定也是一种基于 VTable 的机制，但在这里，VB 调用的 VTable 入口不是目标函数。IDispatch 绑定使用的 VTable 入口为 GetIDsOfNames 和对 IDispatch 接口的 Invoke 函数。这两个函数分别位于 VTable 的偏移地址为 20 和 24 字节处。IDispatch 使用一种基于 ID 的系统，每一函数都带有一个长整形标识符即 MEMBERID 或 DISPID。Invoke 函数使用 INVOKEKIND 类型的参数，这个参数控制调用函数的方式：作为一个普通函数（函数在等号符的右边或者不获取函数的返回值）、作为 Property Get（类似于一普通函数，但读者不能忽视它的返回值）、作为 Property Let（函数名位于等号左边）或者作为 Property Set（在 Set 语句中，函数名位于等号左侧）来调用。ID 和 INVOKEKIND 可以惟一的标识函数。Invoke 函数也可接收保存在一个逆序变体数组里的一个参数数组，返回数据为输出变体类型。除此之外，各种错误返回信息也可通过 Invoke 函数来获得。很明显，使用一个一般的 Invoke 函数来完成绑定比直接调用 VTable 要多做许多工作。

IDispatch 绑定可分为两类。第一类叫做“早期 ID”(“early-ID”)绑定。在这种绑定时，编译器已从类型库中获取要绑定对象的类型信息，但该对象不支持 VTable 中除 IDispatch 入口之外的 VTable。在这种情况下，编译器在编译时寻找 ID、验证要传送给函数的参数、产生智能提示(IntelliSense)列表和函数描述。实际上，在 VB 集成开发环境（IDE）中，“早

期 ID"绑定与 VTable 绑定并没有区别。读者应该检查类型库（typelib）来确定是否可以使用 VTable 绑定。在下一章中我们将会看到，目前不支持 VTable 绑定的惟一通用对象是窗体里的用户定制控件。

IDispatch 绑定的第二种类型常称为后期绑定。在这种绑定中，编译器在编译时不能提取函数的有关信息。实际上，这种情况只有在读者所要绑定的变量或者表达式返回一个变量或对象类型的数据时才会发生。很明显，当读者在代码中添加一个后期绑定的调用时，在编码时将得不到任何智能提示信息。

与早期绑定不同，后期绑定需要查询名字并调用一个 Invoke 函数。可使用 IDispatch.GetIDsOfNames 函数来将该名字翻译成 ID 值。这种名字/ID 翻译过程显著的增加了调用的花费，特别是当采用跨线程或跨进程调用时更是如此。如果在代码中使用实现关键字（在第五章中我们将对其进行讨论），就没有必要使用后期绑定。如果读者在程序中仍使用后期绑定，程序的性能将会大为降低，而且如果在后期绑定中使用硬型（hard-typed）对象变量的话，编译器将会捕捉到运行时错误。惟一的例外发生在要求运行时的名字绑定的时候。

4.1 何时绑定对象

除避免后期绑定外，读者几乎不能影响 VB 是如何将对象绑定到函数上。但是，读者可以影响 VB 何时进行绑定。很明显，当读者在对象变量或表达式和函数调用之间使用“.”操作符时，VB 就将函数绑定到对象上了。但在实际情况中，为了让 VB 产生一个对绑定函数的调用，读者并不需要在代码中添加“.”操作符。

VB 为读者自动产生绑定代码的第一个地方是在使用应用程序对象（application object）时。当类型库指定一个能够创建的类的应用程序对象属性时，VB 使用该类的名字和类型声明一个自动隐含的变量。然后，编译器试图将这一名字绑定到该隐含变量上。例如，在 DAO 对象模型中，DBEngine 类被标识为应用程序对象，因此 VB 自动为读者产生以下声明。

```
Public DBEngine As New DBEngine
```

这种隐式声明的作用范围是工程级的。编译器在解析了局部、模块级和全局范围的名字之后再解析这种工程级的名字。在下面代码段的起始两行中，库名起到名字解析符的作用，它能告知编译器绑定到哪一个库上。这使编译器不再在具有更高优先级的范围内的和比 DAO 具有更高优先级的参考库内查询名字。但是，指定这一库名并不生成其他的代码。由于 DBEngine 是一个应用程序对象，所以无论读者是否键入了 DBEngine 的名字，它总是绑定到 Workspaces 函数上。下面四行代码生成了同样的代码。

```
Dim iCount As Integer
iCount = DAO.DBEngine.Workspaces.Count
```

```
iCount = DAO.Workspaces.Count
iCount = DBEngine.Workspaces.Count
iCount = Workspaces.Count
```

VB 有它自己的应用程序对象 Global.App，Screen、Forms 和其他的对象都由 Global 的函数派生而来。当使用 App.Title 函数时，读者实际上是在使用 VB.Global.App.Title 函数，这是两个绑定调用。在处理应用程序对象时，VB 不仅生成了调用隐含的 VTable 函数的代码，而且生成了确保变量已经响应 As New 声明进行了实例化的代码。App.Title 后面的代码完成下面的任务，并在读者直接引用全局应用程序对象时重复执行这些任务。

```
Dim tmpGlobal As VB.Global
If VB.Global Is Nothing Then Set VB.Global = New VB.Global
Set tmpGlobal = VB.Global
tmpGlobal.App.Title
Set tmpGlobal = Nothing
```

自动绑定的第二种对象类型是 Me 对象。例如，把一个“命令”按钮放在窗体上并修改 Text1.Text 为“Power VB”，在这里 Text1 是一个函数调用，而不是一个变量。与应用程序对象不同的是，隐含的 Me 对象可确保已经实例化，并且它还很容易作为一个局部变量来访问。这就使将一个函数绑定到 Me 对象上比将一个函数绑定到应用程序对象所用的时间要少。

第三种隐含的类型调用一个缺省的方法或属性。让我们继续看一个 DAO 的例子，WorkSpaces 有一个 Item 对象，这就是它的缺省成员函数。Workspaces(0).Name 实际上是三个绑定调用：DBEngine.Workspaces、Workspaces(0).Item、Workspaces.Name。在键入代码时，看一下智能提示窗口，可以看到提示窗口中一些名字与代码输入窗口中光标所指示的名字不同，这些不同的名字就是缺省的成员。键入 Workspaces (打开一个标为“Item(Item) As Workspace”的窗口，将光标向左移动一个字节就会将窗口改为“Workspaces As Workspaces”)。由于在括弧前后的绑定名字不同，所以可以认为在 Workspaces 后添上一个非空的参数列表时，VB 自动的产生了一个对 Item 的调用。

程序代码的执行速度通常与程序所做的绑定调用的次数成反比。因此，在编程时要尽可能的减少绑定调用的次数，读者可以采取以下几种方法来迫使编译器尽可能的少用绑定调用。

- 要经常使用局部变量来暂存结果。VB 会在后台创建隐含的局部变量以保存函数和属性调用的返回值。读者最好也应保存这些值以备将来使用。下面的代码选择 Text1 中的所有文本。这两段代码完成相同的功能，但具体执行过程不同，第二个小程序只绑定 Text1 一次。这样的处理对只调用一次的程序没有性能上的改善，但是如果设置了 Text1 对象的 n 属性，第二种函数就减少了 n-1 次对 Text1 的调用。

```
'Use Text1 for each call
Text1.SelStart = 0
Text1.SelLength = &HFFFF&

'Use a local variable
Dim txtTmp As TextBox
Set txtTmp = Text1
txtTmp.SelStart = 0
txtTmp.SelLength = &HFFFF&
```

- 记住对一个对象使用 With 语句等同于将一个对象赋给一个局部变量。使用 With 语句可以带来许多好处：不必声明变量、减少要键入的代码并且使代码更有规则。如果使用 With 语句来访问应用程序对象，只需另外执行一次隐含的 Is Nothing 检查和方法调用，这通常发生在直接使用方法名的时候。使用 With 语句的一个弊端是同时只能执行一个 With 语句。

```
With App
   .Title = "I'm Back"
   .TaskVisible = True
End With
```

- 不要在同一函数中重复同样的一串调用，除非读者期望得到不同的结果。我每次看到这种类型的代码时，都感到十分畏惧。下面的几行代码运行起来速度非常慢，难以读懂并且难于维护。读者不会用 C++语言写出这样的代码，因为用 C++实现同样的操作要写比这长得多的代码。而在 VB 中读者只需写很少的代码，但这并不意味着 VB 会执行起来很快。

```
TreeView1.Nodes.Add , , "P1", "ParentNode1"
TreeView1.Nodes.Add _
   TreeView1.Nodes("P1"), tvwChild, "P1C1", "ChildNode1"
TreeView1.Nodes.Add _
   TreeView1.Nodes("p1"), tvwChild, "P1C2", "ChildNode1"

'Repaired code: cuts bound calls from 15 to 5
'and is a lot easier to read.
Dim ParentNode As Node
```

```
With TreeView1.Nodes
    Set ParentNode = .Add(, , "p1", "ParentNode1")
    .Add ParentNode, tvwChild, "P1C1", "Child1Node1"
    .Add ParentNode, tvwChild, "P1C2", "ChildNode1"
End With
```

减少绑定调用可在三个方面对读者有所帮助。首先，VB 不必多次产生绑定代码。其次，读者调用的对象不必重复的运行同样的代码。最后，代码的长度也会显著的减少。在许多情况下，代码的长度是影响程序性能的最大因素。由于将代码载入内存的代价通常是很昂贵的，所以载入的代码越少，程序的性能就会越好。

我并不是说对同一线程里对象作 VTable 绑定调用执行起来性能会很差，并且不认为如果读者调用的次数过于频繁的话，调用将会成为程序的“瓶颈”。实际上，VTable 调用的性能十分优良。但是，如果在一个程序中作上成千上万个绑定调用，调用的时间就会累积起来。读者只需简单的使用局部变量和 With 语句，并且不重复同样的代码，就会发现整个程序的性能会大为改观。

4.2 运行时间的名字绑定

在某些特定的应用程序中，需要对一些编译时不存在的对象进行编码。例如，一个数据驱动的应用程序会动态的创建对象，并确定数据库或对象本身的方法和属性，然后通过 IDispatch 调用对象。尽管这种做法十分灵活，但它是以性能的恶化为代价的：无编译检查、大量的附加代码。（见 1999 年十一期 Visual Basic Programmer's Journal 中的一篇名为“Inspect Dynamic Objects”的文章）

VB6 提供的 CallByName 函数允许做真正的运行时调用，但使用这种函数一段时间后，会发现许多问题。这种调用的任何参数都是在 Args 参数中传送，其中 Args 参数是 ParamArray 类型的数据。这意味着需要知道尚未定义的方法中所包含参数的个数。但是如果已知参数的个数，读者就已经知道了参数的名字并且不会首先使用名字绑定。

CallByName 函数也要求每次都传递同样的名字而不是传递 MemberID。在多次调用同样的成员函数时，由于每次调用时都使用 GetIDsOfName 将名字翻译为 MemberID，时间花费将增加多倍。并且调用另一不同的线程、进程或机器里的对象是十分费时的，没有十分必需的理由却将调用的次数增加数倍是一种很不明智的做法。

IDispatch 接口并不是一个对 VB 十分友好的接口，但也可以考虑调用它。IDispatch 的接口和函数都是限制性的，因此在 VB 中既不能声明一个 IDispatch 类型的变量，也不能调用它的函数。除此之外，许多在 stdole2.tlb（OLE Automation）中定义的类型都对 VB 不是十分友好的，并且需要调用函数指针来填充一些错误信息。

VBoostTypes 库通过定义 IDispatch 接口和对用户友好的 EXCEPINFO 和 DISPPARAMS 来解决所有的声明问题。有一个定义的函数指针原型可以延迟填充详细的错误信息。（见第十一章“调用函数指针”）

4.2.1 Call ByName 的替代

使用在 VBoostTypes 中的 IDispatchCallable 定义，我编制了 CallByName 的替代代码。第一个函数是 GetMemberID，它可将名字翻译为一个 MemberID 并将其传递给 CallInvoke。CallInvoke 除了接受的是 MemberID 而不是名字之外，很类似于 CallByName；它的参数也必须以 IDispatch.Invoke 所要求的逆序排列。以后将介绍三个 CallInvoke 版本。这里列出的 GetMemberID 和 CallInvoke 函数可在本书附带的光盘中看到。它们在 CallDispatch.Bas 文件中，读者应将 FunctionDelegator.bas 添加到自己的文件中。

```
'Shared variables and constants
Private GUID_NULL As VBGUID
Private Const DISPID_PROPERTYPUT As Long = -3
Public Const VT_BYREF As Long = &H4000
Public Enum DispInvokeFlags
    INVOKE_FUNC = 1
    INVOKE_PROPERTYGET = 2
    INVOKE_PROPERTYPUT = 4
    INVOKE_PROPERTYPUTREF = 8
End Enum

'GetMemberID Function, calls GetIDsOfNames
Public Function GetMemberID( _
  ByVal pObject As Object, Name As String) As Long
Dim pCallDisp As IDispatchCallable
Dim hr As Long
   VBoost.AssignSwap pCallDisp, pObject
   hr = pCallDisp.GetIDsOfNames( _
     GUID_NULL, VarPtr(Name), 1, 0, VarPtr(GetMemberID))
   If hr Then Err.Raise hr
End Function

'CallInvoke Function calls Invoke
```

```
Public Function CallInvoke( _
  ByVal pObject As Object, _
  ByVal MemberID As Long, _
  ByVal InvokeKind As DispInvokeFlags, _
  ParamArray ReverseArgList() As Variant ) As Variant
Dim pSAReverseArgList() As Variant
  'Swap the ParamArray into a normal local variable to enable
  'passing it to CallInvokeHelper. This is done by reading
  'the VarPtr of the stack's previous variable.This is
   'possible because InvokeKind is a simple type and passed
   'ByVal, so its VarPtr points to the area of
   'the stack used for the initial incoming function
   'parameters. For ByVal String and Object types, VarPtr
   'refers to a local variable that is a copy or reference
   'to the location on the stack, so you can use this
   'trick only for normal parameters. This is required because
   'VB doesn't let you pass a ParamArray to a normal array,
   'including the array parameter defined by VarPtrArray.
   With VBoost
     .AssignSWap _
       ByVal VarPtrArray(pSAReverseArgList), _
       ByVal .Deref(.UAdd(VarPtr(Invokekind),4))
   End With
   'Call the helper with pVarResult set to the address
   'of this function's return value.
   CallInvokeHelper pObject, MemberID,Invokekind, _
     VarPtr(CallInvoke),pSAReverseArglist
   End Function

   Private Sub CallInvokeHelper(pObject As Object, _
     ByVal MemberID As Long, ByVal Invokekind As Integer, _
     ByVal PvarResult As Long, ReverseArgList() As Variant)
   Dim pCallDisp As IDispatchCallable
   Dim hr As Long
   Dim ExcepInfo As VBEXCEPINFO
   Dim uArgErr As UINT
```

```
Dim FDDeferred As FunctionDelegator
Dim pFillExcepInfo As ICallDeferredFillIn
Dim lBoundArgs As Long
Dim dispidNamedArg As DISPID
Dim Params As VBDISPPARAMS
    'Fill the fields in the DISPPARAMS structrue
    lBoundArgs=LBound(ReverseArgList)
    With Params
        .cArgs = UBound(ReverseArgList) - lBoundArgs + 1
        If .cArgs Then
            .rgvarg = VarPtr(ReverseArgList(lBoundArgs))
        End If
        If Invokekind And _
            (INVOKE_PROPERTYPUT or INVOKE_PROPERTYPUTREF) Then
            dispidNamedArg = DISPID_PROPERTYPUT
            .cNamedArgs = 1
            .rgdispidNamedArgs = VarPtr(dispidNamedArg)
            'Make sure the RHS parameter is not VT_BYREF.
            VariantCopyInd ReverseArgList(lBoundArgs), -
                        ReverseArgList(lBoundArgs)
        End If
    End With

'Move the incoming variable into a type we can call.
VBoost.AssignSwap pCallDisp, pObject

'Make the actual call
hr = pCallDisp.Invoke(MemberID,GUID_NULL, 0, _
    InvokeKind, Params, pVarResult, ExcepInfo, uArgErr)

'Handle errors
If hr = DISP_E_EXCEPTION Then
    'ExcepInfo has the information we need
    With ExcepInfo
        If .pfnDeferredFillIn Then
            Set pFillExcepInfo = _
```

```
                InitDelegator(FDDeferred, .pfnDeferredFillIn)
            pFillExcepInfo.Fill ExcepInfo
            .PfnDeferredFillIn = 0
        End If
        Err.Raise .scode, .bstrSource, .bstrDescription, _
            .bstrHelpFile, .dwhelpContext
    End With
    ElseIf hr Then
        Err.Raise hr
    End If
End sub
```

可参考 MSDN 得到对调用 IDispatch 的细节的解释。但是，对 CallInvoke 中的两行代码还需做进一步的解释。第一行使用 VBoost.AssignSwap 将新来的 Pobject 变量传递给 pCallDisp 变量。正如在第三章的“Set 和 Unknown”中介绍的那样，对 IDispatch 使用 QueryInterface 有恼人的负面效应。尽管 IDispatchCallable 和 Object 有相同的 IID，它们是不同的类型，但用 Set 语句赋值会导致对 IID_IDispatch 使用 QueryInterface 函数。

第二行代码不太常见，它是 VariantCopyInd API 调用。当第一个参数传递给 INVOKE_PROPERTYPUT 函数(在 VB 中为 Property Let)或 INVOKE_PROPERTYPUTREF 函数（Property Set）时，需要做这一调用。绝大多数的 IDispatch 实现不能将一个 ByRef 参数传递给数值属性，因此要对数据做完全拷贝。如果传送来的参数不是 ByRef，VariantCopyInd 不做任何操作。

4.2.2 CallInvoke 的不同版本

正如前面指出的那样，存在三种 CallInvoke 版本：CallInvokeSub、CallInvokeArray 和 CallInvokeSubArray。CallInvokeSub 是一子程序(Sub)而不是一个函数并且不需要返回值。绝大多数 IDispatch 实现通过清空返回值来处理对子程序的返回值请求。但是，在有些情况下一些 IDispatch 实现会失败，因此需要一个不要求返回值的特定机制。这些函数的*Array 变量从 ParamArray 最后一个参数中移出 ParamArray 关键字。除了不需要特定的代码将数组提交给 CallInvokeHelper 外，这些函数几乎与 CallInvoke 相同。

这些函数的数组版本允许传递任意数量的参数，这主要是通过动态调整参数的个数来实现的。但是，读者不应忽视 ParamArray 的一个关键特征：传递 ByRef 变体的能力。当读者在对 ParamArray 函数的调用中传递一个非变体类型时，VB 传递一个设置了 VT_BYREF 标志的变体。VT_BYREF 与所指向的数据的类型相结合，然后 VB 存储指向数据的指针，而不是存储数据本身。如果赋值给一个变体数组，VB 复制数据并且去除 VT_BYREF 标志。

这就表明使用带 ByRef 参数的 CallInvokeArray 函数要做一些额外的工作。

```
'call a sub named "Simple" With1 ByRef Long parameter
Dim Args(0) As Variant
Dim pObj As Object
Dim lData As Long
    Args(0) = VarPtr(lData)
    VBoost.Assign Args(0), VT_BYREF or vbLong
    CallInvokeArray pObj,GetMemberID("Simple"), _
        INVOKE_FUNC, Args
```

VT_BYREF 存在一个很有意思的锁定变体的类型的副作用。当 VB 赋值给 VT_BYREF 变体时，它总是试图将包含的数据转化为正确的类型，如果不能完成转化则赋值失败。然后 VB 释放现在指针指向的数据（由类型决定的）并给指针赋予新值。VT_BYREF 标志不发生改变，指针指向的内存空间通过变体来更新。这里要注意的一点是 VB 假设所有的局部变量和模块级变量都不是 VT_BYREF 类型的，因此它只检查变体参数上的 VT_BYREF 标志。

4.3 VTable 绑定用户定制控件接口

以前曾提到过窗体上的用户定制控件只支持 IDispatch 绑定。但是，绝大多数 OCX 文件中的控件对象都支持 VTable 绑定。之所以要深入控件内部，寻找 VTable，是基于下面两个原因。首先，将 IDispatch 绑定变为 VTable 绑定可以使性能显著增加。其次，VB 使用的特定的 IDispatch 实现是由开发者机器里的 OCA 文件动态产生的，这时 IDispatch 绑定是一个基于特定机器的实现。如果要在组件间传递控件并在别的机器上使用这些控件，就不能使用私有的 IDispatch 接口。因此除非读者要完全依赖于后期绑定帮助组件，否则就都要像在 OCX 中定义的那样得到一个对真正的 VTable 引用。

以前介绍了如何访问 VTable，读者应该对 VB 如何处理用户定制控件有了进一步的理解。VB 中的控件定义和 OCX 中的控件定义不同。如果在对象浏览器下拉列表中选择一个控件库，描述面板上显示了一个对 OCA 文件的引用，而对 OCX 文件的引用在这里没有找到。

OCA 文件包括对象浏览器中的类型库和一些其他的数据，它是在添加用户定制控件到工具箱时动态产生的。OCA 类型库中描述的对象包括 CCX 的固有控件的所有属性、事件和方法。但是，现在这个固有的函数集已经扩展，能够包含 VB 控件容器所特有的一些属性、方法和事件。例如，读者可以看到每个放置在窗体上的控件的 Top 和 Left 属性。要看

到 VB 可能添加的项的一个超集，可在对象浏览器中选择 VBControlExtender 对象。

类型库中描述的扩展对象是作为固有对象和 VB 提供的对象的 IDispatch 封装在运行时实现的。扩展对象是一个复合体，它将传送来的 Invoke 调用交给固有 OCX 的 IDispatch 实现或者 VB 的控件扩展对象。在这里不需要 VTable，并且固有对象也是通过 IDispatch::Invoke 来接受请求的。

当以用户的速度（击鼠标键、敲键盘、思考或喝茶的速度）响应事件时，由 VB 的扩展对象实现的双重 Invoke 系统工作情况良好，但当调用代码对控件作多次调用时，系统的性能就会出现问题。例如，如果读者处理一个产生图形的控件或填充一个网格，一旦固有的 OCX 取得了控制权，单个调用就会很快，但调用是所有 IDispatch 任务的瓶颈。就控件来说，直接调用 VTable 可以使运行时间减少至少两个数量级。（我测量的为一百三十分之一）。读者所要做的所有工作就是在封装的扩展对象中找到 VTable。

VB 的扩展对象具有对象属性，它可以返回一个对未封装的内部控件的引用。得到对象的引用是正确解决问题的第一步，但是由于对象属性返回对象类型，这里可能要碰上麻烦：读者能做的就是对其做后期绑定。这一练习的要点是得到一个支持 VTable 的对象引用。但是这个引用的类型是在 OCX 类型库中定义的，您的工程并没有引用这一个类型库。实际上，读者甚至不能使用 Project\References\Browse 来给添加一个对 OCX 的引用，因为 OCX 可能与 OCA 有相同的库名。即使在库名不同的情况下，读者也会得到两套相似的对象名，这会给人带来许多令人困惑的东西。

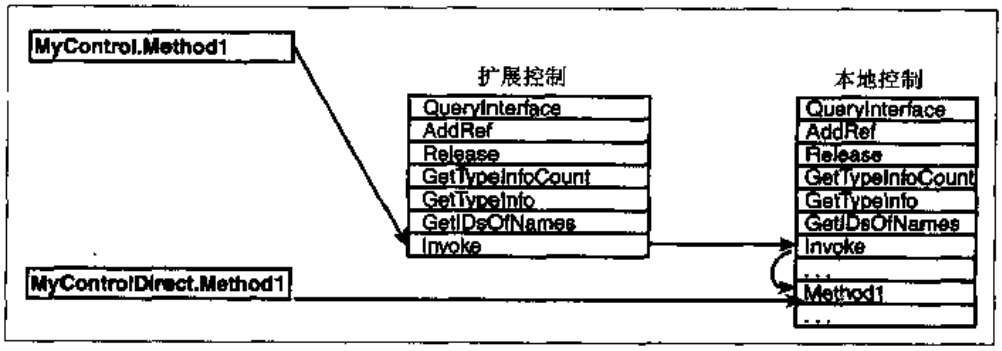

图 4.2 绕过两个 Invoke 间接调用，对一个控件方法进行直接路由

4.3.1 绑定方法

为了对由对象属性返回的固有对象作 VTable 绑定，工程包括对一个类型库的引用，运行时控件将这一个类型库中的类型看做固有接口。使用这本书中包括的一个插件 OCXDirect.Dll，可以自动的产生一个这样的类库。OCXDirect.Dll 遍历工程中的所有引用，寻找 OCA 文件，确定哪一种类型已被扩展并创建一个包含固有 OCX 文件中定义的类的别名的类型库。步骤很简单。

（1） 将 OCXDirect.DLL 复制到机器上并用 regsvr32 注册该文件。

（2） 在插件管理器（Add-In Manager）对话框中，选择“Direct OCX 类型产生器(VB6)”(Direct OCX Type Generator(VB6))。

（3） 保存工程。

（4） 在插件（Add-Ins）菜单中，选择“生成 Direct OCX 类型”（Generate Direct OCX Types）插件自动生成一个叫作 *ProjectName*DirectOCXTypes.tlb 的文件并将该类库添加到读者的工程引用中。

（5） 改变引用的控件时，重复步骤 4。Tlb 文件将被删除并被重新生成。

对于工程中引用的每一个扩展控件，读者现在都有了一个与添加到后面的 Direct 同名的类型。例如，如果 RichTextBox1 是窗体上的一个控件，要对它作 VTable 绑定，应运行插件并添上下面的代码。

```
Dim RTDirect As RichTextBoxDirect
Set RTDirect = RichTextBox1.Object
'RTDirect is a direct reference to the native control
```

现在将*Direct 类型定义为固有 OCX 的简单别名。手工生成这样的库并不是很困难，看一下它的代码是很有用的。下面是库模块的正文，它为每一个库添加一个引入（import）并为每一控件添加一新的类型定义(typedef)行。

```
importlib("c:\\winnt\\system32\\RichTx32.Ocx ");
typedef [public] RichTextLib.RichTextBox RichTextBoxDirect;
```

当 VB 编译器看到像这样由 OCXDirect 产生的那样的别名类型时，它就会解析类型定义行，然后忽视开头的“typedef”。这对用户是很有利的，可以在一个公有函数中使用 RichTextBoxName 名作为参数或返回类型。当 VB 编译组件时，它去除中间类型，生成的类型库使用与 RichTx32.Ocx（而非 ProjectNameDirectOcx.Olb）的外部关联（dependency）来构建。Direct 类型就像性能良好的催化剂：它能触发反应并且得到的反应后化合物中不留半点自己的痕迹。

借助于一个对在一个实际的、分布式的 OCX 中（而不是在局部定义的 OCA 文件中）定义的类型的引用，读者可以创建一个接受运行控件的引用的可配置组件。按下面的步骤可以创建一个帮助动态连结库(DLL)，它可以对 RichTextBox 控件做一些标准的处理。

在帮助动态连结库中，选择 Project|Preferences|Browse，打开 RichTx32.Ocx。控件库不能作为普通引用来显示，因此读者必须手工寻找它们。注意：读者可以添加这些隐含引用，但当再次打开对话框时，这些隐含引用并不显示在当前引用的列表中。所以，必须手工的编辑 VBP 文件并重新载入工程以去除这些引用。

帮助动态连结库中的一个方法应该以按值(ByVal) RichTextBox 的方式来接收控件引用。在这里，RichTextBox 被解析为 OCX 控件中的固有类型。

要从读者的 UI 工程中传递一项给帮助动态连结库的方法，应使用 RichTextBox1.Object。

如果要在 UI 工程中包含一个公有函数（假设读者有一个动态连结库，它可显示一个带有 RichTextBox 控件的窗体），应生成 direct 类型并定义参数或返回值类型为 RichTextBoxDirect。从外部看来这样产生的类型与帮助动态连结库中可见的 RichTextBox 相同。

读者现在可以在组件间来回传递控件的固有接口，但如果要看整个控件而不是仅仅它的固有方法、属性和事件时，应该怎样去做呢？这可以通过使用控件支持的 IOleObject 接口以从固有对象中返回到扩展控件中去。下面的例子包含一个带有 RichBox 的窗体。工程的 Direct 类型库已经生成，并且该工程还引用了本书提供的 OleTypes(VBoost:OLE 类型定义)类型库。

```
Private WithEvents SyncRT As RichTextBoxDirect
Private WithEvents SyncTRExt As VBControlExtender

Private Sub Form_Load()
Dim pOleObject As IOleObject
Dim pOleControlSite As IOleControlSite
    'Get the native object and sync its events
    Set SyncRT = RichTextBox1.object


    'Get the IOleObject interface from the
    'native object. IOleObject gets us back
    'to the client site, which can then give
    'us the original extended object.
    Set pOleObject = SyncRT
    Set pOleControlSite = pOleObject.GetClientSite
    'Sync the extender events.
    Set SyncRTExt = pOleContrclsite.GetExtendedControl
    'Demonstrate that we are back to the original
    'control. This prints the same value twice.
    Debug.Print ObjPtr(SyncTRExt), ObjPtr(RichTextBox1)
End Sub
Private Sub RichTextBox1_Change()
    Stop
```

```
End Sub
Private Sub SyncRT_Change()
    Stop
End Sub
Private Sub SyncRTExt_ObjectEvent(Info As EvenInfo)
    If Info = "change" Then Stop
End Sub
```

Form_Load 结束时会存在三个对象：原先的 RichTextBox1 控件引用、对固有对象（SyncRT）的直接引用和对扩展对象的引用。在一个普通的工程中，现在可以继续用 WithEvents SyncRT 对象接收固有事件并且使用 SyncRTExt 接收扩展事件。但是，VBControlExtender 有一名为 ObjectEvent 的特殊事件，这一事件在固有事件发生时发生。通常可以直接访问固有函数事件原型，并不需要使用这一事件。我在这里提出这一事件是为了表明控件中存在三种活动的事件。如果对 RichTextBox1 作改变，所有三个事件都会发生。

我们从上面这个例子中学习到的重要的一点是不使用 OCA 定义的 RichTextBox 类型，就可以访问固有或扩展的方法、属性和事件。而且，读者可以编写一个能够处理控件的固有部分以及扩展部分的帮助动态连结库。更进一步，读者应该让帮助动态连结库与自己机器相脱离，将它包含在产品中。

4.3.2 OCA 文件中的伪记录类型

必须指出：OCA 文件中存在一个微妙但却是致命的错误。这一错误让读者不能在 VB 生成的 OCXs 中使用公有的用户自定义类型。当 VB 生成 OCA 文件时，它会产生 OCA 中所有公有记录类型的完全拷贝。尽管这些类型与固有 OCX 中的原始类型有相同的 GUID 和布局（layout），但编译器并不认为它们是相同的类型。

读者通过生成一个带有公有记录类型的 OCX，就可以清楚地看到这一问题。控件应该包含一个使用这种类型的方法。构建该控件并把它放入另一个工程，然后生成 Direct 类型。当读者试图传递一个 OCA 定义的 UDP 给一个 OCX 定义的方法时，编译器给出永久类型不匹配的信息。

这一错误也可能发生在发布控件的时候，这时的错误要更严重一些。我们需要一个类型库来描述公有 UDT；但不幸的是，包含 UDP 的类型库存在于开发者机器上的 OCA 中，而不是存在于用户机器上的 OCX 中。除非读者使用 OCX 来注册 OCA，否则所有的地方都会得到类型不匹配的错误信息。

我所了解的修正这一错误的一个实用程序就是本书中包含的 OCARecordFix.exe 文件。读者要使用这一工具程序，应该使用对象浏览器来寻找有错误的 OCA 文件。现在退出 VB

并运行“OCARecordFix <Oca 路径名>”。这一个实用程序会用 OCX 中的直接类型定义代替 OCA 中的类型重定义。而且 OCARecordFix 还改变所有参数和一些方法返回值的类型。编译器认为类型定义和目标类型是等价的，所以在重新载入 VB 时，错误就已经消除了。OCARecordFix 需要不被 Windows 95 或 Windows 98 支持的资源 API，因此它在这些操作平台上不能使用。这一错误也已经在 Visual Studio service pack 4 中得到了修正。

第五章

对象的设计结构

代码的结构和设计是一个软件工程成功的基本要素。在软件开发的后期，读者已经写出大量的代码，但是如果设计做得不好的话，读者所做的工作实际上是试图加固一个内核不稳定的系统。确定代码结构的困难之处在于代码写出之前，无法证明代码结构的有效性。尽管在生活中很少有可以确保正确的事情，但是根据在过去被证明行之有效的设计原则来确定代码的结构，我们可以将系统由于自身的原因而崩溃的可能性降到最低限度。

本书的目的不是深入的探讨代码的结构。相反，本书将重点介绍一些经过时间考验证明的有效的设计原则，并介绍这些原则在 VB 中的应用。本章重点介绍工程内部设计方法。不同工程建立的二进制 EXE 文件、DLL 文件和 OCX 文件的重用性和兼容性将在本书的后面予以介绍（见第十五章“二进制兼容性”一节）。

- 可插入（pluggable）组件。替换或更新工程中的一部分代码而无需修改其余代码的能力是十分重要的。我们都曾经经历过连锁反应：对一部分代码的微小修改影响到了整个工程。除了减少与连锁反应有关的毛病外，要尽可能设计可插入组件，这样可使新代码易于测试。如果改变一个组件会使调用该组件的代码发生改变，那一旦出现问题，我们就无法判断是调用者还是被调用者出现了问题。但是如果修改了可插入组件内一个等式的变量，在发生错误时我们就能准确的知道错误发生在哪里。并且，从当前的工程中取出可插入组件以备将来重用是很容易就能做到的事情。
- 抽象。抽象是实现“可插入性”的核心方法，因为同一抽象对象能生成多种实现而无需改变抽象对象的定义。调用代码是仅仅需要考虑抽象对象，而不用考虑抽象对象的各个实现。从调用代码的角度来说，同一抽象的两种实现完全可以相互替换。在 COM 中，对象的抽象描述称为接口，在这里接口就是一些方法的有序列表。接口的方法与实现这一接口的对象的 VTable 方法一一对应。所有对象都是抽象接口的具体实现并且外部代码只有通过接口才能访问 COM 对象。
- 代码重用。代码的稳定性与所写的代码的行数成反比。如果读者有一段正常运行

并经过反复测试的代码，应在将来尽量重用这段代码。在这里我并非指的是代码重用的剪切和粘贴机制（也称为“错误的快速繁殖机制”），而是在一些不同的地方使用同一实现。代码重用经常与可插入性和抽象性发生矛盾。它与可插入性相矛盾是因为当代码重用时，新的代码通常有新的要求，需要添加新的参数到现在的方法上去（有可选参数可好些）。由于抽象对象只有描述、没有代码，所以在一些对象上支持这一接口就要求在多个地方使用同一接口定义的一个实现，在这里，抽象性与代码重用发生了冲突。本章的大部分内容都是试图在抽象和代码重用之间做出折衷。

5.1 使用 Implements 来实现抽象

我在第三章的“Set 语句和 IUnknown 接口”中曾经谈到过，COM 借助于 QueryInterface 方法来提供多重接口支持。VB 允许使用实现（Implements）关键字来扩展 QI 能够响应的一组接口。实现关键字允许读者的对象提供一个抽象接口的一个实现。

使用实现关键字就像添加一个事件过程到窗体上一样简单。在类模块的顶部键入“Implement <interface>”并提交该行。在左下拉列表里有带有接口名字的入口。在左下拉列表中选择入口并在右下拉列表中选择名字，就可以给接口中的每一项插入函数原型。要想编译成功的话，读者必须为接口中的每一项插入程序代码。

在选择接口中的各项时，VB 将会以“Interface_Function”的名字插入一些私有程序。如果使用一个普通的函数名来实现两个接口，我们借助名字的格式可以区分出这两个接口。这一点是于分重要的，它使读者能在一个类中支持任意多的接口而不会发生名字冲突。（C++和 Java 只在名字基础上进行绑定，当要添加接口时就会发生名字冲突，它们分辨不出 Name 函数是通过 IFoo 接口还是通过 IBar 接口访问的）。

将实现的函数标为私有函数意味着它们不可以通过主接口来访问。如果喜欢的话，读者可以将这些函数设为公有函数，但我并不提倡这样做。一旦将函数设为私有，读者仅能通过定义函数的接口来调用该函数。下面我要重点介绍的设计原则都受益于这一特征：每一接口都惟一的代表从外部看待对象的一个角度。如果要从另一角度来操作对象，就只能转换到另一个接口。（见 “调用代码的可插入性”一节）

为了实现一个接口，首先要在工程中对它进行定义。读者在自己的模块中既可以使用类型库中定义的接口，也可以使用程序中另一个类的主接口。VB 类的主接口是自动进行定义的，以与类模块中的公有属性和函数相匹配。通常，我建议读者使用类型库中定义的接口，但选择方式并不是一件很容易的事情。当读者决定在哪里定义接口时，要考虑下列问题。

- VB 不区分作为接口定义来创建的类和真正的类。尽管用来定义接口的类模块仅仅包含空程序，但是 VB 会自动提供类的实现。也就是说，在程序中最后也只是有一

些“死代码”。

- VB 类都是由 IDispatch 派生而来的。如果使用接口来通信，就不会用到 IDispatch。强迫 VB 提供一个 IDispatch 实现，但在以后不使用它，这是在浪费时间和资源。如果用脚本语言来访问接口，那就要求接口支持 IDispatch。否则的话，这样做只会产生一堆垃圾。
- 如果在公有类中实现接口并且这一接口是这一公有类的主接口时，必须设置这一 VB 类的实例属性（Instancing property）为 PublicNotCreatable 或更高的级别。 VB 本应该提供一个私有实现以保证在工程的类型库中访问不到专门属于这一工程的接口实现，但是 VB 没有做到这一点。
- 如果要使用 Implements 但又不允许从外部访问接口，就应将定义放于类型库中并且不要重新发布外部库。只要读者不跨线程使用接口，这样做就已经有效的“私有化”了接口。VB 创建的类型库对读者的私有类型库有一个未被解析的引用，它会影响到库的完整性。如果对这个问题感到困惑，请参考第十五章的相关介绍，在那里我们将讨论如何去除这个接口入口。注意：在 VB5 中，客户定制控件中那个未被解析的接口引用将导致一个无效的 OCA 文件的产生。在 VB5 中惟一的解决办法是将私有类型库传递给使用或修改读者的控件的人员，并由他们来注册该类库。否则的话，只有升级了。
- 如果要跨过程来使用一个接口，必须在目标机器上注册该接口，那就意味着要使用读者发布的软件来在目标机器上配置和注册这个类型库。如果在工程中使用 PublicNotCreatable 类来定义接口，就不需要外部库。读者也可以使用 PowerVB 后期构建类型库修改符插件来将外部类型库引入读者的可执行类型库中。这种插件允许传递单个二进制位。

尽管 VB 类是从 IDispatch 接口（由 IUnknown 继承而来）的实现继承而来的， 但 VB 不支持从读者所写的代码中继承类。继承是一种通用的面向对象程序设计（OOP）结构，它允许一个类从另一个类中得到接口的实现。提供实现的类称为基类（base class）。继承也允许重载基类的某一实现中的特定函数。函数重载允许在函数-函数的基础上提供接口的用户实现，而不是为接口中的每种函数提供一种实现。一些语言，例如 C++，允许多重继承，这使读者能为多个接口派生出实现。另一些语言，例如 Java，只允许单重继承。VB 不支持继承，所以在 VB 中重用实现时，需要做一些创造性的工作。

尽管在许多情况下，如果 VB 中存在继承，一些问题处理起来会更容易些，但是基于继承的系统的运行情况也不是很好。为了讲明“草未必尽绿”这个道理，我下面将花几分钟讨论一下继承存在的弱点。

第一个问题称为脆弱的基类。当使用继承时，经常出现多个类继承自同一个基类的情况。这时基类如果发生改变，由它继承的子类的行为也会随之改变。用户的开发部经理不会在交货期的前两个星期内让你修改基类，那样会导致整个系统的不稳定，所以用户只有重载基类的许多（如果不是全部的话）函数。这就带来了其他的一些问题，特别在出问题

的基类和当前类之间有几层基类时更易出现问题。Java 的 final 关键字并没有解决这一问题，它只是迫使你定义一个必须由基类和当前类实现的接口。在这里，final 关键字使问题变得更难以处理。

第二个问题是函数调用目标的不可预料性和不同实现间的相互作用的不可预料性。由于在接口定义中的每一个函数都可由不同的类来实现，所以在每一函数中，调用代码并没有显示正在调用哪一个类。在基于接口的模型中，同一对象实现所有的函数。这时的代码更容易预料，避免发生在子类重载父类的中一部分（并非全部）函数时出现的一些微妙的问题。下面将会看到，解决这一问题的另一种方法是允许一个对象提供多个不同的实现，因为能提供多个行为的一个类比多个类更易于集中管理，也更易于预测。

继承存在的最后一个问题（至少是我们讨论的最后一个问题）是版本问题。当升级到高一级的版本时，通常要给基类添加一些新特征和新能力。这时，不可避免的与继承自基类的子类发生名字冲突。毕竟，读者在基类中添加 Validate 函数的最可能原因是多个继承自该基类的子类中已经有了该函数。通过在父类中添加 Validate 函数，可将 Validate 这一概念更加标准化。修改基类就会将子类中的方法变为重载方法，即使它们无意做这样的改变。

当使用一个组件提供同一接口的两个版本时，会遇到更多的有关继承和版本的问题。如果读者要求在支持基接口的老版本的同时也支持基接口的升级版的话，新的基类必须继承自老的基类。但是如果在改变基类时就会遇上麻烦（特别是在单继承系统中），因为继承自基类一的类参与到老系统中去，而继承基类的类参与到新系统中去。

5.2 调用代码的可插入性

作为程序员，要花费大量的时间和精力来编写类的代码。如果我们精心编写了类的代码，我们使用新的调用代码就可多次重用这些代码。毕竟，我们花费了至少与编写代码相等的时间来编写对类的操作代码。有关编程的书往往着力于介绍如何重用类，而忽视了对调用代码的说明。我想说明的是在设计类时，调用代码的重用性和类的重用性一样重要，也可节省大量的时间。

可在 VB6 的 PropertyBag 对象中找到许多有关调用代码重用的例子。读者可用下面的例子保存所有 Persistable 属性为 1（也就是持久对象）的公有类的当前状态。持久（Persistable）属性告诉 VB 为 VB 类提供 IPersistStream 接口的一个实现。PropertyBag 对象使用 IPersistStream 产生一个单字节数组来反映对象的当前状态和对创建该对象的一个新实例有用的信息。

```
'Generate a Byte array that contains the current
'State of the passed class.
Function GetClassBytes(ByVal AnyClass As Object) As Byte()
Dim PB As New PropertyBag
```

```
Dim bArr() As Byte
    With PB
        .WriteProperty "Dummy", AnyClass
        bArr = .Contents
    End With
    GetClassBytes = bArr
End Function

'Create a New instance of the class with the state
'written to the ClassBytes array by a previous instance.
Private Function RestoreClass(ClassBytes() As Byte) As Object
Dim PB As New PropertyBag
    With PB
        .Contents = ClassBytes
        Set RestorClass = .ReadProperty("Dummy ")
    End With
End Function
```

因为 PropertyBag 代码中的类都支持 IPersistStream 接口，所以这段代码可重用。通过在代码中使用实现并遵循一些简单的规则，可以不改变调用代码对多个不同的类进行操作。同样重要的是，读者还可以不用重写调用代码就替换整个类。这种不用改变调用代码就可即插即用（plug-and-play）类的实现的能力对于代码维护是十分重要的。除了可以大大减少重写的代码的数量之外，读者也为新的执行代码创建了一个内置的测试点。

要求一段操作特定接口的代码能够被重用，读者应该在每一个特定程序中只对一个接口进行操作。通过将代码分为几个自治单元，其中每一个单元处理一个特定的接口，读者可大幅度提高代码的可重用性。例如，假设已经定义了 IReport 接口和 IOutput 接口。读者是通过几个类来实现 IReport 接口和 IOutput 接口的。你可以写类似下面的非重用代码。

```
'clsOrders implements IReport
'clsRecordToFile implements IOutput
Sub OutputOrders(Orders As clsOrders, strOutFile As String)
Dim FileRecord As clsRecordToFile
Dim IReprt As IReport
Dim IOutput As IOutput
    'Initialize output class
    Set FileRecord = New clsRecordToFile
```

```
    FileRecord.File = strOutFile

    'Bring Orders up to date
    Orders.Freeze
    Orders.Reconcile

    'Output orders
    Set IReport = Orders
    Set IOutput = FileRecord

    'Output processing goes here

    'Clean up
    FileRecord.Close
    Orders.Unfreeze
End Sub
```

在这个程序中，任何对 IReport 和 IOutput 接口操作的代码都是不可重用的。要想重用代码，只有进行剪切和粘贴操作，而这会对读者的代码维护造成很坏的影响，因为在粘贴代码的同时，也会将错误复制过来。

将这个程序一分为二，它立即变为能重用的代码。OutputReport 代码仅仅知道 IReport 和 IOutput 接口，并且它是通过传递参数而不是通过 Set 语句获取这两个接口。应注意到输入参数是按值(ByVal)传递的，以便消除程序返回时的另外两个 QI 调用。

```
Sub OutputOrders(Orders As clsOrders, strOutFile As String)
Dim FileRecord As clsRecordToFile
    'Initialize output class
    Set FileRecord = New clsRecordToFile
    FileRecord.File = strOutFile

    'Bring orders up to date
    Orders.Freeze
    Orders.Reconcile

    'Output orders
    OutputReport Orders,  FileRecord
```

```
    'Clean up
    FileRecord.Close
    Orders.Unfreeze
End Sub
Sub OutputReport(ByVal IRep As IReport, _
                 Byval IOut As IOutput)
    'Output processing here
End Sub
```

5.3 实现和实现重用

实现关键字对于创建重用的调用代码是十分有用的。因为实现关键字提供了一个构建复杂应用程序的稳定框架，在这个程序中各模块通过抽象接口定义来进行通信，所以它是一个很好的设计工具。实现的主要缺点是它没有提供固有的代码实现，因此尽管调用代码可重用，类模块代码却不能。当然，这也是 COM 接口的主要特征：读者可以使用所喜欢的任何实现，只要接口中的程序的行为符合用户的要求。

只有一种办法可以重用一个给定接口的实现：代理（delegation）。通过代理，一个类可以拥有它的接口的另一个实现的引用。这时读者只要提交所有的调用给被引用的实现中的相应的方法就可以了。实现代理的最明显、最常用、也是最费力的办法是在类中使用实现关键字、找到所有的有关方法并在每一方法中都添加一行代码。添加的这一行代码调用接口实现中的相应方法。

```
Private Function IFoo_DoBar _
    ByVal BarData As String) As String
        IFoo_DoBar = m_IFoo.DoBar(BarData)
End Function
```

这种技术有几个明显的缺点，其他一些缺点可能不太明显。首先写入这些代码本身就是一种挑战：VB 没有为读者产生任何形式的向导。当接口改变时，重新生成这些代码更是困难。尽管由于对象类型完全匹配，不存在 QueryInterface 调用，但在按值传送(ByVal)对象类型调用中，存在 AddRef/Release 方法。在字符串按值传送和变体按值传送中还有附加的字符串拷贝。在内部方法调用之前，需要对其他的按值传送参数重新进行检查。

如果不在字符串和变体中移动大量数据，代理中由参数传递造成的额外时间开销并不

算太大，但它也是确实存在着的。开始的编程和维护花销也是实际存在的。不幸的是，这种粗糙的代理模型是 VB 中存在的惟一方法。尽管以后将要讨论的聚合(aggregation)技术消除了代理维护的花费并使代理的花费减为最小，我还是要在此介绍在 VB 中使用一些技巧来避免使用代理的方法。

5.3.1 间接的接口实现

实现关键字存在的最大问题是读者不得不为每一个成员函数提供一个实现。当在第一次实现一个接口时，这不过是类似点击鼠标的工作，尽管有些让人讨厌，但并没有坏到让人放弃使用这种特征的地步。但是，如果对一个已实现的接口做了一些添加或者修改，并且在多个类中使用这一接口，就会遇上真正的麻烦：代码维护的困难足以让读者下次再也不会使用实现方法。

通过提高调用和实现代码的间接程度，读者可以完全避免代码的维护问题并大幅度的减少所要编写的代理代码。间接使用一个接口意味着将一个通常使用的接口的实现移入另一个独立的类中。下面的类实现一个简单的接口，在这个接口中仅有一个用来获取接口的实现的属性。

```
'clsExposed
Implements GetIStuff
Private m_Stuff As IStuffImpl
Private Sub Class_Initialize()
    Set m_Stuff = New IStuffImpl
    'See multiple behaviors section
    m_Stuff.Type = ForClsExposed
End Sub
Private Property Get GetIStuff_GetStuff() As IStuff
    Set GetIStuff_GetStuff = m_Stuff
End Property
```

```
'Calling code
Dim clsX As clsExposed
. . .
'Call helper
DoStuff clsX

'Helpers
```

```
Sub Dostuff(ByVal Indirect As GetIStuff)
Dim Stuff As IStuff
    Set Stuff= Indirect.GetStuff
    'IStuff  code here
End Sub
```

间接性的第二种表现是读者可以对 IStuff 接口作出修改而不影响 clsExposed 实现。例如，如果要添加一个方法到 IStuff 接口上以支持类 clsOverExposed 的一个新属性，仅仅需要改变 IStuffImpl 和实际调用这一新方法的代码。这一改变对 clsExposed 或其他间接使用 IStuff 的类不产生影响。没有这种间接性，就需要对每一个类做出改变。

5.3.2 与基类共享数据

当调用对象的一个程序时，程序的行为通常由三类相互作用的数据所决定：它们是程序中的代码、传递的参数和成员变量的当前值，所有这些数据共同作用于程序。第四类数据（我在下面的讨论中将忽略它）是工程的全局状态。（读者并不需要有一本有关高级编程的书来了解如何尽可能的减少实例与全局变量间的相互作用。）下面看一下在前面讨论过的 IStuffImpl 类。

IStuffImpl 实现 IStuff，因此对它的外部行为的定义是在 IStuff 接口的定义中给出的。clsExposed 必须使间接实现的 IStuffImpl 的行为如同直接实现的 IStuffImpl 一样。这就意味着 clsExposed 必须设置 IStuffImpl 类的所有成员变量。如果 clsExposed 能够完全控制 IStuffImpl 的成员变量状态，它也就能完全控制 IStuffImpls 运行时的行为。

只创建一个成员变量是共享一个类的所有成员变量的最容易的方法。这个成员变量可以是用户自定义的类型。Class 模块并不支持用户定义类型的公有变量，因此必须把类型定义置于标准模块中。然后 clsExposed 填充成员变量并通过一个友元函数将其赋给 IStuffImpl。

```
In a module
Public Type StuffImpleData
    Name As String
End Type
```

```
'In IStruffImpl
Private m_Data As StuffImplData
Friend Property Get Data() As StuffImplData
    Data = m_Data
End property
```

```
Friend property Let Data(RHS As StuffImplData)
    m_Data = RHS
End property
```

```
'In clsExposed
Private m_Stuff As IStuffImpl
Private Sub Class_Initialize()
Dim Data As StuffImplData
    Data.Name = "clsExposed"
    Set m_Stuff = New IStuffImpl
    m_Stuff.Data = Data
End Sub
```

尽管这段代码可以立即访问所有的数据，但同时拷贝数据来完成变量的初始化并在设置变量时作深度拷贝都成为必要的操作。深度拷贝会复制结构中的所有数据，而浅拷贝只复制指向原始数据的指针。如果用户自定义类型不包含可变长度的字符串或可变长度的数组，只是一些小的结构，这时作深度拷贝效果良好。但如果结构中包含一些指针类型，拷贝将会变成十分费时的工作。

如要避免作拷贝或使 clsExposed 更直接的访问 IStuffImpl 中的数据，就应使用第二章"写入数组变量"一节中介绍的数组描述符技术。这种技术可以使 clsExposed 直接访问 IStuffImpl 实例中的 m_Data 结构。通过将每一个类中的一个变量指向同一数据，我们可以消除用来同步结构的两个拷贝的通知和拷贝代码。

```
'IStuffImpl  class
Implements IStuff
Private m_Data As StuffImplData
Friend Property Get DataPtr() As Long
    DataPtr = VarPtr(m_Data)
End Property
Public Property Get IStuff_Name() As String
    IStuff_Name = m_Data.Name
End Property
```

```
'clsExposed class
Implements GetIStuff
Private m_Stuff As IStuffImpl
```

```
Private m_SAStuffData As SafeArrayId
Private m_StuffData() AS StuffImplData
Private sub Class_Initialize()
    Set m_Stuff = New IStuffIMpl
    ShareMemoryViaArray VarPtrArray(m_StuffData), _
       m_SAStuffData, m_Stuff.DataPtr, LenB(m_StuffData(0))
    m_StuffData(0).Name = "clsExposed"
End Sub
Private Sub Class_Terminate()
    'clean the array variable
    UnshareMemory VarPtrArray(m_StruffData)
End sub
Private Property Get GetIStuff_GetStuff() As IStuff
    Set GetIStuff_GetStuff = m_Stuff
End Property
```

```
'Reusable helper function to process IStuff
Function GetName (ByVal Indirect As IGetStuff) As String
    GetName = Indirect.GetStuff.Name
End function

'calling code snippet
Dim clsExposed As new clsExposed
Debug.print GetName (clsExposed)
```

使用这段代码时，应确保子类的 Class_Terminate 不共用数组的存储空间。这可以防止子类在基类的 Class_Terminate 获取数据之前就清空这一数组变量。另外要注意不必使基类访问到所有的变量。如果要求数据为私有类型，应将它们移到共享结构之外。

5.3.3 一个类的多重行为

在代码重用的继承模型中，读者经常要重载函数来提供类的不同实现。当子类重载基类接口中的方法时，调用代码会直接跳到子类的方法中去。在许多情况下，重载会改变类的行为，但重载的函数并不比基类中的相应函数需要更多的数据。

尽管 VB 不支持继承，读者可以在“同一数据/不同算法”重载例程中使用一个简单的 Select Case 语句和共享的数据结构中的 BehaviorType 字段来模拟函数重载。改变

BehaviorType 字段可以有效的将不同算法加入读者的类中。由于 Case 语句中包含常量表达式，Select Case 程序块运行起来十分迅速，所以这种技术的运行开支微乎其微。例如，假设读者有一个可画直线、圆和矩形的绘图类。在下面的继承模型中，可看到三个类都是继承自同一个抽象基类（抽象基类要求重载某一给定的方法）。

```
'In a standard module
Public Enum DrawType
   ddtLine=0
   dtRectangle=1
   dtCircle=2
End Enum
Public Type DrawImplData
   X1 As single
   Y1 As single
   X2 As single
   y2 As single
   DrawType as DrawTYpe
End type

'The implementation of the DrawImpl class. For simplicity,
'IDraw simply draws to a Form. I'll let you deduce the IDraw
'interface.

Implements IDraw
Private m_Data As DarwImplData
Friend Property Get DataPtr() As Long
   DataPtr = VarPtr(m_Data)
End Property
Private Sub IDraw_Draw(ByVal Target AS Form)
Dim Width AS single
Dim Heitht AS single
   With m_Data
      Select Case .DrawType
         Case dtLine
         Target.line(.x1, .y1)-(.x2, .y2)
      Case dtRectangle
         Target.Line (.x1, y1)- (.x2, .y2), ,B
      Case dtCircle
         Width = .x2 - .x1
         Height = .y2 - .y1
```

```
            If Width > Height Then
                Target .Circle _
                 ((.x2 + .x1) / 2, (.y2 + .y1) / 2), _
                  Height , , , , Height / Width
            Else
                Target.Circle -
                    ((.x2 + .x1)/2, (.y2 + .y1)/2), _
                    Width, , , , Height/Width
            End If
        End Select
    End With
End Sub
Private Property Get IDraw_x1() AS Single
    IDraw_x1 = m_Data.x1
End Property
private Property Let IDraw-x1(ByVal RHS AS Single)
    m_Data.x1 = RHS
End Property
Private Property Get IDraw_y1() AS single
    IDraw_y1 = m_Data.x1
End Property
Private Property Let IDraw_y1(ByVal RHS As Single)
    m_Data.y1= RHS
End property
Private Property Get IDraw_x2() As Single
    IDraw_x2= m_Data.x2
End Property
Private Property Let IDraw_x2(ByVal RHS As Single)
    m_Data.x2 = RHS
End Property
Private Property Get IDraw_y2() As Single
    IDraw_y2=m_Data.x2
End Property
Private Property Let IDraw_y2(ByVal RHS As Single)
    m_Data.y2 =RHS
End Property
```

在实际情况中，很少要对一个工程的基类做太大的改变。通过将所有的实现移入一个类中、在类上设置一个类型属性，就可以共享大量代码并能迅速跳转到正确的代码上。这样就可得到有一个核心实现的可预测代码。如果要对对象的行为做很大的改变或者对象的

一个实现所需的数据与以前的实现完全不同，就应使用实现(Implements)来编写另一个与前面有同样接口的基类。

5.4 聚　　合

聚合（Aggregation）这一术语广泛应用于 COM 编程中，它指的是对象创建时的一种标识技术。COM 对象的标识是由它的控制 IUnknown 指针创建的（见第三章）。当对象作为一个聚合类型创建时，它得到控制 IUnknown 的指针，这时，它将任何外部 IUnknown 操作都交给这个控制指针。外部对象可获得用来与内部对象通信的另一个 IUnknown 实现。COM 对象是通过创建另一个内部对象并将其作为外部对象的控制 IUnknown 的聚合，来提供它的接口的。

下面介绍的聚合的概念远不如 COM 编程中的定义严格。通常，聚合指的是将多个对象合并入一个对象中。尽管这一复合对象的各部分来自于不同的对象，它还是作为一个对象来使用。更深一层的概念是，聚合用很少的工作，就可使对象提供它所有的接口实现。也就是说，可以使用一个实现来完成数个类才能完成的工作。

VB 对象不能作为 COM 聚合来创建，并且在 VB 中不存在一个内在机制使一个 VB 对象能以聚合的形式创建另一个对象。这些支持的缺乏使 VB 程序员不能使用代码重用的一整套机制：为了将整个接口的控制权交给一个以前实现的对象，必须在函数的层次上实现一个接口，而不是使用 COM 聚合。

VB 不能在对象创建时聚合，并且在对象创建之前获得对对象的控制是非常困难的（如果可能的话），因此我并不打算告诉读者如何支持严格的聚合。但是，下面将介绍如何将一个或多个以前创建的对象合并然后聚合到现在的对象中去。通过使用现在对象的 IUnknown 方法并用 super-lightweight 代理层（称为盲 VTable 代理）来封装 COM 对象，可以创建一个包含聚合能提供的所有代码重用好处的复合对象。如果读者只对创建聚合有兴趣，而不关心具体是如何完成的，请直接跳到本章的“聚合现存的对象”一节，读者还可参考本书后面附录中的“聚合函数”一节。

5.4.1 盲 VTable 代理

用于封装对象的 VTable 代理技术是一种底层的技术（用汇编代码实现），它可使封装的对象的行为与未封装的对象完全相同。VTable 代理可修改 VTable 中的一些函数的属性而不改变其余函数的行为。在用于聚合的 VTable 代理中，我提供了三个 IUnknown 函数的实现，并且保留了其余的函数。

为了理解 VTable 代理是如何工作的，先应了解 VTable 是如何工作的。COM 二进制标准规定每一个 COM 对象的第一个元素都是指向一个的虚函数表（VTable）的指针。因此

如果 pObj 是指向 COM 对象的指针，*pObj（内存 pObj 的值）就是 VTable。VTable 是函数指针的有序数组，函数指针的排列顺序与描述 COM 对象的接口中定义的顺序相一致。因此，(*pObj)[0]是函数指针数组的第一个函数，并且它是 IUnknown.QueryInterface 函数的一个实现。所有对 COM 对象的外部调用都来自于 VTable，因此如果两个对象的 VTable 中的函数有同样的行为，它们的外部行为也会完全相同。

为了搞清楚盲代理是如何工作的，可看一个纯粹的盲代理的例子。在纯粹的盲代理中，所有函数都从属于函数的一个实际实现。我仅拿它当作说明问题的一个例子，毕竟创建一个盲代理是毫无意义的。盲代理对象由两个必需的元素：VTable 指针（m_pVTable）和指向实际对象的指针（m_punkInner）。m_pVTable 指针指向一个函数数组，在执行时，函数的偏移量移入 CPU 寄存器中，然后跳转到中心程序。中心程序用 m_punkInner 指针代替堆栈上的 this 指针，并在 m_punkInner VTable 中寻找真正的函数指针。为证实上面讲的这些，读者可以查看 CPU 寄存器寻找函数号为 40 的盲代理。在 Intel 汇编语言中的显示如下，其中 ESP 是当前的堆栈指针，ECX 传递函数偏移量。

堆栈如下所示：

```
ESP        ;return adress (jump back here upon
                           completion))
ESP+4      ;the this pointer (points to the blind delegator
                              structure)
ESP+8      ;the first parameter
ESP+12     ;the second parameter
etc
```

伪代码：

```
ECX = 40
Jump to main function
ESP+4 = m_punkInner from blind delegator structrue
Jump to the correct function in the real vtable, calculated
 With
(*m_punkInner) [ECX]
```

在这里应注意到参数在整个过程中没有改变。仅仅是堆栈中的 this 指针发生改变，因此这段代理代码可对任意函数进行操作。使用对 VTable 入口进行操作的代理代码，可将一个 VTable 用作任意对象的盲代理。尽管这段代码有一定的时间开销，但它明显要比在 VB 中写一个代理函数要快得多，那种函数要改变所有参数。通过盲代理来调用也不会对性能产生很大的影响。

为了将一个纯粹的盲代理转变为对于聚合有用的代理，应添加一些成员变量并重载 IUnknown 的函数。第一个新变量是 m_punkOuter，它是一个指向控制 IUnknown 的指针。IUnknown 建立了对象的识别符并且添加了一个 m_cRefs 成员变量以保持引用计数。被替换的 QueryInterface 函数向 m_punkOuter 提交 QueryInterface 调用，因此封装的对象与控制 IUnknown 有相同的 COM 识别符。代理结构可见图 5.1。

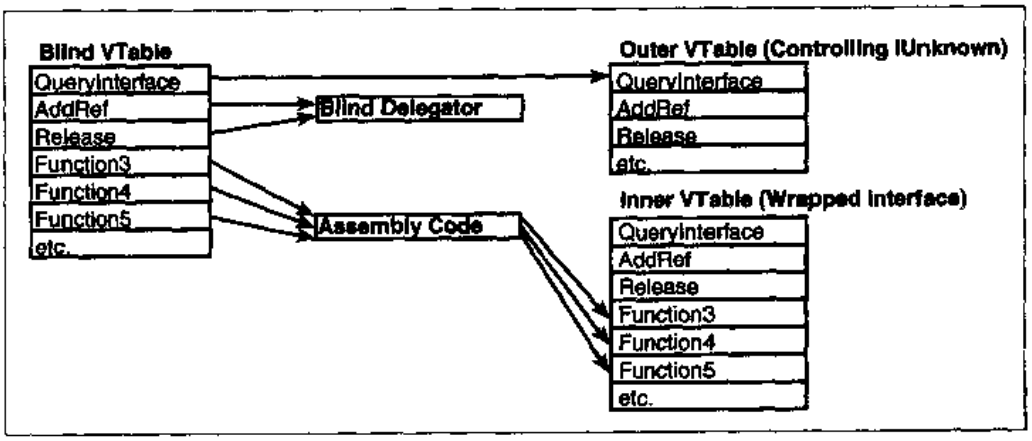

图 5.1 重定向盲代理中的 VTable 函数

5.4.2 IUnknown 钩入

Visual Basic 提供了 IUnknown 函数的实现。但是，聚合要求对 QueryInterface 函数进行控制，以使对象能够响应对那些编译后的并不知道是否支持的接口的请求。所有 QI 调用都返回到控制 IUnknown，因此所有的操作都要对控制 IUnknown 进行。 IUnknown 钩入（Hooking）的目的是在对象上的 QueryInterface 被调用时接收调用返回（并且使这些在 VB 中更易于实现）。

IUnknown 钩入与盲代理十分相似，但不同的是，IUnknown 钩入直接修改对象而不是封装该对象。钩入一个接口是十分简单的。指向对象的指针值就是指向 VTable 的指针值。要钩入一个 VTable 中的函数，只要记录下当前的 VTable 指针并且用一个指向替换后的 VTable 的指针覆盖它就可以了。VTable 调用现在指向了另一个函数，这一函数将对象的 VTable 恢复为它的初始值并提交该调用。在返回之前，对象的钩入状态恢复为初始值，具体可见图 5.2。

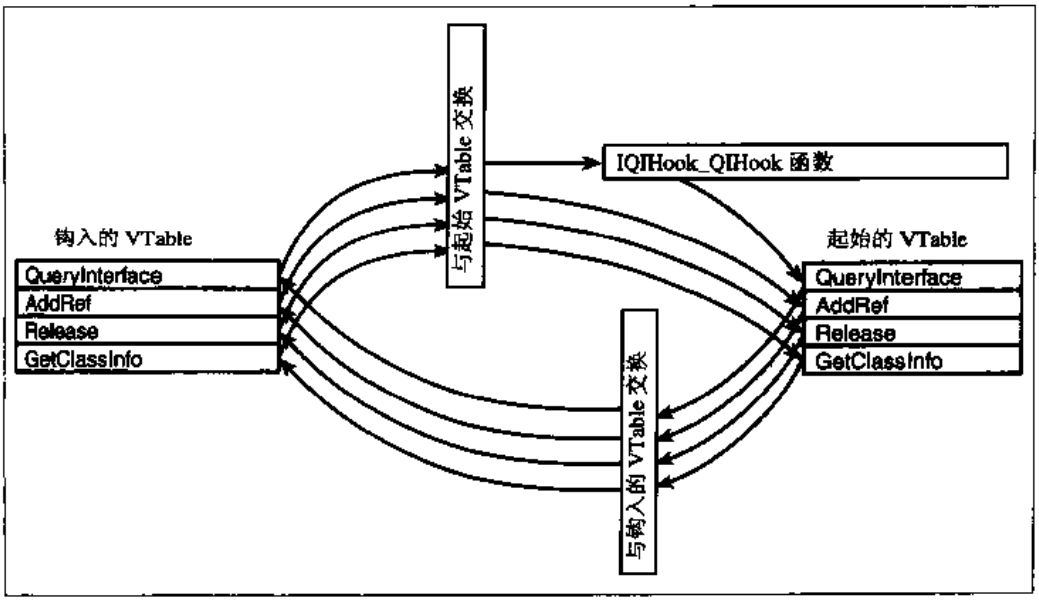

图 5.2 IUnknown 钩入为 VB 对象控制 IUnknown 提供了一种替代 VTable 的方案，并允许对 QueryInterface 调用进行定制

本书中的 VBoost 对象提供了钩入能力。它的钩入函数的最快的实现是用 C++语言编写的 VBoost6.Dll 实现，同时在文件 VBoost.Bas 里 VBoost.Bas 对象的 compiled in 版本中，也可找到这一函数（使用本书中介绍一些技巧）。在内部的 QueryInterface 调用完成之前和（或者）之后，这一实现通过 IQIHook 接口回调返回。它还提供一个阻塞请求的 IID 或将被请求的 IID 映射为另一个不同的值的机会。读者也可以使用 IQIARHook 接口来观察 AddRef 和 Release 调用。

读者经常会发现一个 COM 对象的实现支持另一个与控制 IUnknown 有相同的 ObjPtr 的接口。任何替换 QI 调用的 VTable 必须能够支持别的函数。所有 VB 对象也都支持控制 IUnknown 上的 IProvideClassInfo 函数，并可向钩入代码中添加一些别的函数。VBoost 中的 IUnknown 钩入提供了一个修改 QueryInterface、AddRef 和 Release 的 VTable。它还对 ProvideClassInfo 函数做了一个简单的遍查（passthrough）操作。

只要对除了 IID_IUnknown 和 IID_IProvideClassInfo 之外的接口作 QueryInterface 请求，不返回与控制 IUnknown 相同的指针，读者就可以使用 VBoost 来钩入别的对象。即使内部 QI 调用成功了，VBoost 也会释放数据并返回 E_NOINTERFACE 错误信息。如果读者试图钩入一个用 C++编写的对象，必须对应每一类型的接口来分别修改钩入程序的源代码。

下面是一个钩入 QI 调用的简单例子，其中有一个类，读者可观察它的 QI 调用是如何完成的。在这个例子中，读者可追踪到最后一个 IID。如果读者工作在 VB 集成开发环境中，必须一次运行所有的代码，而不是一步一步的执行，因为在做分步执行时调试器实际上会

作数个 QI 调用。

```
'clsLastIID
Implements VBoostTypes.IQIHook
Private m_Hook As VBoostTypes.UnKownHook
Private m_LastIID As VBoostTypes.VBGUID
Private Sub Class_Initialize()
    'The first parameter is the controlling
    'Iunknown. The second parameter is the IQIHook callback.
    'The third is the initial flags that indicate
    'a mapiid and/or before notification and/or
    'after call notification.The fourth
    'parameter is the variable that owns the hook.
    VBoost. HookQI Me, Me, uhBeforeQI, m_Hook
End Sub

Private Sub IQIHook_MapIID(IID As VBoostTypes.VBGUID)
End sub
Private Sub IQIHook_QIHook(IID As VBoostTypes.VBGUID), _
    ByVal uhFlags As VBoostTypes.UnkHookFlags, _
    pResult As stdole.IUnknown, -
    ByVal HookedUnkown As stdole,IUnknown)
       m_LastIID = IID
End Sub
Friend Property Get LastIID() As VBoostTypes.VBGUID
    LastIID = m_LastIID
End property
```

```
'Calling code that determines the IID of Class1
Dim clsLastIId As New clsLastIID
Dim cls1 As Class1
Dim IID_Class1 As VBoostTypes.VBGUID
On Error Resume Next
'This will fail, but try it to see the IID
Set cls1 = clsLastIID
On Error GoTo 0
IID_Class1 = clsLastIID.LastIID
```

使用钩入对象很长一段时间后，读者会注意到局部窗口和观察窗口是如同读者所期望那样工作的。如果工程中包含一个对在工程中实现的类的引用，读者可以如同在类内部一

样扩充对象变量、观察成员变量。一旦钩入一个对象，调试器就不把该对象视为 VB 对象（从技术上讲它不是，因为对象是由它的 VTable 标识的），因此不能像对通常的 VB 对象那样对待该对象。但是，在对象里面，局部窗口中的 Me 入口仍然是有效的。

要提醒读者的是：如果使用由 VB 实现的 UnKnownHook，当读者试图扩充对象（除了 Me 对象）的引用时，VB 会崩溃；借助于 C++实现，读者会看到<no variable>的信息。与调试器进行交互是很复杂的一件事情，我也讲不清为何用 VB 编写的实现会崩溃，而 C++的实现却不会（这个原因至少不是轻易就可搞清的）。读者可以通过设置条件编译标志 VBOOST_INTERNAL=0 以在调试过程中使用 DLL 的 C++版本。

在附录 A 中，对 VBoost 聚合对象作了广泛的讨论。现在先看一个使用聚合对象的例子。

使用下面的代码可使 Class2 支持对 Class1 作的 QueryInterface。这段代码使用前面提过的 clsLastIID 类来确定 Class1 的 IID。

```
'Class2 implementation
Implements VBoostTypes.IQIHook
Private m_Hook As VBoostTypes.UnknowHook
Private IID_Class1 As VBoostTypes.VBGUID
Private m_Class1 As Class1

Private Sub Class_Initialize()
Dim clsLastIID As New clsLastIID
    On Error Resume Next
    Set m_Class1 = clsLastIID
    On Error GoTo 0
    IID_Class1 = clsLastIID.LastIID
    Set m_Clss1 = New Class1
    VBoost.HookQI Me, Me, uhBeforeQI, m_Hook
End sub

Private Sub IQIHook_MapIID(IID As VBoostTypes. VBGUID)
End Sub
Private Sub IQIHook_QIHook(IID As VBoostTypes.VBGUID, _
    ByVal uhFlags As VBoostTypes.UnkHookFlags, _
    pResult As stdole.IUnknown, _
    ByVal HookedUnkown As stdole.IUnkown)
        If IsEqualIID(IID, IID_Class1) Then
```

```
        'Create a delegator around m_Class1 using
        'HookedUnknown as the controlling IUnknown.
        Set pResult = VBoost.CreateDelegator _
            (HookedUnknown, m_Class1)
    End If
End Sub
```

```
'Calling code
Dim cls1 As Class1
Dim cls2 As New Class2
Set cls1 = cls2
Debug.Print cls1 Is cls2
'Output
True
```

5.5 聚合现存的对象

读者要实现自己的 IQIHook 和盲代理并非易事，因此 VBoost 为基于传递来的数据创建聚合对象提供了两个方法。第一个方法为 AggregateUnknown 方法，使用 IUnknown 钩入在现存的控制 IUnknown 之上构建聚合。第二个方法为 CreateAggregate，它创建一个聚合多个对象的新的控制 IUnknown。读者必须使用 AggregateUnknown 来添加接口支持到 MultiUse 对象上，因为这些对象都是在外部创建的，这样聚合此对象的最早时刻是在 Class_Intialize 时。如果对象为 PublicNotCreatable 或在内部创建的，读者可完全控制对象的创建并能确定返回给调用者的控制 IUnknown，所以可以使用另一个叫做 CreateAggregate 的方法。使用 AggregateUnknown 的好处是对象支持的固有接口都不是用盲代理来封装的，因此，它的性能不受影响。使用 CreateAggregate 的不利之处是它需要 IUnknown 钩入。

AggregateUnknown 和 CreateAggregate 方法都可获得 AggregateData 结构的数组。AggregateData 定义在 VBoostTypes6.Tlb 中，它包含四个成员。其中 pObject 是对要聚合的对象的控制 IUnknown 的引用，下面要讨论的 Flags 是一个枚举类型，FirstIID 和 LastIID 是一个 VBGUID 结构的数组的索引，这一数组同 AggregateData 数组一起传送给 VBoost 聚合方法。如果 FirstIID 未被填充，对象将视为盲聚合处理；无论传来的 IID 为何值，QI 调用都被提交给该对象。对盲代理编程是十分容易的，因为读者并不需知道一套接口识别符。在所有的聚合对象有机会返回请求的接口之前，VBoost 聚合对象处理所有指定的 IIDs。对盲 QI 调用的处理是按照 AggregateData 数组中指定的顺序进行的。

使用 AggregateUnknown 方法，可以很容易的编写出 Class2。读者通过不指定传递来的 AggregateData 结构的 IID，可以通知 VB 将所有控制 IUnknown 不能处理的接口请求告知 m_Class1。用另一句话来说，我们在盲聚合 m_Class1。最后得到的接口间的相互作用见图 5.3。

```
'Class2 Implementation. Aggregates Class1
Private m_Hook As UnknownHook
Private m_Class1 As Class1
Private Sub Class_Initialize()
Dim AggData(0) As AggregateData
Dim IIDs() As VBGUID
    Set m_Class1 = New Class1
    Set AggData(0).pObject = m_Class1
    AggregateUnkown Me, pAggData, IIDs, m_hook
End Sub
```

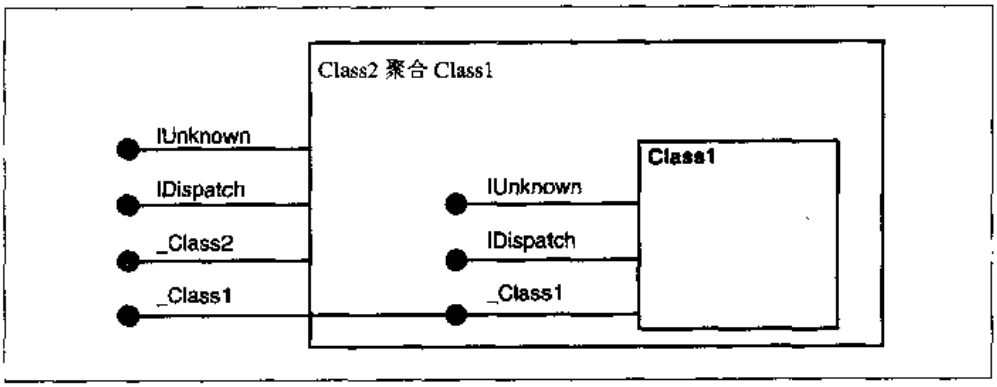

图 5.3 Class2 使用 VBoost.AggregateUnknown 来聚合 Class1

Class2 所支持的本地接口没有被封装，而_Class1 所支持的_Class1 接口使用一个盲代理进行了封装，使之像 Class1 的本地部分一样作用。外部用户无法分辨究竟是聚合接口还是本地实现的接口。

从这一段小程序中可以看出，聚合现存的对象给代码重用带来的好处是很明显的。读者可以用很少的代码，利用一个完成的类来实现别的类。聚合代码也很易于维护。如果一个聚合类或接口改变时，只要它的 IID 不改变，读者就不需对代码作任何改变。

在与基类共享内存空间并将基类聚合入现存的 IUnknown 中时，代码重用就是很自然的事情了。让我们回到以前创建的 IDraw 接口，基于这个接口创建一个 DrawCircle 对象。这一对象与 DrawImpl 类共用数据，它还添加了一个根据环绕矩形计算区域面积的固有方法

（方法名为 Area）。

```
'Class DrawCircle
Private m_SADrawData As SafeArrayId
Private m_DrawData() As DrawImplData
Private m_Hook As UnknownHook
Private Sub Class_Initialize()
Dim DrawImpl AS DrawImpl
Dim AggData(0) As AggregateData
Dim IIDs() As VBGUID
   Set DrawImpl = New DrawImpl
   ShareMemoryViaArray VarPtrArray(m_DrawData), _
      m_SADrawData, DrawImpl.DataPtr,LenB(m_DrawData(0))
   m_DrawData(0).DrawType = dtCircle
   Set AggData(0).pObject = DrawImpl
   'adIgnoreIIDs is the default Flags setting.
   'Since no IIDs are referenced from the AggData array,
   'there is no need to dimension the IIDs array.
   VBoost.AggregateUnknown Me, AggData, IIDs,  m_Hook
End Sub
Public Property Get Area() As Single
'Area = Width/2 * Height/2 * pi
'pi = Atn(1) *4
'The area of ellipse/circle is Width * Height *Atn(1)
   With m_DrawData(0)
      Area = (.x2 - .x1)* (.y2 - .y1) *Atn(1)
   End With
End Property
Private Sub Class_Terminate()
   UnshareMemory VarPtrArray(m_DrawData)
End Sub
```

这是一个简单的对象，但是它其中所体现的原则很容易应用到需要数个聚合接口的更大的对象。以前讲过的间接的接口实现现在看来也不是必要的：因为聚合对象改变时，聚合类不必做出改变，读者得到的代码是不需要维护的。调用代码也变得更加简单。这里显示的 DrawCircle 并不实现 IDraw，甚至没有那种类型的成员变量，但它可充分支持这一接

口。

读者应尽量在另一个聚合对象中封装一个聚合的对象。毕竟，聚合的对象与原先对象有相同的外部特征。但是，如果在同一类中有两个或多个活动的 UnKnownHook 变量，应确保以创建时的逆序来释放钩入对象。例如，如果要实现 IQIHook 接口来调试一个聚合对象，需要一个 UnKnownHook 变量（m_Hook）给 AggregateUnknown 调用，另一个（m_Hook Dbj）给 QIHook。如果在类初始化(Class_Initialize)时调用 AggregateUnknown 和 QIHook，结束事件将以逆序释放钩入对象。

```
Private Sub Class_Terminate()
    Set m_HookDbg = Nothing 'Last hook established
    Set m_Hook = Nothing    'First hook established
End Sub
```

惟一没有实现的一个继承的特征是将一个函数重定向到子类的实现上。这种特征可以借助于调用函数指针的能力（将在第十一章中介绍）来实现，调用函数指针可通过将函数指针放于公有数据结构中，然后从实际的入口点来调用这一指针来完成。例如，为了扩充 IDraw 的功能以使其支持更复杂的对象并可画出更多形状的图形，应让 draw 方法服从于一个由基类指定的函数指针。这些将在第十二章中“协同重定向”一节中详细的加以介绍。

第六章

循 环 引 用

在 VB 中，引用计数控制了每一个 COM 对象的生存期。每次，当我们使用 Set 语句把一个对象表达式赋值给一个对象变量时，变量便拥有了一个对该对象的引用。但这里需要注意的是，持有一个对象的引用并不表明变量持有对象本身。如果我们在同一时刻将同一个对象实例赋给两个不同的变量，则每个变量都各自拥有一个该对象的引用计数。另外，在这两个变量之间，并不存在任何区别—每个对象引用都完全相同。

读者也许会问，如果变量未持有 COM 对象，该怎么办呢？答案当然在于对象本身。我们知道，COM 对象具有自销毁的性质。也就是说，当最后一个引用被销毁时，COM 对象能够自行销毁。通过语句：Set <variable> = Nothing 来释放一个对 COM 对象的引用只是简单的减少了该对象的引用计数。尽管这看起来似乎纯粹是语义上的不同，但我们绝对不能把释放一个对象的引用等同于销毁一个 COM 对象。持有一个引用在这里仅仅是一种说法："我正在使用你。"而与此相对应，释放一个引用则表明："我对你已经使用完毕！"。一旦最后一个引用被释放，对象便自行销毁。

在绝大多数时间里，引用机制能够很好的对一个对象的生存期进行控制。引用计数代价较小，也就是说，如果对于单个对象进行多次引用，并不会增加额外的开销。并且，这种机制还提供了完美的封装性和一致性，因为在代码中，我们无需知道有关如何终止一个对象的具体细节便能够使用该对象。VB 只是简单的释放一个对对象变量的引用，而销毁一个对象的工作则交给对象本身来做。

在这里，应当引起读者警惕的是，当我们使用引用计数来控制对象的生存期时，存在的一个固有问题便是—循环引用。下面举例说明：我们假定，A 对象持有一个对 B 对象的引用，并且，只有在销毁 A 对象时，A 对 B 的引用才会随之释放。同样， B 对象也持有一个对 A 对象的引用，并且 B 对 A 的引用只有在销毁了 B 对象时才会随之释放，这样的话，便会导致整个系统总是无法被销毁。因为这时，对象会处于一个无休止的等待循环之中—即每个对象都在等待另外一个对象首先被拆分，如图 6.1 所示。

导致产生循环引用这一问题的根源在于：在保持对象处于激活状态方面，我们对所有

的引用都是基于同样的考虑。为了使一个循环引用系统能够被拆分，我们希望得到的是这样一种引用系统：在整个系统中，A 对象能够以正常的方式引用 B 对象，而 B 对象对 A 对象则采取一种特殊方式的引用，这种引用并不会保持对象处于激活状态。在这里，我们称能保持对象处于激活状态的引用为强引用，而称不能保持对象处于激活状态的引用为弱引用。对于一个对象变量和 Set 语句来说，强引用是缺省的一种引用方式。

我们知道，一个 VB 对象变量由一个对象的引用和一个指向该对象的指针所组成。当我们在程序中写入语句：Set = Nothing 时，该引用便被释放，并且对象指针被设置成零值。为了得到弱引用，我们需要一个无引用计数并指向对象的指针，然后使用 ObjPtr 函数来返回一个与对象指针具有相同值的长整型值。

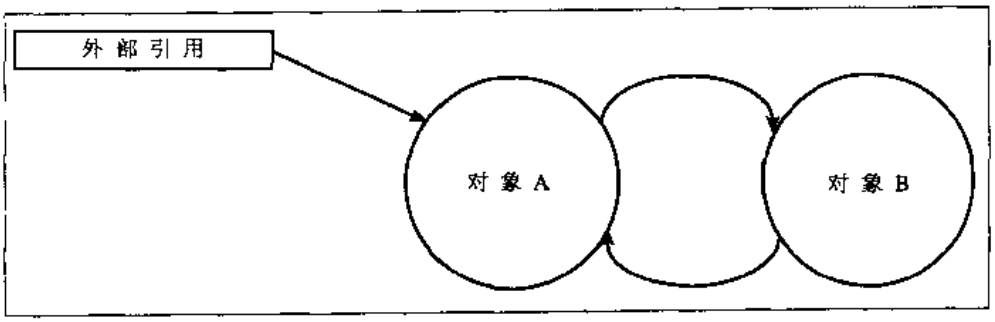

图 6.1 一个循环引用系统

```
Dim StrongRef As SomeObject
Dim WeakRef As Long
Set StrongRef = MyObj
WeakRef = ObjPtr(MyObj)
```

关于弱引用，应该注意的第一个问题便是：VB 中并不提供对使用长整型变量来调用一个方法的保护。有的时候把弱引用称作为“冷于”，与此相对应，我们把由一个弱引用转化为强引用的行为称作为“水合化”（rehydration）。正如这个形象化的称谓所暗示的一样，我们应该首先为弱引用加入一些“水”，将其转变为一个强引用后，才能使用弱引用。使用强引用的一个好处便是：我们可以使用所引用对象的各种属性和方法。读者可以直接在 VB 代码中加入如下函数来实现“水合化”。其中，CopyMemory API 函数在 VBoostTypes 类型库中进行了定义。

```
Funtion ObjFromPtr(ByVal Ptr As Long) As Object
Dim tmp As Object
    'Copy the pointer into an object variable.
    'VB now thinks that tmp contains a proper strong
```

```
    'reference; it will release the reference
    'when tmp goes out of scope or is otherwise
    'Set to Nothing.
    CopyMemory tmp, Ptr 4
    'Use a normal Set statement to place a
    'strong reference in the function name
    Set ObjFromPtr = tmp
    'Clear the temporary variable. The Set
    'statement can't be used here because this
    'would release a reference count that was
    'never actually added.
    CopyMemory tmp, 0&, 4
End Function
```

由于这里我们只是想要分配一个指针值，并为其添加引用，因此，上面的 ObjFromPtr 函数所做的工作也就很有限。而且需要注意的是，ObjFromPtr 函数并不是通用的，需要根据弱引用的不同类型而使用该函数的不同版本。而通过使用 AssignAddRef 函数，VBoost 对象可使之有效的多。该函数中并不需要调用 QueryInterface 函数，而是使用 As Any 类型来将一个长整型值转变成对一个任何对象类型的引用。为了能用 AssignAddRef 来生成一个指定类型的函数，只需要简单的将 AssignAddRef 包含在一个指定类型的函数中就行了。读者可以将下面所列出的函数作为任何指定类型的转换函数的一个原型。

```
Function MyTypeFromPtr(ByVal Ptr As Long) As MyType
    VBoost.AssignAddRef MyTypeFromPtr, Ptr
End Function
```

现在，我们便可以为一个自定义类型的变量分配一有效的 MyTypeFromPtr 返回值了，或者通过 With 语句来使用该函数。

```
With MyTypeFromPtr(WeakMyTypeRef)
    .MyMethod
End With
```

对于弱引用来说，其安全性也是一个应该注意的问题。可以想像，如果我们只是得到一个对象的 ObjPtr，而并不持有对该对象的引用的话，便不能保持该对象处于激活状态。如果某对象指向的是一块已被释放的内存，VB 中采用了引用计数机制来保证对其进行禁

用，因此，我们在使用 AssignAddRef 从弱引用中取得一个强引用之前，必须百分之百的确信该弱引用确确实实指向的是一个活对象。

总而言之，对于一个弱引用，我们应当保证该弱引用总是有效的。最简单的一种方法是让目标对象来设置和清除弱引用。由于弱引用总是在循环引用的情况下使用，所以弱引用的对象（class A）已经持有一个对持有相应弱引用的对象（class B）的强引用。值得注意的是，在这里应该使用友元函数来建立弱引用。与公有函数相比，使用友元函数的一个优越性在于：它使得我们能够对弱引用进行完全的内部控制。

```
'Class A
Private m_B As B
Private Sub Class_Initialize()
    Set m_B = New B
    m_B.SetAPtr ObjPtr(Me)
End Sub
Private Sub Class_Terminate()
    m_B.SetAPtr 0
End Sub
```

```
'Class B
Private m_Aptr As Long
Friend Sub SetAPtr(ByVal pA As Long)
    m_Aptr = pA
End Sub
Public Property Get A() As A
    'AssignAddRef handles a 0 in m_Aptr set the same way
    'a Set statement handles the Nothing value: it checks
    'the pointer value before calling AddRef.
    VBoost.AssignAddRef A, m_Aptr
End Property
```

6.1 中间对象解决方案

如前所述，弱引用在本质上是不安全的，这样，我们便自然而然会考虑：是否能够有一种替代方案来解决循环引用的问题。下面将会就“如何来避免无休止的循环引用”这一

问题进行讨论，并得出一些解决的办法。不过，同时也希望本节能够向读者表明一点：为了避免使用弱引用机制而导致增加麻烦和开支是不值得的。首先，让我们来检查一下一个仅仅使用引用计数机制的系统是否能够正确的被拆分。

如果 A 对象引用 B 对象并且同时 B 对象需要依赖于 A 对象的功能的话，那么这时我们可以使用一个中间帮助对象 A′ 来完成中间的衔接工作。在这样一种方案中，各对象间的关系为：A 引用 B，并且 A 和 B 同时都引用 A′，而在 A′ 与 B 中都没有对 A 的引用(既没有强引用，也没有弱引用)。这样，通过把 B 对象需要依赖的代码从 A 对象中移到了 A′ 对象，从而无需引用 A 对象便提供了对 B 对象的支持。如果读者深入的做了这一练习，便会发现原来由 A 对象来完成的功能大部分被移到了 A′ 中，这样相比于前一种方法，A 对象就变小了许多。该系统如图 6.2 所示。

大家可以看到，在这个系统中没有循环引用的问题，但同时在很大程度上也增加了局限性。例如，尽管 B 通过调用 A′ 中的相应方法能够有效的调用 A，但是对于 B 来说，它并不能返回一个对 A 本身的引用。这样，如果 A 和 B 都是一个对象模型中的公共模型，并且 A 是父对象的话，那么在 B 对象中并不支持公共的父属性。

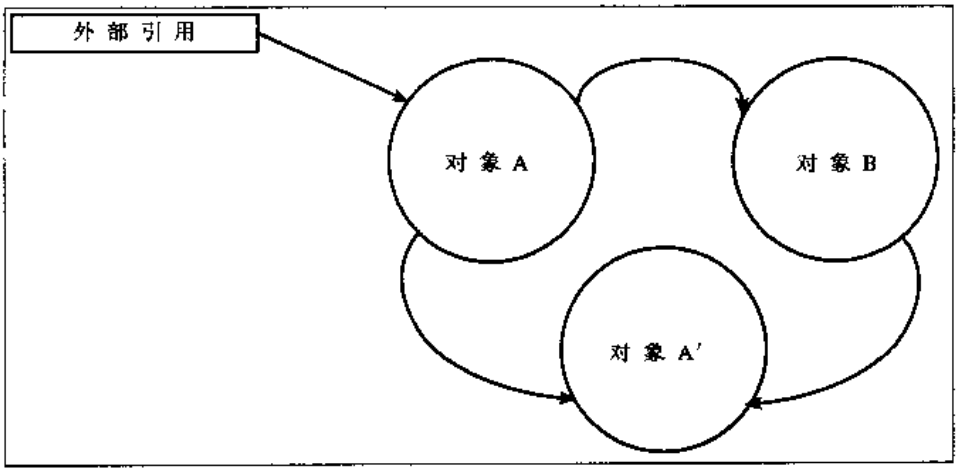

图 6.2　采用中间对象来消除对象 A 与对象 B 之间的循环引用

该系统中存在的另一个局限性是：由于只有 A 持有对 B 的引用，A′ 并不能对 B 进行访问。这里只是临时性的避开了弱引用，所以 A′ 不能引用 A 和 B。实际上，这一局限性比上一个局限性要严重的多。这是因为，A′ 作为 A 的一个实际上的实现，不能对其子对象进行访问。与第一种局限性不同，我们可以通过移除系统拆分单方面的依赖于引用计数这一要求来解决这个问题。引用计数机制之外的附加步骤被放在了 A 对象的 Class_Terminate 事件过程中，在这里，它必须调用 A′ 对象中的 Terminate 函数。该函数释放了所有对 B 的引用，并且该系统能够被拆分。该系统的示意如图 6.3 所示。

当然，这个系统在事实上仍然是循环的，但是中间对象的使用保证了系统中的非循环对象能够产生一个 Class_Terminate 事件，从而允许我们可以从外部终止存在于 A′ 和 B 之间的循环引用。另外需要注意的是，在实际编写拆分代码的时候，应该小心谨慎，否则系

统就会如同一个筛子一样产生对对象及其相应内存的泄漏。

```
'Class A
Private m_Helper As AHelper
Private Sub Class_Initialize()
    Set m_Helper = New AHelper
End Sub
Private Sub Class_Terminate()
    m_Helper.Terminate
End Sub
Public Property Get Child() As B
  Set child = m_Helper.Child
End Property
```

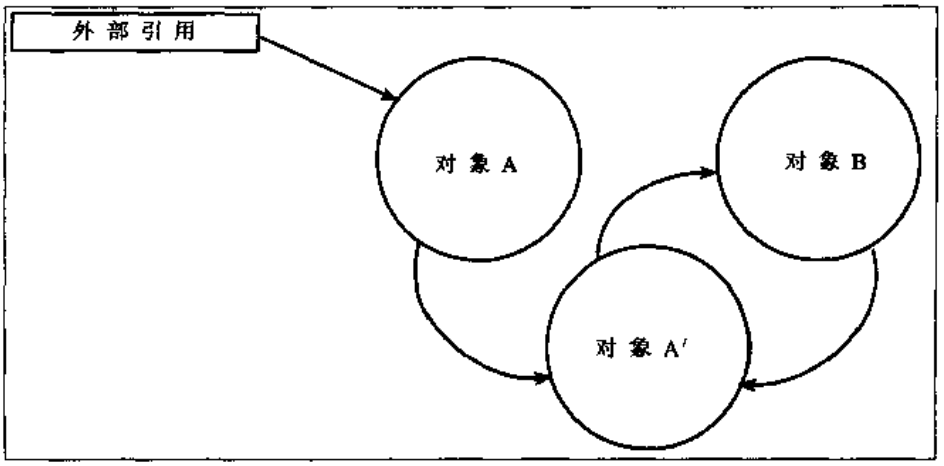

图 6.3 一种由 A 对象中的 Class_Terminate 方法所调用的外部终止方法中间对象通知释放对 B 对象的引用

```
'Class AHelper
Private m_Child As B
Friend Property Get Child() As B
    If m_Child Is Nothing Then
        InitChild
    End If
    Set Child = m_Child
```

```
End Property
Friend Sub Terminate()
    Set m_Child = Nothing
End Sub
Private Sub InitChild()
    Set m_Child = New B
    m_Child.SetParent Me
End Sub
```

```
'Class B
Private m_Parent As AHelper
Friend Sub SetParent(Parent As AHelper)
    Set m_Parent = Parent
End Sub
```

尽管这个系统显得有些杂乱，但是我们可以对其进行很好的管理。下面，我们将在该系统中为子类加入父属性，这样，系统就变得更为复杂。由于在系统中，弱引用这一原本可以利用的工具被忽略，所以我们不能从中返回原始的 A 对象。不过，我们仍然可以创建一个新的 A 对象并让其指向同一个 AHelper 对象，这样我们就能在系统中返回一个功能等同的对象。由于我们所希望的情况是：只有当 A 对象是最后一次引用 AHelper 对象时，它才调用 Terminate 这一事件过程，而通常情况下，这一信息是不能通过访问直接取得的。这样，就使得情况变得更为复杂。它导致我们需要让 AHelper 函数保持一个明确的被 A 对象使用过的次数，并且对于 A 对象，应该在每次开始使用 AHelper 对象以及使用完毕时都能够通知 AHelper 对象。最后，当我们重新审视这一新系统时，会发现所有这些代码的目的都只不过是为了避免使用弱引用。

```
'Class A
Private m_Helper As AHelper
Private Sub InitHelper()
    Set m_Helper = New AHelper
    AHelper.Advise
End Sub
Friend Sub SetHelper(Helper As AHelper)
    If Not m_Helper Is Nothing Then m_Helper.UnAdvise
    Set m_Helper = Helper
    Helper.Advise
```

```
End Sub
Private Sub Class_Terminate()
    If Not m_Helper Is Nothing Then m_Helper.UnAdvise
End Sub
Public Property Get Child() As B
    If m_Helper Is Nothing Then InitHelper
    Set Child = m_Helper.Child
End Property
```

```
'Class AHelper
Private m_Child As B
Private m_cAdvised As Long
Friend Property Get Child() As B
If m_Child Is Nothing Then
        InitChild
    End If
    Set Child = m_Child
End Property
Friend Sub Advise()
    m_cAdvised = m_cAdvised + 1
End Sub
Friend Sub UnAdvise()
    m_cAdvised = m_cAdvised - 1
    If m_cAdvised = 0 Then
        Set m_Child = Nothing
    End If
End Sub
Private Sub InitChild()
    Set m_Child = New B
    m_Child.SetParent  Me
End Sub
```

```
'Class B
Private m_Parent As AHelper
Friend Sub SetParent(Parent As AHelper)
    Set m_Parent = Parent
```

```
End Sub
Public Property Get Parent() As A
    Set Parent = New A
    Parent.SetHelper m_Parent
End Property
```

在同一工程中使用一个外部的拆分函数来保证程序能够正确的终止是合情合理的，但是，我们应该尽量避免使用交叉组件。如果我们能够做到使对象模型的拆分规则对客户来说尽可能的简单(这通常意味着可以采取随机的顺序释放所有引用)，那么对象的用户便总能知道如何来使用对象模型。微软在过去的几年内已经对这些简单的指导方针修改过好几次。

关于这种拙劣的对象模型的第一个设计范例便是 Excel.Application.Quit 方法。该方法供用户在不再需要使用 Excel 的时候调用。这里存在的一个问题便是：Quit 方法在终止某个人对 Excel 的应用的同时也终止了其他人对 Excel 的应用。最终，Quit 方法被修改，修改的结果是使之并不关闭整个程序，从而避免了这一极不友好的行为。一些数据对象，例如 DAO 也是这种对象模型的一个例子。在 DAO 中，必须按照正确的顺序来调用其 Close 方法，而且对象也必须按照一定的顺序进行释放（例如，Recordset 对象应在 Database 对象之前释放）。这个简单拙劣的对象模型很容易会导致一种误解：除非在函数的末尾明确的设置所有的局部变量为 Nothing 值，否则 VB 将产生内存泄漏。对于一个设计完美的对象模型来说，这个观点当然是完全错误的。实际上，VB 会在过程被终止的时候清除所有变量，并且速度要比在代码中进行手动清除快得多。而且即便我们已经明确的释放了所有引用，VB 也会对各变量进行检查。因此，我们所做的手动清除工作实际上是多余的。

6.2 弱引用和集合

当我们在一个 VBA 集合中存储对象时，ObjPtr 是非常有用的。ObjPtr 不仅提供了一种弱引用的方法，而且提供了对象的一个字符串标识。即使没有使用弱引用，我们也经常能发现自己正在使用 ObjPtr 产生整个对象的惟一标识。由于 ObjPtr 实际上是一个指向某一对象并在整个对象的生存期内保持不变的指针，因此当由 ObjPtr 返回的数转换成一个字符串时，便产生了一个惟一的关键字。对于一个集合对象来说，它并不接受一个数字型的关键字，但是它能够接受字符串，哪怕在该字符串中包含了数字。下面，让我们来看一段能有效的跟踪一个集合中类的所有实例的代码。对于这样一个跟踪系统来说，它需要用到弱引用，这是因为跟踪程序应该有效的跟踪存在的对象，而并不保持它们为活对象。读者看了程序代码之后也许会大吃一惊，因为它竟是如此的简单。

```
'In a bas module
Public g_CollMyClass As New VBA.Collection
```

```
Public Sub DumpMyClass()
Dim Inst As MyClass
Dim Iter As Variant
    For Each Iter In g_CollMyClass
        'MyClassFromPtr as above
        Set Inst = MyClassFromPtr(Iter)
        'Debug.Print Inst.Data
    Next
End Sub
```

```
'MyClass class module
Private Sub Class_Initialize()
Dim pMe As Long
    pMe = ObjPtr(Me)
    g_CollMyClass.Add pMe, CStr(pMe)
End Sub
Private Sub Class_Terminate()
    g_CollMyClass.Remove CStr(ObjPtr(Me))
End Sub
```

6.3 转移对象所有权

有时候，我们可以故意不把对象的引用存储在某个对象变量之中，以达到大大简化运算法则和简化代码量的目的。例如，可以将列表框（ListBox）中的某项与隐藏在其下面的一个对象相联系。这一问题不需要用到 ObjPtr 便可以解决，即使列表框（ListBox）中的每个字符串都对应一个存储对象的集合中的关键字。不过，在选择一个列表项的时候，要求有一个相应的集合查询来对该项进行访问，并且还要考虑到有关列表框中是否存在着重复项的问题。

为了进一步对该方法进行简化，在列表框中有一个 ItemData 属性，可以用它来使每一项都对应一个长整型值。这样，如果读者想要解决有关关键字惟一的问题，可以让对象集合持有 CStr(ObjPtr(object))类型的关键字，并将 ObjPtr 存储在 ItemData 之中。但是，我们仍然需要执行一个集合查询来取得该对象。这里，不需要将对象存储在别的某个位置，简单的将 ObjPtr 存储在 ItemData 中，并使得 ItemData 拥有对象的持有权就可以了。

```
Private Sub AddlistObject( _
```

```
  Name As String, ByVal Obj As Object)
Dim pObj As Long
    With lstObjects
        .AddItem Name
        'Transfer ownership of the reference to a
        'long value
         VBoost.AssingnSwap pObj, Obj
        .ItemData(.NewIndex) = pObj
End With
End Sub
Private Sub RemoveListObject (ByVal Index As Long)
Dim tmp As Object
    With lstObjects
        'Transfer ownership to a local variable
        'that will be released when this procedure
        'goes out of scope .
        VBoost.Assign tmp, .ItemData(Index)
       .RemoveItem Index
    End With
End Sub
```

我们还可以使用另外一种简单的对象所有权机制，这一机制除使用列表框之外，还使用到了非存储对象。在这种机制中，我们可以在 lstObjects_Click 事件过程中利用 AssignAddRef 函数来得到隐藏于列表框之下对象的一个强引用。当然，这种方法有个缺陷是：必须在列表框被清除之前编写明确的清除代码。很显然，在这里，列表框是一系列对象的持有者，我们应该在拆分时明确的对它进行清除。

6.4 层次化对象模型

几乎所有对象模型都是分层的，这就意味着在对象模型中不同的对象之间存在着某种父子关系。例如，一个 Word 文档对象中存在一个 Paragraphs 属性，它返回一个 Paragraphs 对象。这里，文档对象便是 Paragraphs 集合对象的父对象。对于与之相联系的 Paragraphs 对象，正如所期望的一样，使用 Paragraphs.Parent 或者 Paragraphs（1）.Parent 都能返回其父文档。与此类似，一个 Paragraphs 对象同时也是 ParagraphFormat 对象的父对象，所以，这里 Paragraph 对象既是一个父对象也是一个子对象。

关于层次化对象模型，一直存在许多争议。因为在这种模型中，不管我们是否在程序中调用 Parent、Container、Owner 等诸如此类的属性，这些父属性都客观的存在以保持编程的灵活性。层次化对象模型是一种我们经常要用到的一般化的设计结构。如果能够很好的使用，将会使自己建立起来的组件及编程模型显得十分专业化。

在我们在 VB 中创建层次化对象模型时，必须首先决定好系统中的父子关系。如果父对象持有一个对子对象的强引用并且子对象通过一个在父对象中调用的友元函数得到一个父对象的弱引用的话，我们便有了一个建立对象模型的稳固模型。至此，读者也许会认为以上的材料已经足以用来创建大型化的层次化对象模型。毕竟，我们已经可以支持对父对象的弱引用，并且能够在需要调用父对象或者将一个父对象的引用传递到外部时对父对象指针进行“rehydrate”。那么，下面还有什么需要做的呢？

关于使用对象模型，剩下的问题便是我们怎样来创建足够健康的对象。关于对象模型，其健康性的一个基本要求便是应使各对象功能完整，即便对于对象模型中某对象惟一的外部引用来说也是如此。例如，应该使下面的代码予以正确的执行。

```
Dim P As Parent
Dim C As Child
    Set P = New Parent
    Set C = P.Children(1)
    set p = Nothing
    Set P = C.Parent
```

当我们依照顺序阅读这段代码时，可能会觉得有些可怕。但是出于健康性考虑，对象应该在遇到这种情形时能够正确的进行处理。注意，在上面的模型中，Set P=Nothing 释放了父对象，该父对象对持有一个 0 指针的子对象调用 SetParentPtr 友元函数。因此，最后的 C.Parent 将返回 Nothing。

一般来说，子对象只是在父对象的上下文中进行了很好的定义，因此该子对象的功能便不再是完整的。我们称一个处于功能不完整状态的对象为僵化了的对象（zombied）—这时，其方法和属性都不再有效，这是因为创建该对象的上下文已经不存在。一个真正健壮的对象只有在绝对必要的时候才会僵化。例如，在拆分期间部分的被僵化（即不是所有的方法都有效）对于 Visual Basic IDE 扩展模型来说便是有必要的，这是因为与一个对一系列对象的强引用的持有者相反，延展只是对这些对象的一个概览。

我们需要采用两个版本的子对象来防止子对象进入僵化状态。其中，第一个版本是内部的，并且它是被父对象引用的版本。为了保证能够正确地被拆分，在内部版本中包含了对父对象的一个弱引用。此外，我们还需要一个子对象的外部版本。该版本从一个公共过程中返回至外部客户。为了保持对象的健壮性，在对象的外部版本中必须包含一个对父对象的强引用。除此之外，外部版本在其他所有方面均与内部版本保持一致。

我们的目标在于用最少量的代码来协调所有这些要求。也许，僵化对象会显得更为简单一些。但是，僵化对象常常需要在大部分成员函数中加入查错代码，这样对用户来说，便增加了开销和代码的复杂性，并且同时还需增加错误捕获代码。通过预先保持所有公共对象处于一个完全有效的状态，我们便可以从公共过程中移除有关确认代码。事实上，往往一小段附加代码便可以使我们免除许多不必要的麻烦。

对于一个子对象外部版本的要求是双重性的。首先，作为一个外部对象，它必须支持一个内部子对象所支持的所有接口；其次，外部对象必须对所有其他对象采用强引用，而在内部子对象中则是采用弱引用。如果试图直接在 VB 中这样做，将最终会得到一个实现了多重接口（包括子对象）的对象，并且把所有的调用都传递到一个名为 m_RealChild 的成员变量上。很明显，这种方法存在着一些问题：没有人愿意把它写到最开始的位置上；对代码的维护也存在着争议；并且当所有的调用传递到一个内部的成员变量上时会发生一个运行时的性能碰撞。另外，还有一个不那么明显的问题便是：外部对象并不支持对其主接口的后期绑定，除非在实现了子接口之外，再在子类中准确的复制所有的公共过程。实际上，要进行准确的复制非常困难，这是因为作为一个目标对象的设计必须既考虑到整个项目的兼容性和类模块中各成员函数的顺序，同时也必须考虑一些特殊的过程属性。总之，这种方法存在着各种各样的问题，读者大概不会愿意编写这样的代码。

为了构建一个可工作的层次化系统，我们需要在子对象的顶层使用 VBoost 对象来聚合外部对象（参见第五章中“聚合已存在的对象”部分）。这通过 VBoost.CreateAggregate 函数很容易实现。读者可以通过在子类中使用 GetExternalChild 友元函数来实现，这通常只需要少量的代码。新的系统如图 6.4 所示。

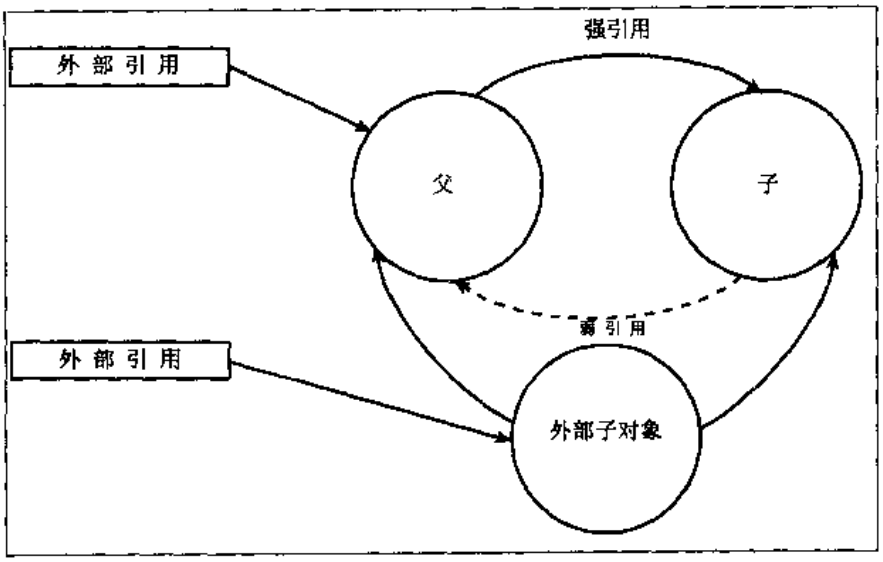

图 6.4 一个外部子对象同样拥有对父对象的一个隐含的强引用

```
'In child.cls
```

```
Private m_ParentPtr As Long
Private m_ExternalPtr As Long
Friend Sub SetParentPtr(ByVal ParentPtr As Long)
    m_ParentPtr = ParentPtr
End Sub
Public Property Get Parent() As Parent
    VBoost.AssignAddRef Parent, m_ParentPtr
End Property
Friend Function GetExternalChild() As Child
Dim AggData(1) As AggregateData
Dim IIDs () As VBGUID
Dim pUnk As IUnknown
    If m_ExternalPtr Then
        'Something is already holding a reference to an
        'external object. Just rehydrate its IUnknown
        'pointer and return
        VBoost.AssignAddRef pUnk, m_ExternalPtr
    Set GetExternalChild = pUnk
    Else
        'Defer all QueryInterface calls to the child
        With AggData(0)
            Set .pObject = Me
            .Flags = adIgnoreIIDs
        End With

        'Let the aggregator hold the parent reference for us.
        With AggData(1)
            'Rehydrate the parent into the AggData structure
            VBoost.AssignAddRef .pObject, m_ParentPtr

            'Tell the aggregator to own this object, but
            'not to publicly expose it via a QueryInterface
            .Flags = adDontQuery
        End With

        'Create the aggregate object and write the pointer of
```

```
        'its controlling IUnknown to the given memory
        'location. This memory is cleared when the external
        'object is destroyed and rehydrated in the mean time.
        Set GetExternalChild = VBoost.CreateAggregate( _
           AggData, IIDs, VarPtr(m_ExternalPtr))
    End If
End Function
```

```
'In Parent.cls
Private m_Children As Collection
Friend Property Get ChildInternal(ByVal Index As Long) As
                          Child
    Set ChildInternal = m_Children(Index)
End property
Public Property Get Child(ByVal Index As Long) As Child
    Set Child = InternalChild(Index).GetExternalChild
End property
```

如果在所有的子对象中使用该代码，子对象将总是对其父对象持有一个有效的引用。在附录“创建聚合”小节中同样列出了该代码的一般形式。事实上，如果能够确信在自己的工程中，内部子引用同样也持有一个有效的父引用的话，那么当一个父对象被销毁时，便无需对每个子对象均调用 SetParentPtr 0。由于我们已经知道，当一个外部的子引用仍然存在时，其父对象不可能被销毁。因此，父对象正在被销毁，便意味着将不再会对其子对象中的公有函数进行调用。这样，也就没有必要再清除弱引用。很显然，这就需要在自己的内部代码中设置规则：必须确信父对象持有惟一的对子对象的引用。设置这样的规则能够在很大程度上对拆分代码进行简化，并同时大大的提高其性能。

对于上面的代码，使用了一个封装器对象的惟一外部根据在于：子对象的 ObjPtr 可以随时间而改变。同时应注意聚合器内部使用了盲 VTable_delegation。这样，在性能方面会存在一个小小的影响，但它所花费的开销通常是可忽略的，尤其是在与为了达到同样效果而需要编写的实现和传递代码相比时更是如此。

第七章

外部对象的创建

在 COM 世界里存在着丰富的公共可见的对象。作为一个 VB 程序员，其中一个任务便是为其添加一系列公共对象，并使自己及组织中的其他成员能够利用它来建立应用。理解怎样创建外部对象是使用对象的重要一环。本章将介绍 VB 是怎样通过使用 New、CreateObject、GetObject 以及基于 API 的方法来进行对象创建的。

在能够创建一个外部 COM 对象之前，VB 必须首先能够在系统中为对象定位。HKEY_CLASSES_ROOT\CLSID 注册段把一个 CLSID(Class Identifier)映射到本地系统或某个远程系统中的某一物理位置。CLSID 只是 COM 使用 GUID（全局惟一标识符）的一种方法，GUID 是一个用来保证在整个系统中惟一的长度为 128 位的结构。为了支持外部对象的创建，一个 ActiveX 服务器为每一个所支持的对象注册了一个 CLSID，并且发布一个在服务器类型库中所支持类的列表。类型库，一般被做成服务器中的一个可执行文件以作为资源，它包含了对库中支持类以及这些类所支持的接口的描述。

在 VB 中，采用了两种机制来为 COM 提供一个 CLSID，并允许 COM 能够创建一个外部对象。对于第一种机制，VB 把一个工程引用添加到含有 CLSID 的类型库中。这是创建对象时首选的一种机制；经编译后的 VB 代码知道它必须创建的是哪一个 CLSID 而并不需要进行查询。在第二种机制中，VB 在运行时通过一个对对象来说颇为友好的名字（一般称为 ProgID）来查看 CLSID。一个 ProgID 通常采用 ServerName.ClassName 的形式。该字符串同样也存储在系统注册表中并简单地指向 CLSID 关键字。不幸的是，ProgID 这一字符串值并不完全可信。这是因为不能保证该字符串是惟一的。如果某个特定 ActiveX 服务器的用户完全依赖于 ProgID，而另一个 ActiveX 服务器又正好注册了相同的 ProgID 的话，则外部对象实际上将会从系统中移除。这样，该关键字潜在地被重写，同时还要求额外的注册表查询。这时候，在可能的情况下最好直接使用 CLSID。

如果使用了 VB 中的 New 关键字，编译后的代码在运行时便总能知道 CLSID。如果使用 CrearObject，VB 通过使用存储在注册表中的 ProgID 来找到 CLSID。这样，对于所创建的每个对象都产生了一个额外的系统调用。使用 CreatObject 同样容易导致名字冲突，因为

ProgID 不是一个惟一的标识符。

另外，使用 New 和 CreatObject 会返回不同的初始接口。对于 New，正如类型库中所描述的一样，它所要求的第一个接口是类的缺省接口。如果在另一台机器、另一个进程或另一个线程中创建一个对象，一般我们都希望它只具有所需要的接口。这是因为附加的接口将增加开销。而对于 CreatObject，它返回 IDispatch(作为一对象)接口，如果不能得到 Idispatch 则返回 IUnknown 接口。任何对一个附加接口（包括对象的主接口）的要求都需要使用 Set 语句。

在大部分应用中，New 和 CreatObject 对外部 COM 对象的创建来说都是适用的。由于 VB6 在 CreatObject 中引入了第二机器名参数，从而使我们能够指定一个希望在其之上创建对象的 DCOM 服务器。尽管在一般情况下，这一新特征避免了以前必须使用 API 函数来创建一个对象的情况，但是相比于 VB 中内建的创建函数，API 仍然提供了更多的灵活性。下面，就让我们来看一看怎样才能使用 VB 中的 GetObject 函数以及 CoGetClassObject API 函数来为任意的 CLSID 定位一个类工厂。这样，就能使得我们可以从除创建一个指定对象类的第一个实例之外的所有实例中除去对象创建机制所占用的空间。所有本章代码中需要的 API 函数及接口的定义都在 ObjCreate.Olb（VBoost:Object Creation and Security）中进行了描述。并且，我们假定工程中已经包含了对 VBoostTypes6.Olb 及一个实例化的 VBoost 变量的引用。

7.1 使用类工厂进行对象的创建

在 COM 规范中，并没有包含一种用来在 DLL 或 EXE 中直接创建对象的机制。COM 中为了完成这项工作，采用了一种被称为 IClassFactory 的中间接口，而该接口实际上是一种用以创建对象的系统标准。当我们调用 CreatObject 或 New 时（它被映射至 CoCreateInstance 和 CoCreateInstanceEx API 函数上），实际上是告诉 COM 为 CLSID 定位一个 IClassFactory 实例。然后，COM 再调用 IClassFactory.CreatInstance 方法来创建一个所要求的对象。

当我们想要创建同一类型的多个对象时，如果不断地释放和重新定位类工厂（class factory）的话，将会导致多余的循环处理过程。实际上，如果我们在一次性的得到一个类工厂之后再多次调用 CreateInstance 的话则会有效的多。例如，在第 13 章关于多线程的讨论中，我们将看到，为了在不同的线程中创建对象，必须对自己工程中的对象使用 CreatObject 方法。其实，在主线程中取得一个类工厂，然后再调用 CreateInstance 方法，并发送到各个工作线程上要更为有效一些。

要得到一个类工厂接口，最简便的方法便是使用内建系统中的 clsid 标识符和 GetObject 方法。下面的代码所完成的功能与使用 CreatObject 方法相同。它没有调用对象初始化接口（参见本章中后面的“初始化可持续对象”部分）。然而在这里，该函数返回了一个类工厂，最后一步即调用 CreatInstance 方法则留给函数调用者去完成。

```
Public Function CreateClassFactory( _
    CLSID As CLSID) As IClassFactory
Dim GuidString As String * 38
    StringFromGUID2 CLSID, GuidString
    Set CreateClassFactory = GetObject("clsid:" & GuidString)
End Function

'Calling Code, roughly equivalent to CreateObject
Dim MyObj As MyObject
Dim MyObjeCF As IClassFactroy
Dim IID_IUnknown As IID
    IID_IUnknown = IIDFromString(strIID_IUnknown)
    Set MyObjF = CreateClassFactory( _
      CLSIDFromProgID("MyProj.MyObj"))
    Set MyObj = MyObjCF.CreateInstance(Nothing, IID_IUnknown)
```

通过显式的引用类工厂，我们可以选择是采用 CLSID 还是采用 ProgID 作为识别目标对象的主要方法。实际上，包括 Windows 2000 在内的所有 Win32 平台都要求我们遍历整个类工厂而跳过整个 ProgID-resolution 代码。Windows 2000 允许在 GetObject 中使用“new:”来代替“clsid:”以直接创建单个实例，这样就避免了获取类工厂。

7.1.1 获取类工厂

获取一个类工厂的标准机制是可变的，并依赖于我们所得到的对象是来自于 DLL、一个局部 EXE 还是 DCOM。下面，让我们来仔细考察一下 DLL 及局部 EXE 的结构，以便能够更好地控制对象的创建。

通过调用 CoGetClassObject API 函数便可以对所有获取类工厂的进程进行管理。该 API 函数共包括 4 个参数。第一个参数为所需创建对象的 CLSID，最后一个参数则是一个接口的 IID，由该接口我们期望通过 IID_IClassFactory 来调用对象。而其他参数，dwClsContext 和 pServerInfo，则提供了从中创建对象的上下文信息（DLL、EXE 或 DCOM）。如果是远程对象的话，还包括 DCOM 服务器的有关信息。

除了三个最一般的值隐藏于 VBoostTypes 中，其上下文值全部在 CLSCTX enum 中列出。这三个 VB 中通常使用的隐藏值分别为 CLSCTX_INPROC_SERVER、CLSCTX_LOCAL_SERVER 和 CLSCTX_REMOTE_SERVER。这些上下文值分别从一个 DLL、EXE 或 DCOM 服务器中加载一个对象。与调用者位于同一线程中的一个类工厂被称作 INPROC，即便它是由 EXE 而不

是 DLL 提供时也是如此。如果指定了不只一个值并且多种上下文都支持同一个 CLSID 的话，COM 将把它解析为最有可能的上下文值（优先次序分别为 DLL、EXE、DCOM）。

当多个运行着的服务器都支持同一个 CLSID 时，可能会导致对上下文的解析产生模糊。当 EXE 被当作一个独立的应用来使用时，我们通过发送同一 ActiveX EXE 的多个实例便可以很容易的看到这个问题。一个由 VB 创建的 ActiveX EXE 仅仅在主线程中注册类工厂。这意味着一个 CLSCTX_INPROC_SERVER 需要位于同一 EXE 中的一个多用途对象。并且这只对主线程适用，而不适用于其他线程。如果第二个线程使用 CLSctx_LoCAL_SEVER 返回一个类工厂，并不保证这个类工厂来自同一个进程的第一个线程。实际上，所返回的类工厂总是来自于注册该类工厂的第一个 EXE 实例。其解决办法是指定 CLSCTX_INPROC_SERVER 或 CLSCTX_LOCAL_SERVER 为上下文参数。CoGetClassObject 将对这一要求进行特殊处理，并且在一个类工厂可得的情况下，总是从同一进程中返回该一个类工厂。

如果不想创建一个 DCOM 对象或者想要使用当前注册的服务器来创建一个对象的话，应将 pServerInfo 参数设置为 0。通过 COSERVERINFO 结构能够识别一台远程机器并提供用户的有关证明和授权信息。尽管我们已经定义了所有的 COM 安全性结构和 ObjCreate 类型库中的 API 函数，但这里并不想就安全性这一主题展开讨论。读者如果想了解这一方面的详细信息，可以参考 Keith Brown 所著的《Programming Windows Security》一书。为了从一台假定具有足够安全性的机器中获取一个类工厂，使用了如下的程序代码。其中，CreateRemoteFromCLSID 在功能上等同于带有一服务器名参数的 CreateObject 方法。

```
Public Function CreateRemoteFromCLSID( _
  ServerName As String, CLSID As CLSID) AS IUnknown
Dim ServerInfo As COSERVERINFO
Dim pCF As IClassFactory
    ServerInfo.pwszServerPrincName = StrPtr(ServerName)
    Set pCF = CoGetClassObject(CLSID, CLSCTX_REMOTE_SERVER, _
       VarPtr(ServerInfo), IIDFromString(strIID_IClassFactory))
    Set CreateRemoteFromCLSID = pCF.CteateInstance( _
       Nothing, IIDFromString(strIID_IUnknown)
End Function
```

DLL 和 EXE 之间有着本质上的差异。这种差异在于：一个 EXE 控制了它的整个生命周期，而 DLL 却没有。DLL 总是被加载它的 EXE 进程所控制。而且，与 EXE 不同的是，一个 DLL 能够导出函数。这些差异导致它们在应用类工厂方面采用不同的机制。

COM 通过使用导出的 DllGetClassObject 函数来从一个 DLL 中获取一个类工厂。具体过程是：COM 首先装入由注册表中 CLSID 段中的 InproServer32 项所指定的 DLL，然后使用 GetProcAddress 来定位 DllGetClassObject 项，并调用该函数指针来为所要求的 CLSID 得

到一个类工厂。一旦进程内的类工厂返回到请求一个对象的时候，COM 便完全脱离了对象之间的相互影响。对于 COM 来说，所剩下的责任只是对 DLL 进行卸载。我们可以调用 CoFreeUnusedLibraries API 函数，并通过该函数来为由 CoGetClassObject 所加载的所有 DLL 调用 DllCanUnloadNow 条目项，从而查看一下当前的线程是否释放了该 DLL 中的所有对象以及类工厂。当 DllCanUnloadNow 成功返回时，该 DLL 便被 FreeLibrary API 函数从内存中移除。这时如果显示了一个表格，VB 便会调用 CoFreeUnUsedLibraries。不过，如果能够确信有一个或多个 COM DLL 将不再被使用，那么任何时候也都可以通过调用该函数来对其进行卸载。

对于 EXE 的情况，COM 将从 CLSID 段读取 LocalServer32 关键字来定义一个 EXE 服务器，然后发送该进程。EXE 通过调用 CoRegisterClassObject API 函数来注册所有 COM 中所支持的类工厂。一旦该对象被完全注册，COM 便在内部表中查看所请求的 CLSID 并返回一个类工厂引用给调用者。如果 COM 已经拥有了一个 IClassFactory 引用，那么它将使用已存在的类工厂而不是重新发送该 EXE。在进程终止时，EXE 使用 CoRevokeClassObject 来取消该类工厂的注册。

7.1.2 初始化可持续对象

如果采用一个类工厂而非采用“New”或“CreateObject”创建了一个外部对象的话，我们实际上通过 VB 便完成了初始化一个新对象所需的几乎全部工作。然而，VB 中还采用了额外的一个步骤来确保由它所创建的所有对象都被完全地初始化。紧接着在创建了一个对象之后，VB 试图取得 IPersistStreamInit 接口以使得能够调用 InitNew 方法。如果该外部对象由 VB 所创建且其 Persistable 属性被设置为 1-Persistable，该代码便运行 Class_InitProperties 事件（如果已经对其进行定义）。如果 VB 不能得到一个 IPersistStreamInit 接口的话，它便试图得到一个 IPersistPropertyBag 接口。IPersistPropertyBag 也同样拥有自己的 InitNew 方法。

ObjCreate.Olb 中包含了对接口的定义从而允许使用这些接口来对对象进行初始化。另外，如果我们确知所创建的对象在 InitNew 中实际上并没有完成任何功能，便可直接跳过这些有关初始化的代码。下面的程序片段中使用到了我们在前面曾经用到过的 CreatClass Object 函数，但不同的是在下面的程序片段中，该函数也试图对对象进行初始化。

```
'Calling Code, even closer to CreateObject
Dim MyObj As MyObject
Dim MyObjCF As IClassFactory
Dim IID_IUnknown As IID
Dim pInit As IPersistStreamInit
    IID_IUnknown = IIDFromString(strIID_IUnknown)
    Set MyObjCF = CreateClassFactory(CLSIDFromProgID( _
```

```
    'MyProj.MyObj"))
Set MyObj = MyObjCF.CreateInstance(Nothing, IID_IUnknown)
On Error Resume Next
Set pInit = MyObj
On Error GoTo 0
If Not pInit Is Nothing Then pInit.InitNew
```

7.2 直接加载 DLL 对象

尽管我们可以对 CoGetClassObject 方法从一个 DLL 中取得一个类工厂的所有步骤进行复制，但是对我们来说，并不存在任何固有的原因使得我们需要依赖一个注册项来对一个 DLL 进行定位或加载。基于可执行的路径而非系统注册表中的某条路径来加载一个 COM 对象具有两大方面的优势。

首先，我们可以在同一线程中使用 DLL 而不用由 COM 对其进行注册。这种方法在发行版本上的优势是多方面的。我们可以分发同一 DLL 的多个版本并隔离每个版本以完全消除与先前所安装的使用了同一 DLL 的应用程序相冲突的可能。如果我们计划在线程和进程之间传递一个对象，便必须注册类型库，但即便是这种情况，我们也无需使用注册表来定位和加载该 DLL。

其次，我们可以加载多个相同的 DLL 并使用同样的代码来控制它们的对象。可以很容易地对多个 DLL 工程使用同一二进制兼容文件来给这些 DLL 赋予同一套 CLSID 和 IID。这样，我们就可以定义一系列接口来为主应用程序增强可兼容性，然后即可通过在应用程序目录下的某个子目录或数据文件中添入或移除 DLL 来动态地改变整个应用程序。通过直接地加载 DLL 以及对它们进行卸载或更新，我们可以在不用关闭应用程序的情况下对它们进行更新。这里需要注意的是，应当对每个 DLL 采用相互之间不产生冲突的基地址。

MSDN 文档中曾经提到过，CoLoadLibrary 函数使用 bAutoFree 参数来说明一个 DLL 若不再被使用，它将在调用 CoFreeUnusedLibraries API 函数时自动地被卸载。读者千万不要过分的相信该文档（除有关不应直接调用 CoLoadLibrary 的说明以外）。如果在系统符号都处于正确位置的情况下逐步的调用 API 函数，我们会发现，bAutoFree 参数被忽略。CoLoadLibrary 只不过是在 LoadLibraryEx 的基础之上进行了一下小小的包装。CoGetClassObject 为 CoFreeUnusedLibrary 维护一个所加载库的列表而无需使用 CoLoadLibrary。由于直接加载不需要使用 CoGetClassObject，这就意味着无需使用 CoFreeUnusedLibraries 来隐式地释放 DLL。

CoLoadLibrary 所做的一件特殊的事情便是使用 LOAD_WITH_ALTERED_SEARCH_PATH 标志来调用 LoadLibraryEx。当我们为 DLL 说明整个路径时，该标志便告诉 Windows 应从 DLL

目录下而非应用程序当前使用的目录下去搜索所依赖的 DLL 文件。如果我们从另一个 DLL 而不是 EXE 中调用 LoadLibraryEx 的话，那么使用 DLL 搜索路径对我们来说将显得格外的重要。

我们可以直接使用 LoadLibrary[Ex]来为 Declare 调用指定一个 DLL 的路径。这种方法在 DLL 工程中显得特别有用。该方法试图从加载类库的 EXE 文件的路径出发来对类库名进行解析。一个 DLL 文件中的 Declare 声明语句甚至不能找到同一目录下的另一个 DLL，除非该目录也是应用程序路径的一部分。如果不能直接确定 DLL 文件所在的具体路径，但是知道该文件相对于 EXE 或 DLL 文件的相对路径的话，那么可以为 API 调用设置一个错误陷阱。如果发生了文件错误，会首先使用到 DLL 的全路径来调用 LoadLibrary，然后执行在 VB 中进行了声明的 API 函数，最后调用 FreeLibrary。因为 VB 一旦使用 Declare 调用加载了一个类库，它不会轻易的释放该类库，所以先前的 API 函数声明对随后的调用仍然适用。而使用在类型库中定义的函数声明时，就没有必要如此费力，因为在进程载入时，所有的声明已经被解析了。并且，不应在 Declare 声明中硬性地指定任何路径，以免将程序锁定到某个指定的安装目录上。

下面这些函数载入 DLL 并且提取出一个类工厂，然后检查一下看一看 DLL 能否被卸载掉。这一代码可以在 COMDllLoader.Bas 中找到，它使用 FunctionDelegator（我们将在第 11 章对其进行讨论）来调用 DllGetClassObject 和 DllCanUnLoadNow 函数指针。

```
'Requires a reference to ObjCreate.olb and VBoostTypes6.olb
'FunctionDelegator.Bas must also be loaded in the project.
Private m_fInit As Boolean
Public IID_IClassFactory As IID
Public IID_IUnknown As IID
Private m_FDDllGetClassObject As FunctionDelegator
Private m_pCallDllGetClassObject As ICallDllGetClassObject
Private m_FDDllCanUnloadNow As FunctionDelegator
Private m_pCallDllCanUnloadNow As ICallDllCanUnloadNow

Private Sub Init()
    IID_IClassFactory = IIDFromString(strIID_IClassFactory)
    IID_IUnknown = IIDFromString(strIID_IUnknown)
    Set m_pCallDllGetClassObject = _
        InitDelegator(m_FDDllGetClassObject)
    Set m_pCallDllCanUnloadNow = _
        InitDelegator(m_FDDllCanUnloadNow)
    m_fInit = True
```

```
End Sub
Public Function GetDllClassObject(ByVal DllPath As String, _
    CLSID As CLSID, hModDll As HINSTANCE) As IClassFactory
        If Not m_fInit Then Init
        If hModDll = 0 Then
            hModDll = LoadLibraryEx( _
    DllPath, 0, LOAD_WITH_ALTERED_SEARCH_PATH)
        If hModDll = o Then
            Err.Raise &H80070000 + Err.LastDllError
        End If
    End If
    m_FDDllGetClassObject.pfn = GetProcAddress( _
        hModDll, "DllGetClassObject ")
    If m_FDDllGetClassObject.pfn =0 Then
        Err.Raise &H80070000 + Err.LastDllError
    End If
    'The function declaration specifies an HERSULT return
    'value, so this can raise a standard error.
    Set GetDllClassObject = m_pCallDllGetClassObject.Call ( _
        CLSID, IID_IClassFactory)
End Function

Public Sub TestUnloadDll(hModDll As HINSTANCE)
    If Not m_fInit Then Init
    If hModDll Then
        m_FDDllCanUnloadNow.pfn = GetProcAddress( _
            hModDll, "DllCanUnloadNow")
        If m_FDDllCanUnloadNow.pfn = 0 Then
            Err.Raise &H80070000 + Err.LastDllError
        End If
        If m_pCallDllCanUnloadNow.Call = 0 Then
            FreeLibrary hModDll
            hModDll = 0
        End If
    End If
End Sub
```

为了能够对 DLL 进行加载和卸载，应在 DLL 中确定对象的 CLSID 和对象的位置，并在载入 DLL 后跟踪它的模块句柄。如果不将 DLL 的兼容性程度设为工程级或二进制级，CLSID 是一个活动的目标。有很多方法可以用来确定 CLSDLD。读者可以使用低层次的类型库浏览器（例如 OleView.Exe）、VB 自带的 TLI 对象（类型库信息对象）或者本书中带的一个叫 DumpClassData.Exe 的实用工具。DumpClassData 为类型库中的可创建类产生一个 CLSID 常量，并且为 DLL 或 OCX 中的已注册对象生成运行时注册关键字。通过在命令行下键入文件名或在对话框中进行选择，就可以运行 DumpClassData。无论使用哪一种工具，都会产生类似下面的剪贴板数据。

```
'Target control
Private Const strCLSID_Target As String = _
    "{C33A1760-A296-11D3-BBF5-D41203C10000)"
Private Const RTLic_Target As String = "mkjmmmrlhmknmo"
```

一旦获得了 CLSID，就可以使用下面的调用代码来载入和卸载 DLL。这些代码从单个 DLL 中载入，并且它不缓存类工厂。读者可以对下面的代码作出修改来对多个 DLL 进行跟踪或者缓存类工厂。

```
Const strDllObjectCLSID As String = _
    "{C33A1760-A296-11D3-BBF5-D41203C10000)"
Dim CLSID_DllObject As CLSID
Dim m_hModMyDll As Long

Sub CreatMyDllObject(RelPath As String) As MyDllObject
Dim pCF As IClassFactory
    If m_hModMyDll = 0 Then
        'Load or reload the DLL
        CLSID_DllObject = GUIDFromString(StrDllObjectCLSID)
        Set pCF = GetDIIClassObject(App.Path & "\ " & _
            RelPath, CLSID_DllObject, m_hModMyDll)
    Else
        Set pCF = GetDllClassObject(vbNullString, _
            CLSID_DllObject, m_hModMyDll)
    End If
    'Create the instance. IID_IUnknown is declared in
    'COMDllLoader.Bas.
```

```
    Set CreateMyDllObject = _
        pCF.CreateInstance(Nothing, IID_IUnknown)
End Sub

Sub UnloadDll()
    TestUnloadDll m_hModMyDll
End Sub
```

只有在应用程序是单线程的或 DLL 支持单元线程时，才有必要对 DLL 加载作出控制。不要使用类似“load-it-yourself”的代码，因为这样会破坏 COM 的单元规则。如果目标 DLL 是单线程的，COM 会在一个单独的线程上创建所有的对象，并且它会提供一个代理对象而不是一个直接创建的对象。幸运的是，很容易对工程的属性作出改变并使 VB 创建的 DLL 单元线程化。

7.3 自定义加载定制化控件

一个 OCX 控件实际上就是一个带有文件扩展的 ActiveX DLL。一个 OCX 控件和一个 DLL 中都出现了同样的 4 个条目项。然而 OCX 通过改变一些类型库标志和注册表设置而以不同的形式出现。一个 OCX 类型库的库属性由控件属性所标记，这就防止了 VB 在工程/引用对话框中列出该库并作为一个可得到的选择。在类型库中对每一个 OCX 控件的 coclass 定义同样也以控件属性及不可创建属性为标记。该属性锁定了用“New”关键字来进行对象的创建。一个已注册的控件同样也在它的 CLSID 字段中注册了一个控制关键字，并用它来通知 VB 在工程/组件对话框中的“控件”项中显示该项控件。

读者可能会发现，将一个 DLL 转变成 OLX 控件的各种方法表面上只是包含了一系列的类型库属性和注册表设置。而当我们一旦深入到 DLL 内部，则会发现在 OCX 和 DLL 之间没有本质上的不同。正如处理一个标准的 DLL 一样，COM 通过为 IClassFactory 调用 DllGetClassObject 方法，然后再在类工厂中调用 CreateInstance 方法来从 OCX 中载入控件。当从 VB 中的一个 OCX 中使用对象时，我们用到了另一个不同的对话框来将对象加入到自己的工程中。这样，VB 便可以自动的处理对象的创建工作了。

当我们从一个 OCX 控件中自定义加载一个对象时，必须考虑有关注册的问题。而如果是在一个 DLL 中使用对象时，则未必需要对这一问题进行考虑。为了创建一个已注册的控件，我们需要用到注册表中 HKEY_CLASSES_ROOT\Licenses 段中的一个特殊条目项，或者需要在 IClassFactory2 接口上使用 CreateInstanceLic 方法而不是在 IClassFactory 接口上使用 CreatInstance 方法。如果并没有提供常规的注册关键字，CreateInstanceLic 方法将会采用一个字符串参数，该参数将被用来创建一个控件实例。对于一台其上所有控件都在设计时

进行了注册的机器来说，同样可以从 ICassFactory2 接口上得到该运行时注册字符串。我们可以利用在本章前面部分曾经提到过的 DumpClassData 工具来产生除 CLSID 值之外的运行时注册关键字。不过，这里需要注意的是，只有对于那些在当前机器上具有设计时注册的控件来说，该注册字符串才是可得的。

由于在这里并没有使用 New 或 CreateObject 来加载一个定制控件，我们也就不能像对待 DLL 中的对象那样，使用 COMDllLoader.bas 中的函数来直接加载一个 COM 控件。然后再重新把所有的对 VB 所创建控件的要求下达给一个已从一个 OCX 定制装入的控件对象。对于 VBoost 对象来说，要使一个对象看起来与另一个对象一样是一件很容易的事。不过，在给出其代码之前，有必要提及的是：这项技术还存在着一些缺陷。因此读者必须自己来考虑这个问题，即将定制 DLL 加载所具有的代码重用等优势舍弃掉而引入这些限制的做法究竟是否值得。

- 定制 OCX 加载中不允许使用属性表单来设置本地控件属性，不过，所有的扩展属性以及属性页都是完全可以被访问的。而且，在设计时所设置的任何属性都将被正确无误地写入到 containing.Frm，.ctl(用户控件)，或者.Dob（用户文档）文件中，并且当控件在运行时初始化这些属性会被反映出来。读者可以参看第 4 章中的“VTable 绑定定制控件接口”部分以获取有关定制控件本地和扩展属性方面的更多的信息。
- 定制 OCX 加载中并不允许通过控件变量直接接受本地控件事件。取而代之，它必须通过常规机制来接收扩展事件（例如获取焦点等），并且使用一个单独的 WithEvents 变量来取得本地事件。
- 用户必须锁定对 IQuickActivate 接口的支持，该接口使加载一个定制控件的性能得到提高。IQuickActivate 接口并不能在所期望的事件接口与自定义加载控件实际所支持的接口之间进行很好的协调。

如果用户能接受上述这些限制的话，那么便可以像 DLL 中一样自定义的加载 OCX 文件了。用户仅仅需要知道该 OCX 的位置（相对于应用程序路径）以及该控件的 CLSID 就可以了。然后动态的加载控件、添加一个新的用户控件至自己的工程中并且为想要加载的每一个控件运行如下代码。这里，用户需要自己设置的惟一属性是 WindowLess 属性，该属性应和试图加载的控件相匹配。另外，在用户自己的工程中还必须具有一个可得到的 VBoost 对象、一个对 ObjCreate.olb 的引用以及 OcxLoader.Bas、COMDllLoader.Bas 和 FunctionDelegator.Bas。

```
'wrap
Private Const strCLSID_Target As String = _
  "{662AE532-AFF5-11D3-BC07-D41203C10000}"
Private Const strTargetOCX As String = "Target.ocx"
Private m_OcxloadData As OcxLoadData
```

```
Private Sub UserControl_Initialize()
    LoadWrappedOcx m_OcxloadData, Me, _
      strTargetOcx,GUIDFromString(strCLSID_Target)
End Sub

Private Sub UserControl_Terminate()
  UnloadWrappedOcx m_OcxLoadData
End Sub
```

```
'Ocxloader.Bas
'External dependencies:
'COMDllLoader.Bas
'FunctionDelegator.Bas

 Public Type OcxLoadData
    hInstDll As Long
    Hook As UnknownHook
End Type

Public Type  OcxTargetData
    TargetOcx As String
    CLSID As CLSID
    RTLicenseKey As String
    fTargetAbsolute As Boolean
End Type
Public g_OcxTargetData As OcxTargetData

Private Type OcxHookAggData
    AggData(1) As AggregateData
    pIIDs(0) As IID
End Type

'Load an OCX into the Wrapper object based on a relative
'path. This should be called from the UserControl_Initialize
'event of the wrapper UserControl class.
'
'TargetOcx:  Path to the OCX relative to the application ,
'          unless fTargetAbsolute is ture,in which
```

```
'          case this is an absolute path.
'CLSID_Target:  CLSID supported by the wrapped OCX.
'RTLicenseKey:  The key to create a control that doesn't
'                   have a registered license key. Can be
'                   obtained from IClassFactory2.RequestLicKey
'                   using the GetRTLicKey utility on a machine
'                   where the control is fully licensed.
'fTargetAbsolute: An optional flag indicating that the
                     TargetOCX argument is an absolute, not a
                      relative, path.
Public Sub LoadWrappedOcx( _
  LoadData As OcxLoadData,_
  ByVal Wrapper As IUnknown, _
  TargetOcx AS String, _
  CLSID_Target As CLSID, _
  Optional RTLicenseKey As String, _
  Optional ByVal fTargetAbsolute As Boolean = False)
Dim HookData As OcxHookAggData
Dim pCF As IClassFactory
Dim PCF2 As IClassFactory2

      'If this is used at design time,then Sub Main may
      'not have run, so the VBoost object has not been
      'created .
      If  VBoost Is Nothing Then InitVBoost

      'Use the helper function to load the OCX.
      If fTargetAbsolute Then
         Set pCF =GetDllClassObject (TargetOcx, _
           CLSID_Target,LoadData.hInstDll)
       Else
         Set pCF= GetDllClassObjcet(App.Path & "\"&
                                    TargetOcx, _
           CLSID_Target,LoadData.hInstDll)
       End If

       'See about licensing.
        If Len(RTLicenseKey) Then
```

```
            Set pCF2 = pCF
            With pCF2.GetLicInfo
                If.fLicVerified Then
                    'pCF.CreateInstance will succeed
                    Set pCF2 = Nothing
                Else
                  If 0 = .fRuntimeKeyAvail Then
                      'Can load only a licensed control
                      TestUnloadDll loadData.hInstDll
                      Exit Sub
                   End If
                End If
            End If
        End With
    End If
    '
'Aggregate the created object with priority before the
'wrapping object.The adNoDelegator removes the
'per-interface overhead of the aggregator ,but breaks COM
'identity with the wrapping object.In practice ,we don't
'really care much about the wrapping object,and there are
'a substantial amount of interfaces that need to be
'wrapped with a custom control,so we use adNoDelegator.
  With HookData .AggData (0)
     .Flags = adIgnoreIIDs Or adBeforeHookedUnknown _
      Or adNoDelegator
       If pCF2 Is Nothing Then
           Set .pObjcet = pCF.CreateInstance ( _
            Nothing,IID_IUnknown)
       Else
           Set .pObject= pCF2.CreateInstanceLic( _
             Nothing,Nothing,IID_IUnknown,RTLicenseKey)
       End If
End With

'Block IQuickActivate
With HookData.AggData(1)
    HookData.pIIDs(0) = IIDFromString(_
```

```
            strIIDI_QuickActivate)
          .FirstIID = 0
          .Flags = adBlockIIDs
    End With

    VBoost.AggregateUnknown _
      Wrapper,HookData.AggData,HookData.pIIDs,
          LoadData.Hook
End Sub

'Call LoadWrappedOcx using the global data
Public Sub loadWrappedOcxFromGlobalTargetData(_
  loadData As OcxLoadData,ByVal Wrapper As IUnkown)
    with g_OcxTargetData
      LoadWrappedOcx LoadData,Wrapper,_
      .TargetOcx , .CLSID, .RTLicenseKey,.fTargetAbsolute
  End With
End Sub

'Unload the OCX file.This should be called from the
'UserControl_Terminate event of the wrapper control.
Public Sub UnloadWrappedOcx(LoadData As OcxLoadData)
       'Unhook to release all references on the wrapped object.
       Set LoadData.Hook = Nothing
      'Let go of the OCX file if no one else is using it .
      TestUnloadDll LoadData.hInstDll
End Sub
```

7.3.1 Control.Add 和动态控件

上面所列出的示例代码对每一个封装起来的控件都使用了一个用户控件文件并用到了经过硬编码（hard coded）后的类信息。不过，这并不是使用 OCX 封装代码所必需的要求。这里，惟一的要求是在用户控件初始化时必须要能够取得用以加载类工厂的有关信息。由于 VB 中不允许传递参数给初始化事件，用户便需要对这些值进行硬编码或者从全局变量中读取这些值。如果在设计时将控件装入到表格中，说明用户想要使用这些私有值。如果想让整个过程动态的进行并用 Control.Add 加入自己的控件，假如 OcxLoader.Bas 中的 g_OcxTargetData 变量具有所有加载信息，我们可以使用一个用户控件封装器（wrapper）。

最终的代码应该大致如下。

```
'DynamicLoad UserControl Class
Private m_OcxLoadData As OcxLoadData

Private Sub UserControl_Initialize()
    If Len(g_OcxTargetData.TargetOcx) = 0 Then Exit Sub
    LoadWrappedOcxFromGlobalTargetData _
        m_OcxLoadData, Me
End Sub
Private Sub UserControl_Terminate()
    UnloadWrappedOcx m_OcxLoadData
End Sub
```

```
'Snippet to load the control
Dim ctl As VBControlExtender
    With g_OCXTargetData
        .CLSID = GUIDFromString( _
            "{662AE532-AFF5-11D3-BC07-D41203C10000}")
        .RTLicenseKey = "mkjmmmrlhmknmo"
        .TargetOcx = "Target.Ocx"
        .fTargetAbsolute = False
    End with
    Set ctl = Controls.Add("MyProject.DynamicLoad ", "ctl1")
```

对于这种情况，有一些问题需要处理。首先，如果所要加载的既包含带有窗口的控件也包含不带窗口的控件，则对每一种类型的控件都需要一个动态安装类型的用户控件。其次，如果已经移除未用过的 ActiveX 控件选项的有关信息（用户可在工程/属性对话框中的 Make 项中的底部选择该选项），便不能再成功地对一个不包含在表格中的控件调用 Control.Add 方法。之所以设计该选项是为了省去用以支持那些当前工程中并没有用到的控件所花费的开销，并对工程中的控件以及实际上并不在表格中的所有工具箱控件禁用 Controls.Add 方法。用户可以清除这一选项（缺省情况下为有效），或者只把一个动态加载控件拖入到至少一个表格中并将该控件的 Visible 选项设置为 False 就行了。

把所有表格动态的载入一个 MDI 应用的能力体现了动态控件加载的强大功能。通过把单个动态加载控件置入一个 MDIChild 表格中并在创建一个新表格之前填充好 g_OcxTarget Data 结构可以达到等同于使用 Controls.Add 方法的效果。Form_Resize 中的一行代码赋予了动态装入的 OCX 对由自己创建的表格进行完全控制的能力。如果拥有用来支持菜单的表格，则只需要将菜单加入到用户控件中再将 NegotiatePosition 设置为一个非零值，并把表格

的 NegotiateMenus 设为 True 就可以了。现在我们便有了一个已包含少量代码的轻量框架。利用该框架，甚至无需用到注册表便可以动态地将表格加入到一个 MDI 应用中。

在本书中所提供的 DynamicMDI 示例中，包含了怎样用一个带有与应用程序相关的 OCX 文件的数据文件来填充 MDI 菜单和装入表格的使用说明。DynamicMDI 还在 LookupCLSID.Bas 中包含了用以注册一个公共上下文对象的样例代码（本章中的“定制类对象”部分对其进行了讨论），并决定了基于某一类名的一个 OCX 和 DLL 中的 CLSID。该样例代码的结果是生成一个带有由动态加载、未进行注册的 OCX 所提供的子表格且易于定制的数据驱动的 MDI 框架。这里，对于单个的 OCX 并没有特别的接口要求。

7.3.2 使用属性、方法和事件

创建一个控件当然是一个很好的选择，但要是不能设置控件的属性、调用控件的方法以及对由控件所产生的事件作出响应的话，那么控件本身并不能带来太多的好处。简而言之，用户最终将只是得到一个精致美丽的 UI（用户接口），却没有办法与其进行交互。VB 仅仅知道一个瘦封装（thin wrapper）控件的有关接口和事件，而对由用户自己动态装入的控件的有关接口和事件则一无所知。如果用户想要与这些控件进行交互则必须让 VB 编译器知道这些由自己封装的控件的有关接口和事件。在 VB 中任何与本地接口的直接交互都要求有一个类型库的引用。

加入一个定制控件的引用并不像加入一个对标准 DLL 的引用那么直接。VB 中隐含的从可得到引用列表中去除了具有控件属性的所有类型库。为了使用工程/引用对话框加入一个对控件的引用，用户必须单击“浏览”按钮并找到对应文件。这里需要注意的是：在 VB6 中存在一个缺陷，它使得我们可以加入这些隐藏了的引用，但是当我们重新打开工程/引用对话框时，这些引用并不出现在当前引用列表中。我们必须手工编辑 VBP 文件并重新加载整个工程来去除这些引用。

如果我们的工程包含文件中已经有了一个直接类型库的引用，我们只需要声明一个正确类型的变量（如果需要可使用 WithEvents）并使用 Set 语句来从控件的对象属性得到该本地对象。这样，我们就可以使用该变量来操纵该控件的原有部分并使用 VBControlExtender 对象来操纵其扩展部分。用户不能使用 VBControlExtender 对象的 ObjectEvent 事件来侦听本地事件，这是因为，就好比 IQuickActivate 接口一样，VB 期望能对一个特殊事件设置提供支持。

7.4 定制类对象

当第一个线程在 VB 中的 ActiveX EXE 服务器中发送时，VB 通过在 COM 中调用 CoRegisterClassObject API 函数来注册所有公共可创建的对象。类对象常常作为一个支持

IClassFactory 接口的类工厂来使用。然而，绝对没有任何东西能够阻止我们使用 COM 来注册自己的类对象。惟一需要注意的是：必须在那些 VB 知道已被终止的公共对象之前调用 CoRevokeClassObject（以使得服务器关闭）。

这里，从对 CoGetClassObject 的经验出发，应该记住 CoRegisterClassObject 的前三个参数：rclsid 是一个 CLSID，pUnk 是对对象的一个引用并被 CoGetClassObject 返回，dwClsContent 指定了对象的可见范围。CLSCTX_INPROC_SERVER 指定了该对象仅对当前线程中的其他对象可见，CLSCTX_LOCAL_SERVER 则意味着其他线程也能够取得该对象。第四个参数为一个标识，从 REGCLS 列中取出一个值。一般来说，我们将该值指定为 REGCLS_MULTIPLEUSE。这与 VB 类实例属性中的 MultiUse 值相一致，并意味着该类对象一直到被撤销之前都保持为可见。

如果我们正确的注册了一个类对象，在注册范围内便可以从任何代码中取得该对象。一个类对象是一个独立的对象（singleton），这意味着我们可以在多段代码中取得该对象的同一实例以及对它的调用。如果我们已经采用 ATL 编程的话，便知道可以经常用类工厂中的 CreateInstance 方法来返回同一个对象，从而创建一个独立的对象（singleton）。然而，这个附加的 CreateInstance 调用并非必需的，因为所有的注册类对象都是独立的对象（singleton）。

一个注册类对象可以为跨线程和可执行边界的应用提供全局数据。例如，如果想为自己的组件提供上下文信息而又不想将对象传递给每一个调用的话，那么我们便可以注册一个类对象并发布 CLSID。注意：这里，CLSID 并不一定是一个注册了的 CLSID。我们可以使用任何惟一的 GUID，包括该上下文对象的接口标识符。下面的代码将显示注册一个类对象的表格，以及一个使用 CoGetClassObject 来取得类对象的 DLL。

```
'AppConText.class
'Instancing = PublicNotCreatable
Implements IAppContext
Public Property Get IAppContext_AppPath() As String
    IAppContext_AppPath = App.path
End Property
'etc.
```

```
'frmMain.frm
'Constant pulled from the type library for this executable.
Private Const cstrIID_IAppContext As String = _
    "{C33A1761-A296-11D3-BBF5-D41203C10000}"
Private m_dwRegisterCustomCF As DWORD
Private Sub Form _Initialize ()
    m_dwRegisterCustomCF = CoRegisterClassObject ( _
```

```
        IIDFromString(cstrIID_IAppContext), _
        New AppContext, _
        CLSCTX_INPROC_SERVER Or CLSCTX_LOCAL_SERVER,_
        REGCLS_MULTIPLEUSE)
End Sub
Private Sub From_Terminate()
    If m_dwRegisterCustomCF Then
        CoRevokeClassObject m_dwRegisterCustomCF
        m_dwRegisterCustomCF = 0
    End If

End Sub
```

```
'Class in a DLL loaded by main app.
Private Const cstrIID_IAppContext As String = _
    "{C33A1761-A296-11D3-BBF5-D41203C10000}"
Private IID_IAppContext As IID

Private Function Get AppPath() As String
Dim pContext As AppContext
    If IID_IAppContext.Data1 = 0 Then
        IID_IAppContext = IIDFromString( _
            cstrIID_IAppContext)
    End If
    Set pContext= CoGetClassObject( _
        IID_IAppContext,_
        CLSCTX_1NPROC_SERVER Or CLSTX_LOCAL_SERVER, _
        0, _
        IID_IAppContext)
    AppPath = pContext.AppPath
End Sub
```

在 Windows 2000 之前的 Win 32 操作系统中允许用户使用 GetObject("CLSID:C33A1761-A296-11D3-BBF5-D41203C10000") 来取得任何已注册的对象。GetObject 代码并没有 CoGetClassObject 代码直接，这是因为它遍历(go through)了一至两个附加层，却更易于使用。Windows 2000 中的 CLSID 解决方案则显得更为聪明：它实际上是要求 CLSID 在系统中进行注册来解决这一问题。如果我们想要使用 GetObject 语法，则必须在 ActiveX EXE 中使用

一个 MultiUse 类并且对该类进行了正确注册。然而，直接创建该对象违反了 singleton 设计规则。如果不考虑跨线程的支持，我们可以采用 CLSCTX_INPROC_SERVER 标识来对标准 EXE 中的任何类进行注册并忽略所有的注册要求；不过，千万不要试图在 Windows 2000 中采用 GetObject 来获取引用。

有了上面所列出的代码，我们便能够从任何线程或任何已载入到应用程序中的 DLL 中使用注册了的 AppContext 对象。AppContext 中的 PublicNotCreatable 属性以及 CLSCTX_LOCAL_SERVER 标识中的内容使得我们可以安全的取得并排列跨线程边界的对象。如果并不需要跨线程的支持，可以使该类为私有类并使用 CLSCTX_INPROC_SERVER 上下文。第 10 章中的“自定义 ROT 名字”部分将说明如何利用一个自定义名字来注册和取得一个运行对象表中的进程私有对象，从而产生类似于跨组件单元（cross-component singleton）的结果。一般来说，类对象注册是一种更为清楚的方法。

第八章

轻量 COM 对象

COM 对象的生存期可以通过一种引用计数机制来进行控制。这使得我们可以在同一时刻保持对同一对象的多个引用。正如我们在第五章中的“使用基类来共享数据”小节中所看到的一样，VB 中没有提供一种内建机制来对结构进行引用（结构也可被称作“用户自定义类型”），这是因为并不存在一种内在机制来对一个结构所占内存进行引用计数。在本章中，我们将对如何来创造轻量对象进行讨论，这里，我将其作为一种结构来进行定义。当然，同时它也是 COM 对象。

读者可能会问到：既然 COM 对象可以从每个类模块中自动的实体化，为什么还要从一个 VB 结构中创建一个 COM 对象？下面将给出一个关于本章及后续章节中所出现的轻量对象用途的列表，以便由读者自行决定哪些项与自己的应用相关联。

- 对于带有大量小对象的系统而言，轻量（light weight）对象提高了内存的使用率。相比于一个类实例（每一实例占有 96 字节），轻量对象要使用少得多的内存（每一实例占 4-12 字节）。另外，我们也可以用所分配的单个内存块来创建任意数目的轻量对象并释放所有只进行了一次调用的对象。轻量对象使我们能够设计可以处理好几千个对象的系统。
- 轻量对象并不要求单独的堆分配。一个类模块的实例总是在堆中进行分配，不过，我们可以在堆栈中直接分配轻量对象或者将其作为一个类的嵌入对象。需要注意的是，一个在堆栈中分配的轻量对象并不是一个标准的 COM 对象，这是因为一个对象的引用并不保持该对象所占内存为有效。不过，对于一个 COM 对象来说，这已经足以使 VB 能够调用它的方法。
- 我们可以实现任意的接口。而有一些接口，即便在拥有一个 VB 化的类型库的情况下，在类模块中也不能实现。而对于轻量对象则没有这些限制。我们可以使用前面曾描述过的聚合技术来将自己的轻量对象实现与一个类模块连接起来。
- 我们可以忽略接口。轻量对象使得封装一个已存在的接口实现变得容易许多。当我们在封装一个接口时，一般只关心如何去修改一系列函数的功能，而轻量对象

则能让我们只为那些函数写入代码。

- 我们可以给这些结构加入终止代码。轻量对象则允许我们赋予这些结构与终止事件等效的功能。这只需要对这些结构进行初始化并让 VB 中的范围（scoping）规则来完成剩下的工作就可以了。一个结构中的终止代码使我们用到堆栈分配的结构，在这些结构中我们将用到堆栈分配的对象，这样就大大节约了内存的使用并极大的提高了性能。

上面只是部分的说明了轻量对象的用途。和标准结构或类一样，用户可对轻量对象进行扩展：我们可以任意的对它们进行加工。轻量对象提供了极大的灵活性，它可同时作为一个结构和一个 COM 对象来使用。

8.1 关于轻量的基础知识

在结构与 COM 对象之间只有一个微小的差别。简单的说，一个 COM 对象可以被认为是首元素指向一个基于 IUnknown 的 VTable 的结构。为了将一个结构转变为 COM 对象，必须设计一个函数指针数组，并使得该结构中的首元素指向这个函数指针数组的起始点。一旦该结构的首元素指向一个有效的 VTable，一个简单的 VBoost.Assign(或 CopyMemory)调用即可将该结构指向一个 VB 可用的 COM 对象。

VTable 中的函数列表，与描述该 VTable 的接口一起，决定了一个类的行为。因为一个类的每一实例都具有同样的行为，我们自然想让该类的所有实例都共享同一个 VTable。这样，我们就必须在对象第一次创建时填充 VTable 数组。一个 COM 对象在它的 VTable 中至少包含了三个函数：QueryInterface、AddRef 和 Release。对于一个 lightweight 或其他 COM 对象的 VTable 而言，这些函数都是必须的。

对于所有对象来说，无论它们是结构还是类模块，都起始于一个未进行初始化的内存块，然后通过以某种给定类型来填充（populate）该内存块，从而使得该内存块具有某种特征。

- 将其推入堆栈。堆栈内存是一个临时的地方，并仅限于一个函数调用的范围。如果要求的内存较小，那么堆栈内存是具有最小开销的获取内存的一种方法。然而，堆栈是有限的，因此将大量数据推入堆栈可能会导致溢出错误。实际上，VB 具有一个很大的堆栈，一般很难超出其限制。无指针类型局部变量总是在堆栈中进行分配。
- 从堆（heap）内存中进行分配。“堆”对于支持“allocate”函数和“free”函数的所有内存管理对象来说，是一个很普遍的术语。每个堆对象在使用这些函数时采用的名字略微有些不同，但是它们都具有同一目的。当需要一块内存来创建对象时，我们可以调用堆分配函数。处理完毕之后再调用释放函数将内存释放。VB 中的类、变长数组以及变长字符串等等都是采用了堆分配类型。

- 嵌入一个堆或堆栈分配。在实际中，使用嵌入内存与使用堆栈内存非常类似。用户不能直接释放内存，但只要是范围之内的内存就可以使用。对于堆栈分配的情况，其范围由函数决定。而在这里，所嵌入的范围要由所包含的内存块的生命周期所决定。一个类模块中的结构类型成员变量便是嵌入内存的一种形式。

为了最优化的利用内存，应通过使用堆栈内存和嵌入式对象来限制堆分配的数量和大小。如果非要使用堆分配的话，使用两个嵌入式对象来创建一个堆对象总是要比创建两个单独的堆对象花费更少的开销。尽管 RAM 的价钱已经降得很低，但对所有堆内存进行管理仍然极大的降低了其性能。滥用内存很容易降低应用程序的性能。

8.1.1 堆分配的轻量对象

现在让我们作好准备，开始创建一个轻量对象。其中第一个例子并不十分有用，因为它实际上并没有做任何事情，但它却为其他对象提供了一个起始点。下面我们将使用 COM 分配函数 CoTaskMemAlloc 和 CoTaskMemFree(在 VBoostTypes 中进行了定义)作为堆分配器并创造一个传统的进行引用计数的 COM 对象。

```
Private Type LightEmptyVTable
   VTale(2) As Long
End Type
Private m_VTalbe As LightEmptyVTable
Private m_pVTable As Long

Private Type LightEmpty
   pVTable As Long
   cRefs As Long
End Type

Public Function CreateLightEmpty() As IUnknown
Dim struct As LightEmpty
Dim ThisPtr As Long
   'Make sure we have a vtable
   If m_pVTable = 0 Then
      With m_VTable
         .VTable(0) = FuncAddr(AddressOf QueryInterface)
      .VTable(1) = FuncAddr(AddressOf AddRef)
      .VTable(2) = FuncAddr(AddressOf Release)
```

```
            m_pVTable = VarPtr(.VTable(0))
        End With
    End If

    'Allocate the memory.
    ThisPtr = CoTaskMemAlloc(LenB(Struct))
    If ThisPtr = 0 Then Err.Raise 7 'Out of memory

    'Initialize the structure.
    With Struct
        .pVTable = m_pVTable
        .cRefs = 1
    End With

    'Move the bytes in the initialized structure
    'into the allocated memory.
    'Note that the VarPtr(Struct) and ZeroMemory calls are
    'overkill for this structure, but are often required.
    CopyMemory ByVal ThisPtr, Struct.pVTable, LenB(Struct)
    ZeroMemory Struct.pVTable. LenB(Struct)

    'Assign the ThisPtr into the function name variable
    'to officially vest it as a COM object. Use Assign
    'instead of AssignAddRef since the cRefs field is
    'already set to 1.
    VBoost.Assign CreateLightEmpty, ThisPtr
End Function

    Rrivate Function QueryInterface(This As LightEmpty, _
        riid As Long, pvObj As Long) As Long
            'Just fail for now. (I'll discuss QI later in the chapter)
            pvObj = 0
            QueryInterface = E_NOINTERFACE
End Function
Rrivate Function AddRef(This As LightEmpty) As Long
        With This
```

```
        .cRefs = .cRefs + 1
        AddRef = .cRefs
    End With
End Function
Private Function Release(This As LightEmpty) As Long
    With This
        .cRefs = .cRefs - 1
        Release = .cRefs
        If .cRefs =0 Then
            'Add object spelific cleanup here

            'Free the memory. Don't step over CoTaskMemFree
            'in the debugger because the memory associated
            'with the This parameter is being freed.Now is
            'a good time to hit F5 or Ctrl - Shift-F8
            CoTaskMemFree VarPtr(This)
        End If
    End With
End Function

'Helper function
Private Function FuncAddr(ByVal pfn As Long) As Long
    FuncAddr = pfn
End Function
```

在上面的代码中有几个需要注意的方面。第一件读者可能觉得奇怪的事情便是 VTable 是作为一个结构而不是作为定长数组进行声明。正如第二章中所提及的一样，一个定长数组中的数据实际上是在堆中进行分配的，因此必须在拆分期间将其释放。如果用户碰巧拥有一个模块级的且指向 VTable 的轻量对象，并且恰好它又在 VTable 之后拆分的话，将会使得应用程序显得十分的僵硬。如上所示，通过对 VTable 进行声明，便保证了 VTables 比工程中的任何对象都要更长一些。

在上述代码中，另外还需要注意的一件事情便是只有当对象首次被创建或被释放时，我们才能将所分配的内存作为一个生指针（raw pointer）来使用。而在这期间，我们都可以直接看到 LightEmpty 结构。在三个 IUnknown VTable 函数，以及本章中所给出的所有其他 VTable 函数中，其第一个参数的样式为："This As <Struct>"。正如该参数名所指出的一样，该参数是与当前对象相联系的经过类型化后的内存。在 VB 和 C++的类中，this/Me 参数作

为语言的一部分而隐藏。尽管我们看不到这一参数，但实际上它确是总被传递的一个参数。为了得到对象的内存地址（这只在释放内存时才有必要），只需要调用 VarPtr(This)或 VarPtr(This.pVTable)。

在 CreateLightEmpty 函数中，一个 LightEmpty 结构被填充然后被拷贝到一个已分配好的内存位置。整个传递过程看起来十分复杂。我们首先应该注意的只是简单的调用 CopyMemory 来完成拷贝。

```
CopyMemory ByVal ThisPtr, Struct, LenB(Struct)
```

如果要将该段代码直接应用于任何结构，那么还存在两个问题。而对于 LightEmpty 结构来说则没有这些问题，这是因为该结构相对来说还不算太复杂。但是一般说来，对于这些问题还是需要引起注意。如果结构中包含了字符串参数，便会出现第一个问题。如果使用了在 VB 中声明的 CopyMemory API 函数，则任何字符串参数在调用该函数时都经过了 UNICODE/ANSI 转换。我们可以通过用 ByVal VarPtr(Struct)来代替 Struct 以防止字符串的转换。不过对于轻量对象的情况，该结构中的第一个元素为长整型，而这种类型是不用进行转换的。这样，VarPtr(Struct)和 VarPtr(Struct.<FirstMember>)总是等效的。另外，我们还可看到，ByVal VarPtr(Struct)也总是等价于 ByRef Struct.pVTable。当然，如果 CopyMemory 在类型库中声明了的话，例如 VBoostTypes-declared CopyMemory 函数，我们就不会存在这一潜在的问题，但不管怎样，列表中的代码总是安全的。

对于单个的 CopyMemory 调用来说，第二个问题便是：VB 认为当函数超出了范围时需要释放结构中的数据。VB 无法知道结构的所有权已被转移。如果该结构包含了对象、变长字符串或数组类型，一旦对象超出范围，VB 便会友好的自动将其释放。但这时如果我们还想使用该结构的话，就不是我们所希望看到的了。为了防止 VB 在堆栈对象中释放所引用的内存，可以简单的对该结构内存清零。如果我们像 CopyMemory 调用中一样，在对内存进行清零时妥善考虑可能的 UNICODE/ANSI 转换，便可得到列表中给出的传递代码。

另外，还有一个有关内存的问题需要引起注意。在这种情况中，在结构中，我们应在撤销它所占据的内存之前，显式的释放该结构中指针类型成员。CopyMemory 调用将所有权传给堆分配，这样我们就能具有对象范围之内的所有权。这使得我们在调用 CoTaskMemFree 时需要在结构超出范围之前便释放其中对结构的引用。要做到这一点，有两种方法。首先，我们可以使用强制性的内联代码。它将所有字符串赋值为 vbNullString（而不是“ ”），把所有对象设置为 Nothing 并抹除所有的数组。这对于一个复杂的对象而言，是一个存有争议的问题，它潜在有可能导致内存泄漏的问题。而另一种方法便是编写一个帮助子程序以使得 VB 能够清除该结构，并在 Release 函数中使用该过程来替代对 CoTaskMemFree 的调用。

```
Private Sub DeleteThis(This AS AnyStruct)
```

```
Dim tmp AS AnyStruct
Dim pThis As Long
    pThis = VarPtr(This)
    CopyMemory ByVal VarPtr(tmp), ByVal pThis, LenB(This)
    CoTaskMemFree pThis
End Sub
```

在 DeleteThis 过程中，使用了临时结构变量以防 VB 在创建函数中需要重新用到它。DeleteThis 把该结构的所有权传回给 VB 局部变量，并在过程结束时使 VB 释放结构成员。注意：这里若在 CopyMemory 之后调用 ZeroMemory 便显得多余，这是因为该内存将被释放。当然，如果用户喜欢在废弃它之前清空垃圾，则调用 ZeroMemory 将会是一个不错的选择。

8.1.2 堆栈分配的轻量对象

在基于堆的 LightEmpty 实例中，大部分的代码都用来完成内存分配及数据传递的工作。当轻量对象获得堆栈或嵌入内存并用来创建对象时，该代码便大大的被简化。对于非堆分配的轻量对象来说，实际上可以分为两种情况。第一种情况，COM 对象需要在对象被销毁时运行代码。而第二种情况，并不要求有终止代码。无论是哪一种情况，在整个结构超出范围时，创建 COM 对象所占的内存都将被自动的清空并释放。下面让我们来看一下分别在这两种情况下的 LightEmpty 代码。

```
Private Type LinghtEmptyVTable
    VTable(2) As Long
End Type
Private m_VTable As LightEmptyVTable
Private m_pVTable As Long

Public Type LightEmpty
    pVTable As Long
    cRefs As Long
End Type

Public Function InitializeLightEmpty(Struct As LightEmpty)
    As IUnknown
    'Make sure we have a vtable.
```

```
    If m_pVTable = 0 Then
        With m_VTable
            .VTable(0) = FuncAddr(AddressOf QueryInterface)
            .VTable(1) = FuncAddr(AddressOf AddRef)
            .VTable(2) = FuncAddr(AddressOf Release)
            m_pVTable = VarPtr(.VTable(0))
        End With
    End If

    'Initialize the structure.
    With Struct
        .pVTable = m_pVTable
        .cRefs = 1
    End With

    'Assign the pointer to the structure to
    'the function name
    VBoost.Assign InitializeLightEmpty,VarPtr(Struct)
End Function

Private Function QueryInterface(This As LightEmpty, _
    riid As Long,pvObj As Long) As Long
    'Just fail.(I'll discuss QI later in the chapter).
    pvObj = 0
    QueryInterface = E_NOINTERFACE
End Function

Private Function AddRef(This As LightEmpty) As Long
    With This
        .cRefs = .cRefs + 1
        AddRef = .cRefs
    End With
End Function
Private Function Release(This As LightEmpty) As Long
    With This
        .cRefs = .cRefs - 1
```

```
        Release = .cRefs
        If .cRefs = 0 Then
            'Run termination code here.
        End If
    End With
End Function
```

大家看到，代码相当简单。这里去除了所有的内存管理代码。LightEmpty 被声明为公有类型，因此我们可以在模块之外使用它。对象的类型要求 COM 对象的最后释放必须发生在内存超出范围之前，因此，我们有必要确保在堆栈内存范围之外没有使用由堆栈分配的轻量对象。这里如果不要求有终止代码的话，整个程序将会更加简单。并且，对于这种情况，没有必要采用引用计数，因为引用计数仅仅决定终止代码应在何时运行，而不决定何时释放内存。

```
Private Type LightEmptyVTable
    VTable(2) As Long
End Type
Private m_VTable As LightEmptyVTable
Private m_pVTable As Long

Public Type LightEmpty
    pVTable As Long
End Type

Public Function InitializeLightEmpty(Struct As LightEmpty) _
    As IUnknown
    'Make sure we have a vtable.
    If m_pVTable = 0 Then
    With m_VTable
            .VTable(0) = FuncAddr(AddressOf QueryInterface)
            .VTable(1) = FuncAddr(AddressOf AddRefRelease)
            .VTable(2) = .VTable(1)
           m_PVTable  = VarPtr(m_VTable(0))
        End With
    End If
```

```
    'Initialize the structure.
    With Struct
        .pVTable = m_pVTable
        'More stuff in a non-empty example.
    End With

    'Copy the pointer to the structure into the
    'the function name.
    VBoost.Assign InitializeLightEmpty,VarPtr(Struct)
End Function

Private Function QueryInterface(This As LightEmpty, _
  riid As Long, pvObj As Long) As Long
    'Just fail.(I'll discuss QI later in the chapter).
    pvObj = 0
    QueryInterface = E_NOINTERFACE
End Function
Private Function AddRefRelease(This As LightEmpty) As Long
    'Nothing to do.
End Function
```

由于去除了引用计数，也就无需再使用 AddRef 和 Release。实际上，它们都指向空函数。我还发现，使用最多的一种轻量对象类型便是这种类型。

8.2 结构终止代码

在使用轻量对象的第一个具体例子中，我们创建了一个轻量对象来显示和隐藏一个沙漏式光标。并且它是第一次作为一个标准的 VB 类来进行显示的。

```
'HourGlass class
Private m_PrevCursor As Long
Public Sub ShowHourGlass()
    With Screen
        m_PrevCursor = .MousePointer
        If m_PrevCursor <> vbHourGlass Then
            .MousePointer = vbHourClass
```

```
        End If
    End With
End Sub
Private Sub Class_Terminate()
    'Only change back if we set it
    If m_PrevCursor vbHourClass Then
        With Screen
            'Don't touch if someone else changed it.
            If .MousePointer = vbHourGlass Then
                .MousePointer = m_PrevGursor
            End If
        End With
    End If
End Sub

'Class usage
Sub DoLotsOfStuff()
Dim HG As New HourGlass
    HG.ShowHourGlass
End Sub
```

该沙漏光标类作为一个堆栈分配的轻量对象非常适合，这是因为其范围被限制在一个单一的函数之中。考虑到使用代码及运行数据等一些开销，创建一个由堆分配的 VB 类在这里显得十分必要。该类的一个优势在于只需要调用对象中的一个函数即可显示沙漏光标。当该类超出范围时，沙漏光标自动隐藏。

下面将采用一个简单的小技巧来复制单个函数调用和自动终止代码，其类由轻量对象提供。并且使得轻量对象结构拥有在其之上创建的轻量对象。对于这种方法，一旦结构超出了范围，它便自行释放并运行自己的终止代码。由于对一般的轻量对象无需再调用 AddRef 方法，因此只在范围边界上调用 Release 方法就可以了。并且这里没有必要再使用 cRefs 变量。下面便是沙漏光标代码的轻量版本。

```
Public Type HourGlass
    PVTable As Long
    pThisObject As IUnknown
    PrevCursor As Long
End Type
```

```
Private Type HourGlassVTable
    VTable(2) As Long
End Type
Private m_VTable As HourGlassVTable
Private m_pVTable As Long

Public Sub ShowHourGlass(HG As HourGlass)
Dim PrevCursor As Long
    If m_pVTable = 0 Then
        With m_VTable
            .VTable(0) = FuncAddr(AddressOf QueryInterface)
            .VTable(1) = FuncAddr(AddressOf AddRef)
            .VTable(2) = FuncAddr(AddressOf Release)
            m_pVTable = VarPtr(.VTable(0))
        End With
    End If

    With Screen
        PrevCursor = .MousePointer
        If PrevCursor <> vbHourglass Then
            .MousePointer = vbHourglass
            With HG
                .PrevCursor = PrevCursor
                .pVTable = m_pVTable
                VBoost.Assign .pThisObject, VarPtr(.pVTable)
            End With
        End If
    End With
End Sub

Private Function QueryInterface(This As HourGlass, _
    riid As Long,pvObj As Long) As Long
    Debug.Assert False 'QI not expected
    pvObj = 0
    QueryInterface = E_NOINTERFACE
```

```
End Function
Private Function AddRef(This As HourGlass) As Long
    Debug.Assert False 'AddRef not expected
End Function
Private Function Release(This As HourGlass) As Long
    With Screen
        If .MousePointer = vbHourglass Then
            .MousePointer = This.PrevCursor
        End If
    End With
End Function
```

```
'Calling code
Sub DoLotsOfStuff()
Dim HG As HourGlass
    ShowHourGlass HG
End Sub
```

除通过消除创建类模块所需开销而极大的提高了性能之外，对于沙漏光标对象的轻量版本而言，它还具有在没有实际打开沙漏式光标时不运行任何终止代码的优点。我们现在有一种方法来对一个结构运行终止代码，该种方法无需将结构转变成臃肿的 VB 类。当然，这里的确需要我们自己编写一些代码，但大部分都只是拷贝和粘贴，以及加入一个查找—替换来更新该参数的类型。

8.3 LastIID 的轻量版本

作为不要求具有类型库所提供的接口定义的轻量对象的另外一个例子，下面我们将来制作一个 LastIID 类的轻量版本，该类在第 5 章关于聚合的讨论中曾经给出过。在原始的例子中，我们在类模块中使用了一个 QIHook 来监视 QueryInterface 的要求。并将其作为一个局部变量，所以我们实际上创建了两个堆对象（一个是类，另一个是 UnknownHook）来观察一个调用。该类用来支持轻量对象十分理想:该类较小，并且仅作为一个局部范围的对象来使用。正如沙漏光标代码一样，不管是类模块还是其轻量版本，代码的关键部分都是一样的，用户所需做的只是编制一些边角性的代码。

在下面的 LastIID 示例中，LastIID 结构中的 QIThis 成员变量回指向原结构，它等同于沙漏光标示例中的 pThisObject 变量。VBGUID 定义由 VBoost 引用提供。并且还给出了用

来打印一个 GUID 字符串表达式的 GuidString 帮助函数。

```
Public Type LastIID
    pVTable As Long
    QITHis As IUnknown
    IID As VBGUID
End Type


Private Type LastIIDVTable
    VTable(2) As Long
End Type
Private m_VTable As LastIIDVTable
Private m_pVTable As Long

Public Sub InitLastIID(LastIID As LastIID)
    If m_pVTable = 0 Then
        With m_VTable
            .VTable(0) = FuncAddr(AddressOf QueryInterface)
            .VTable(1) = FuncAddr(AddressOf AddRefRelease)
            .VTable(2)= .VTable(1)
            .pVTable = VarPtr(.VTable (0))
        End With
    End If
    With LastIID
        .pVTable = m_pVTable
        CopyMemory .QIThis,VarPtr(.pVTable),4
    End With
End Sub

Private Function QueryInterface(This As LastIID, _
    riid As VBGUID, pvObj As Long) As Long
    'Cache the IID, then fail.
    This.IID = riid
    pvObj = 0
    QueryInterface = E_NOINTERFACE
```

```
End Function
Private Function AddRefRelease(This As LastIID) As Long
    'Nothing to do.
End Function
```

```
'Helper function to get String version of a guid.
Public Function GuidString(pguid As VBGUID) As String
    GuidString = String$(38,0)
    StringFromGUID2 pguid, StrPtr(GuidString)
End Function
```

```
'Snippet to use LastIID to get the IID of Class1.
Dim LastIID As LastIID
Dim Dummy As Class1
    InitLastIID LastIID
    On Error Resume Next
    Set Dummy = LastIID.QIThis
    On Error GoTo 0
    Debug.Print "The IID for Class1 is:" & _
      GuidString(LastIID.IID)
```

在使用 LastIID 轻量版本时，需要牢记的惟一一件事情便是我们无法在不损坏 IID 值的前提下逐步执行整个代码。调试器本身使用 QueryInterface 来决定它能在本地监视窗口中放入多少信息。在读完 IID 之后，可将光标置于某行上然后键入 Ctrl-F8（注意不是 F8 或 Shift-F8）来对该函数进行调试。

8.4 ArrayOwner 的轻量版本

ArrayOwner 轻量对象在模块拆分期间运行终止代码来清除一个引用了定制数组描述符的模块级数组。第二章中的“模块级数组”部分中列出了该类的用法。ArrayOwner 中包含有 pVTable 和 pThisObject 条目，它们用来对拆分和另一个名为 SA 的 SafeArrayId 域设置陷阱。这里我们假定 ArrayOwner 结构被嵌入到另一个结构中，该结构在紧接着 ArrayOwner 之后具有一个某种类型的数组成员（读者可以参看使用样例）。

ArrayOwner 在已进行了初始化但还未拆分时使用了 VBoost。VBoost 在这个时候是不可靠的。这是因为它是一个模块级的变量，并且它实际上可以在 ArrayOwner 代码运行之前

就已被释放。ArrayOwner 有两个 VTable 可选择。第一个 VTable，DestroyData 销毁了数组中的数据。而第二个 vtable、IgnoreData 忽略了 SA.pvData 域。默认值是将其忽略，允许我们在任何 SafeArray 结构中的域中放入无效数据而无不良后果。在 ArrayOwner.Bas 中我们可以发现这一代码。另外，ArrayOwnerIgnoreOnly.Bas 中也包含了这一代码，不过它不支持 fDestroyData 参数。

```
'ArrayOwner Implementation
Private Type ArrayOwnerVTables
        DestroyData(2) As Long
        IgnoreData(2) As Long
End type
Private m_VTables As ArrayOwnerVTables
Private m_pVTableDestroy As Long
Private m_pVTableIgnore As Long

Public Type ArrayOwner
    pVTable As Long
    pThisObject As IUnknown
    SA As SafeArray1d
    'pSA() As <any type> assumed in this position
End Type

Public Sub InitArrayOwner(ArrayOwner As ArrayOwner, _
    ByVal cbElements As Long, ByVal fFeatures As Integer, _
    Optional ByVal fDestroyData As Boolean = False)
    If m_pVTableDestroy = 0 Then
        With m_ VTables
            .DestroyData (0) = FuncAddr (AddressOf QueryInterface)
            .IgnoreData (0) = .DestroyData (0)
            .DestroyData (1) = FuncAddr (AddressOf AddRef)
            .IgnoreData (1) = .DestroyData (1)
            .DestroyData (2) = _
               FuncAddr (AddressOf ReleaseDestroyData)
            .IgnoreData (2) = FuncAdddr (AddressOf
            ReleaseIgnoreData)
            m_pVTableDestroy = VarPtr(.DestroyData(0))
```

```
            m_pVTableIgnore = VarPtr(.IgnoreData(0))
        End With
    End If
    With ArrayOwner.SA
        If .cDims = 0 Then
            .cDims = 1
            .fFeatures = fFeatures
            .cElements = 1
            .cbElements = cbElements
    With ArrayOwner
                If fDestroyData Then
                    .pVTable = m_pVTableDestroy
                Else
                    .pVTable = m_pVTableIgnore
                End If
                VBoost.Assign .pThisObject,VarPtr (.pVTable)
                VBoost. ByVal VBoost.UAdd( _
                    VarPtr (ArrayOwner), LenB(ArrayOwner)), _
                    VarPtr(.SA)
            End With
        End If
    End With
End Sub

Private Function QueryInterface( _
    ByVal This As Long, riid As Long, pvObj As Long) As Long
    Debug.Assert False 'QI not expected
    pvObj = 0
    QueryInterface = E_ NOINTERFACE
End Function
Private Function AddRef(ByVal This As Long) As Long
    Debug.Assert False 'No need to addref.
End Function
Private Function ReleaseDestroyData( _
    This As ArrayOwner) As Long
    With This
```

```
        If .SA.pvData Then
            On Error Resume Next
            SafeArrayDestroyData VarPtr (.SA)
            On Error GoTo 0
        End If
        'Zero the descriptor and array.
        ZeroMemory This.SA, LenB(This) - 4
    End With
End Function
Private Function ReleaseIgnoreData (This As ArrayOwner) As Long
    'Zero the descriptor and array.
    ZeroMemory This.SA, LenB(This) - 4
End Function
Private Function FuncAddr (ByVal pfn As Long) As Long
    FuncAddr = pfn
End Function
```

8.5 接口位于何处

一个 VB 类模块可以提供多个类实现；并且提供一个接口定义。在大部分编程语言中，接口定义和一个 COM 对象接口的实现是分别独立的实体。一个 COM 类给出了在类型库或头文件中定义的一个接口的实现，但是该类并没有对接口进行定义。如果我们能够在 VB 类模块中禁用公共过程，那么可以只允许通过定义了实现的接口来进行外部调用，这样，我们就可以在接口定义和实现都完全没有重复的情况下来十分近似的编制一个 COM 类。

由于轻量对象并不产生一个接口的定义，故接口定义必然来自于其他的地方。我们一般可以将接口分为两组：其中一组需要 VB 代码直接进行调用，另一组则为标准的由 COM 定义的接口，例如 IUnknown(所有的 COM 对象)，IDspatch（为后期绑定所支持）和 IEnumVARIANT(全部支持)。当使用轻量对象来实现标准接口时，我们需要知道 VTable 的设计和 IID 以提供一个指定接口的实现。然而，如果想用轻量对象来替代 VB 中为工程私有的类模块，我们必须能够从 VB 中调用该接口。VB 编译器只知道那些在引用类型库中进行定义的接口以及同一工程中类类型的模块所定义的接口（Class、Form、UserControl 等等）。为了使一个轻量对象可以从 VB 代码中调用，我们必须给 VB 编译器一个接口定义并和接口定义中所指定的函数顺序相匹配，接口定义包含在轻量对象的 VTable 中。

读者可能会认为：定义一个接口最简单的方法便是在 VB 中编写一个类并使它与轻量

对象中的接口相匹配。使用该方法的时候，我们可能会存在一些困难。首先，一个 VB 类主接口中的 VTable 设计并不是明显的。该类的 IID 甚至很不明显。锁定设计和 IID 的惟一方法是将类实例设置为 PublicNotCreatable 或更高级别并使得工程具有二进制兼容性。如果使用轻量对象来进行内部优化（一般用法因为轻量对象的公共性而变得难以使用）的话，为支持一个内部实现细节而将公共信息加入到可执行程序中将是一个十分极端的策略。并且因为 VB 中的类总是源于 IDispatch 的，故其基 VTable 比我们所见过任何一种简单 IUnknown VTable 都要难以实现得多。而且，如果对象仅作内部使用，我们除了采用 VTAble 绑定外没有理由做其他选择，因此一个 IDispatch 实现将是我们需要携带但又不能打开的一件“行李”。

使用 VB 类来为一个轻量对象定义接口会导致问题的产生，因此这里建议使用一个类型库来为轻量对象定义接口。第 15 章中已对有关使用 VB 创建类型库的问题进行了讨论，因此这里不再赘述。然而，在能够创建一个定制轻量对象之前，我们还有一个方面的问题需要进行讨论。

8.6 错误的产生及避免

在一个 VB 类模块中，所有的公有显式函数总是返回一个 HRESULT。它是一个 32 位的长整型值，用以指明标准 COM 错误信息。这就导致了一个问题的产生：一个 VB 类模块中的所有公有函数都返回一个 HRESULT，那么 VB 函数是怎样返回这些值的呢？从 VB VTable 函数中返回的值总是以输出值的形式而不是在返回值中给出。但实际上返回值作为错误信息而保留并且只能通过 Err 对象进行管理。

一个 VB 类中的简单函数看起来应如下所示。

```
Public Property Get Value() As Long
```

但它实际上对类外部而言如下。

```
HRESULT get_Value(long* retVal);
```

当把这个函数放入一个轻量对象的 VTable 中，结果如下：

```
Private Function get_Value (This As SomeStruct,_
   retVal As Long )As Long
```

使用一个返回 HRESULT 的 VTable 条目涉及到三方面的开销。首先，调用函数必须将一个附加变量压入堆栈以便为返回值参数提供变量地址。其次，调用代码必须处理两个而不是一个输出值。第三，调用代码必须时刻为返回错误作好准备。在整个机制中，有一件让人感到沮丧的事情便是大量函数实际上从不会产生错误，因此为此所作的所有努力可能都是徒劳的。

尽管 VB 在所有类中所遵循的 OLE 自动化标准禁止返回任何除 HRESULT 之外的类型（包括 void，VB 中没有应用它），VB 几乎能够调用所有兼容非自动化的接口。特别的，VB 可以调用所有不返回 HRESULT 的函数以及在返回值中直接返回数据的函数。当我们为一个轻量结构定义其接口时，这一内建的支持可以提供一些选择。下面让我们来看看公共暴露（publicly expose）该轻量结构的 Value 成员的两种方法。

```
Private Type ValueStruct
    pVTable As Long
    cRefs As Long
    Value As Long
End Type

// ODL snippet for first possibility.
Interface IValue : IUnknown
{
    [propget]
    HRESULT Value ([out,retval] long* retVal);
    [propput]
    HRESULT Value ([in] long RHS);
}

'VB functions for first possibility.
Private Function get_Value( _
    This As ValueStruct, retVal As Long) As Long
    retVal = This.Value
End Function
Private Function put_Value( _
    This As ValueStruct, ByVal RHS As Long) As Long
This.Value =RHS
End Function

//ODL snippet for second possibility.
interface IValue : IUnknown
{
    long Value ();
    void SetValue ([in] long NewValue);
```

```
}

'VB functions for second possibility.
Private Function Value (This As ValueStruct) As Long
    Value = This.Value
End Function
Private Sub SetValue (This As ValueStruct, _
    ByVal NewValue As Long)
    This.Value = NewValue
End Sub
```

很明显，很容易编制代码来实现第二种机制。如果没有使进入值变得有效，便没有理由增加 HRESULT 的开销。不幸的是，该值并不真正是轻量对象的一个属性，因此我们必须作为一种方法而非一种属性来调用 SetValue。Propget/propput 属性要求具有 HRESULT 返回代码的函数。忽略掉 HRESULT 具有多方面的优势，在这里，以损失内建的属性支持作为代价是值得的。

- 其代码运行起来要显得稍微快一些。测试表明将有 8%~15%的性能改善，具体情况要依返回类型而定。这听起来似乎改善了许多，但实际上对于其中的任何一种调用方法，都需要在几十分之一秒内进行几百万次的调用。尽管这种改善没有任何不良影响，但还有一些其他的方面需要考虑以决定是否有必要调整应用程序的性能。
- 每次调用产生的代码都十分的小，而函数本身的代码则更小一些，但用以调用函数的代码接近无 HRESULT 函数代码大小的三分之一。大部分附加代码只在产生错误的情况下运行。附加代码虽然不会产生很大的性能上的损失，但的确使可执行程序变得臃肿。而实际上，如果能够保持可执行程序尽量的小，那么应用程序将在启动时很少出现页异常（page exception）并且在低端内存中运行的更好。

如果使用轻量对象，我们可以对对象进行完全的控制。通知 VB 等待给定函数可能返回的错误，这样我们便可以充分利用这种可以控制对象的优势。其代码将更为有效，并且也很容易编写。

8.7 从轻量对象返回错误

COM 方法通过返回 HRESULT 错误代码来指定错误值。VB 函数使用结构化的异常处理来跨函数的返回错误。COM 中指明这些方法必须不能产生跨越一个 VTable 边界的异常。废弃这些跨越函数边界的例外将导致 VB 产生错误。在这两种错误处理方法中，轻量对象

都处于一个危险的境地。

可以很容易的将 VB 中函数分成两大类：一类位于标准模块中，另一类位于类模块中。标准模块中（除主程序过程之外）的所有函数都假定在其之上另有一个 VB 函数来捕获它们可能丢弃的错误。毕竟，一个 VB 进程中惟一有效的入口点便是主过程和公共 COM 对象的 VTable 项。所有 VB 中的类模块函数都被装备好以捕获错误，因此标准模块中的程序都假定不会处于丢弃一个不能被捕获的错误之中。这也便是借用（hacking）一个 DLL 来暴露（expose）直接入口十分危险的一个主要原因：如果一个错误是由我们自己所导致，可以很容易的卸除所调用的 EXE(第二个原因是线程的初始化，我们将在第 13 章中对其进行讨论)。

一个类模块中的公有函数并不能假定其调用者能够捕获一个丢弃的异常。由于这一原因，类模块函数实际上并不捕获自己的错误而是在函数完成之前将其转换成 HRESULT 返回值。不管其错误状态如何，类模块函数都同样的返回对其调用者的控制。对于 VB 来说，处理异常要比处理 HRESULTs 处理得更好，因此由一个 COM 对象方法所返回的任何错误 HRESULT，都立即被转换成一个异常并被丢弃。

一个轻量对象中的 VTable 函数实际上都是 COM 方法，所以决不应该让函数中存有漏掉的未捕捉的错误。如果轻量对象被聚合为公共可见对象的一部分，我们应该虔诚的遵守这一规则。如果在对象仅从内部代码中使用的情况下打破这一规则，实际上并不会有什么不良的影响，因为 VB 随时准备捕获所丢弃的任何错误。实际上，这是用 VB 自己的规则戏弄了一下 VB，因为不管怎样，VB 总是会丢弃一个源于 HRESULT 返回值的一个异常。

我们知道，直接产生一个没有设置捕获陷阱的错误是一件很容易的事情，但若要从该函数中返回一个正确的 HRESULT 却并不是很直接。等到捕获到一个错误时，VB 已将一些标准错误映射到 VB 中的错误号上，并去除掉所有错误号中最上层的 16 字节，从而将其转化成用户所习惯的友好代码。这样，其信息也就掺了杂。所有的 VB 错误号其范围都在 &H800A0001 至&H800AFFFF 之间（该范围常被称为 FACILITY_CONTROL 范围），但是 VB Err 对象所显示的范围为&H00000001 至&H0000FFFF。

VB 的错误号由于其长度被压缩而显得非常便于处理。然而，COM 世界中的其他部分，包括作为调用者身份的 VB，都并不考虑这些错误号，这是因为没有设置 SEVERITY_ERROR 位（&H80000000）。为了让任何调用者能够确信它的确是一个错误值，需要在从某个轻量方法返回它之前先将其高位复原。下面的函数尽管不包含一个高级逆映射来执行诸如将错误 7（内存溢出）映射到系统标准&H8007000E(E_ONTOMEMORY)之类的转换，但在绝大多数情况下是很有用处的。除了没有很好的进行定义之外（VB 将多种系统错误都映射到了单个 VB 错误上），整个逆向错误映射显得很庞大并且没有很好的文档进行说明。这对于大部分应用来说并没有影响，但是我们如果需要对特定的错误值进行特殊处理的话，可以很容易的将一个 Select Case 语句加入到所给出的 MapError 函数中。

```
Public Function MapError() As Long
```

```
Dim Number As Long
    Number = Err.Number
    If Number Then
        If Number > 0 Then Number = &H800A0000 or Number
        MapError = Number
    End If
End Function
```

```
'Calling code.
Private Function MyVTableEntry() As Long
    On Error GoTo Error
    'Perform processing here.
    Exit Function
Error:
    MyVTableEntry = MapError
End Function
```

我们必须设置错误陷阱，并且对那些可能丢弃一个错误的公有的可见 VTable 函数调用 MapError，但要是能够保证某个错误不可能被丢弃的话，便无需进行捕获。如果调用了 COM 方法，我们实际上可以采用返回一个常整型值（取代 HRESULT）的方法来重新定义 COM 接口的一个版本。如果函数定义中没有 HRESULT，VB 将不会认为返回值为一个错误。第 10 章中所给出的 ROTHook 示例中，所有的 COM 调用都采用这种机制来进行错误捕获。最初，ROTHook 代码被设计为一个 DLL 对象，它将继续依赖于 HRESULT 而并不是产生异常，这使得该代码很容易移植到一个帮助 DLL 中。

废除 HRESULT 存在几方面的缺点。首先，它导致我们必须创建类型库。我们不能将具有相同 IID 的两个接口放入同一个库中，但可以将对同一接口的多个定义分别放入到不同的引用类型库中。多个定义使得我们既可拥有一个可在轻量对象代码内部中使用的接口版本，又可同时拥有一个从标准代码中进行调用的接口版本。一旦我们对类型进行了定义，就必须编写附加的代码来对错误返回代码进行检查。但同时，我们也丧失了返回值及属性语句所带来的好处。

8.7.1 详细错误信息

在提供错误信息方面，HRESULT 值仅仅是其中的一个部分。虽然错误号易于为调用对象的代码所理解。但一个生疏的错误号对于最终用户来说却显得用处不大。用户更希望得到的当然是文本信息，因此 COM 允许我们返回一系列详细的错误信息并把这些错误信息

显示给用户。我们可以设置适当的错误描述、错误原因、帮助文件以及帮助上下文，从而帮助用户理解这些错误信息。

这里需要注意的是：调用代码在程序设计上不应依赖于文本信息中的字符串；这些字符串的任务只是将错误信息显示给用户。如果程序依赖于字符串信息的话，将很容易导致对本地的依赖性，有时还可能产生更严重的问题。如果将错误信息继续传递到另一个调用者，我们可以对错误号进行修改以使其包含在由程序返回的错误范围之内，但我们不应整个的替代细节域。对错误源进行说明是向用户提供高质量反馈的最佳途径。我们可以在细节描述的开始处加入文本以提供一个错误序列，但在任何时候都应该注意保持原始信息的完整性。

我们通过使用 Err 对象的属性来设置 VB 中详细的错误信息。Err 对象使得我们在调用 Err.Raise 之前设置详细的错误属性。我们也可使用 Raise 方法自身的任选参数。如果把由错误源返回到调用代码的整个过程看作是一段旅程的话，那么填充（Populate）Err 对象仅仅是整个旅程的开始。COM 现在必须与对象进行交互以确保可以得到详细的文本信息并且所得到的信息为当前信息。

Error 交互将依赖于 IErrorInfo 接口以及 SetErrorInfo 和 GetErrorInfo API 函数调用。IErrorInfo 接口中所具有的方法看起来就好像 Err 对象的属性一样；例如 Err.Description 被映射到 IErrorInfo.GetDescription 等等。SetErrorInfo 设置了线程中当前 IErrorInfo 对一个指定 IErrorInfo 的引用，并且由 GetErrorInfo 取得该引用。GetErrorInfo 则清除了线程中当前的引用，因此我们对于每个 SetErrorInfo 只调用一次。

并不是每个调用者都关心错误信息，因此线程中的详细错误信息与当前正在运行的代码无关也是完全可能的。第二个支持接口，即 ISupportErrorInfo，被用来验证错误信息是否为当前信息。在 ISupportErrorInfo 接口中有一方法即 InterfaceSupportsErrorInfo，该方法采用了一个 IID 参数，该 IID 参数将依据刚返回失败 HRESULT 错误代码的方法所拥有的接口而定。

8.7.2 进行错误交互的步骤

（1）由导致产生错误的对象创建一个 IErrorInfo 实现并调用 SetErrorInfo。

（2）某种方法返回一个失败的 HRESULT。

（3）调用者查看该失败的 HRESULT 并质询对象 ISupportErrorInfo 接口。

（4）如果 ISupportErrorInfo.InterfaceSupportsErrorInfo 成功返回，调用者使用 GetErrorInfo 来取得错误信息。

对于 ISupportErrorInfo 的一个实现来说，可能只是简单的返回 NOERROR 而并不对 IID 参数进行检查。如果用户不想为跟踪当前的 IID 参数所困扰，那么就必须确保在返回一个错误之前已经对错误信息进行了正确的设置。如果得不到详细的错误信息，则可以通过传

递一个对 SetErrorInfo API 函数的 Nothing 引用来清除线程中当前存在的错误。

本书提供了 IErrorInfo 和 ISupportErrorInfo 的一个简单实现，以帮助我们为轻量对象实现加入对详细的错误信息的支持。RichError.Bas 提供了一个常数及四个公有函数来直接集成 Err 对象和详细的错误信息。MapError 如上所示。MapErrorKeepRich 从 Err 对象中植入详细的错误信息并调用 SetErrorInfo，ClearErrorInfo 调用 SetErrorInfo 来清除旧的错误信息。这里，CheckErrorInfo 是 QueryInterface 实现的一个帮助函数。

```
'QueryInterface implementation using RichError.Bas.
Private Function QueryInterface( _
    This As LightStruct, riid As VBGUID,pvObj As Long) As Long
    'Check your own IIDs here, Exit Function on success.
    If riid.Data1 = ISupportErrorInfo_Data1 Then
        If Not CheckErrorInfo( _
            QueryInterface,riid,pvObj) Then
            PvObj = 0
            QueryInterface = E_NOINTERFACE
        End If
    End If
End Function
Private Function MyVTableEntry() As Long
    On Error GoTo Error
    'Normal VB code goes here.
    Exit Function
Error:
    MyVTableEntry = MapErrorKeepRich
End Function
```

所有错误对象的实现都对同样两个对象进行了再利用。这种实现依赖于错误对象都遵循的标准化实践:IErrorInfo 引用在取得详细的错误信息之后立即被释放。并且对于每个错误，一个成员函数只被调用一次。只有在 IErrorInfo 对象当前正被引用时，错误信息才被支持。

```
'Selections from RichError.Bas
Private Declare Function SetErrorInfo Lib "oleaut32.dll" _
    (ByVal dwReserved As Long, perrinfo As Long) As Long
Private Type RichErrorVTables
    SEI (3) As Long 'ISupportErrorInfo
```

```
    EI (7) As Long 'IErrorInfo
End Type
Private m_VTables As RichErrorVTables

'Structure and singleton for IErrorInfo implementation.
Private Type ErrorInfo
    pVTable As Long
    cRefs As Long
    Source As String
    Description As String
    HelpFile As String
    HelpContext As Long
End Type
Private m_ErrorInfo As ErrorInfo

'VTable and singleton for ISupportErrorInfo.
'The VTable is the only required element, so there
'is no reason to bother with a structure.
Private m_pSEIVTable As Long
Private m_pSupportErrorInfo As Long

'Provides a quick check to determine if a call
'to CheckErrorInfo is indicated.
Public Const ISupportErrorInfo_Datal As Long = &HDF0B3D60
Public Function MapErrorKeepRich () As Long
Dim Number As Long
    With Err
        Number = .Number
        If Number Then
            If Number > 0 Then _
                Number = &H800A0000 Or Number
            If m_pSupportErrorInfo = 0 Then Init
            m_ErrorInfo.Surce = .Source
            m_ErrorInfo.Description = .Description
            m_ErrorInfo.HelpFile = .HelpFile
            m_ErrorInfo.HelpContext =  .HelpContext
```

```
            .Clear
            SetErrorInfo 0, m_ErrorInfo.pVTable
            MapErrorKeepRich = Number
        End If
    End With
End Function
Public Sub ClearErrorInfo ()
    If m_ErrorInfo.cRefs Then SetErrorInfo 0, ByVal 0&
End Sub
Public Function CheckErrorInfo (hr As Long, _
    riid As VBoostTypes.VBGUID, pvObj As Long) As Boolean
    If m_ErrorInfo.cRefs =0 Then Exit Function
    If IsEqualGUID(riid, IID_ISupportErrorInfo) Then
        pvObj = m_pSupportErrorInfo
        hr = 0
        CheckErrorInfo = True
    End If
End Function
Private Function InterfaceSupportsErrorInfo( _
    This As Long,riid As VBoostTypes.VBGUID) As Long
    If m_ErrorInfo.cRefs = 0 Then _
        InterfaceSupportsErrorInfo = E_FAIL
End Function
Private Function IEIGetSource( _
    This As ErrorInfo,pBstrSource As Long) As Long
    pBstrSource = 0
    VBoost.AssignSwap pBstrSource, This.Source
End Function
```

这里所列出的绝大部分轻量对象代码都变化多端。另外，只有 IEIGetSource 的实现中包含了特别的处理。pBstrSource 实际上是一个字符串，但被声明为一个长整性值，并且被初始化为零值。我们应该警惕所有字符串、对象以及那些仅在定义接口中标志为 out 的数组参数。并不能保证一个 out-only 参数在传递给一个函数之前已用有效数据进行了初始化。VBoost.AssignZero 是选择用来对一个指针类型的参数执行这一操作的一个函数。轻量对象应该在使用 out_only 参数之前对其明确的赋予零值。

由于被 out 参数所引用的内存没有进行初始化，所以，如果 pBstrSource 被声明为字符

串类型并采用标准字符串分配为其分配一个新值的话，VB 可能会释放无效数据并导致崩溃。不过，在该实现中，对 pBstrSource 的类型作了一点手脚，这样就能够使用一个简单的数值分配来清除它了，然后在程序中使用 VBoost.AssignSwap 来窃取当前的源值。实际情况中，GetSource 只被调用一次，所以没有理由越过 GetSource 调用而死抱 m_ErrorInfo.Source 不放。

8.7.3 不支持详细错误信息的情况

尽管一般情况下，如果一个轻量对象不支持详细的错误信息，我们也不用担心。不过有一种情况例外，如果我们从一个 VTable 调用返回一个失败的 HRESULT 并且不支持 ISupportErrorInfo 接口的话，VB 将保持对该对象的引用直至下一次产生错误。我认为这是 VB6 中存在的一个缺陷。在所有模块所占用的内存都被释放之后，处于悬挂状态的引用也将在拆分期间被释放。这时，由于失去了所有模块所占内存和所有的轻量 VTable 以及对轻量对象的引用，将会导致产生崩溃。

解决这一问题最简便的方法是在工程的整个生存期内将其忽略并在模块拆分期间产生一个已设置了错误陷阱的错误。在本书中，包含了 ResetError.Bas 来保证进程的安全。ResetError 是一个在 Release 入口处包含有终止代码的简单轻量对象，它与 HourGlass 轻量对象十分相似。Release 函数及调用代码如下所示。如果读者使用轻量对象并发现所创建的可执行程序在拆分期间有时会产生崩溃，便可在程序中包含进下面的代码。

```
'Release function.
Private Function Release(This As ResetError) As Long
    On Error Resume Next
    Err.Raise 5
    On Error GoTo 0
End Function
```

```
'Calling code.
Private m_ResetError As ResetError
Sub Main()
    InitResetError m_ResetError
    'The rest of the program.
End Sub
```

8.8 聚合轻量对象

一个轻量对象只不过是某一指定接口的定制化实现。我们常常是自己来定义这一接口，但是发现外部对象却要求我们实现指定了的并且已进行过定义的接口。然而在一个 VB 类模块中却并不总是能够实现这一接口。如果受到了限制，那么我们可以将难以处理的接口实现为一个轻量对象，并将其并入到对象中，这就需要用到第 5 章中"聚合存在的对象"部分所讨论过的聚合技术。

IObjectSafety 接口便是可以采用轻量对象来实现的接口类型的一个极好范例。我们可以将该实现代码作为其他实现的一个模板。除了会遇到一点类型库方面的小小麻烦之外，我们可以十分接近的实现 VB 中的 IObjectSafety。惟一的缺点是有关 IObjectSafety 的文档中明确要求：如果不支持所要求的接口，该实现将返回一个 E_NOINTERFACE(&H80004002)HRESULT 值。而 VB 总是在调用者看到之前便将错误代码映射到一个标准的类型不匹配错误(&H800A000D)上。这样我们最终将违反这一要求。对于其他的要求将成功代码作为 HRESULT 返回值(如 IEnumVARIANT)的接口，采用标准 VB 代码不能实现。

创建一个定制接口实现的最简单的方法便是在嵌入主对象实例的内存中建立一个轻量对象。当我们嵌入该结构时并不需要担心内存分配和引用计数，这样，我们便可以集中精力来实现我们所关心的接口函数。嵌入式内存消除了在 AddRef 和 Release 中外在的进行引用计数的必要性。同时我们也无需花费太多精力来消除 QueryInterface 函数的代码。设计这种类型的轻量对象是为了将其作为聚合的一部分以供外部使用，所以它已经包含了管理 QueryInterface 调用所需的代码。如果指定 adUseIIDs 作为一个 AggregateData 标志，VBoost 将指向我们指定的 IID。另外，我们还可以通过指定 adFullyResolved 来消除 QueryInterface 要求指定 IID 的可能。

为了确保能够采用与轻量对象实现一致的模式来设置聚合标志，我们应该植入聚合数据以及相应的 IID 项，并作为轻量对象初始化函数的一部分。这将有助于减少包含轻量实现的对象中的代码数量，并且使得代码易于维护。

ObjectSafety 轻量对象实现了简单的 IObjectSafety 接口，该接口用以告诉那些在一安全主机上所创建的控件和对象允许做什么。例如，如果控件从 SetInterfaceSafetyOptions 函数返回了成功代码，以说明它对于一个不能确信的调用者来说也是安全的，那么这时如果要求该控件执行编辑注册表或者其他一些恶性操作的话，将会导致失败。下面首先列出了该轻量对象的实现，接着是一个应用样例。

```
'ObjectSafety lightweight object implementation.
'Contained in ObjectSafety.Bas.
'Requires VBoost.Bas.
```

```
Private Const strIID_IObjectSafety As String = _
    "{CB5BDC81-93C1-11CF-8F20-00805F2CD064}"

Private Type IObjectSafetyVTable
    VTable (4) As Long
End Type

Private m_VTable As IObjectSafetyVTable
Private m_pVTable As Long

Private IID_IObjectSafety As VBGUID
Public Enum SafetyOptions
    INTERFACESAFE_FOR_UNTRUSTED_CALLER = 1
    INTERFACESAFE_FOR_UNTRUSTED_DATA = 2
    INTERFACESAFE_USES_DISPEX = 4
    INTERFACE_USES_SECURITY_MANAGER = 8
End Enum

Public Type ObjectSafety
    pVTable As Long
    CurrentSafety As SafetyOptions
    SupportedSafety As SafetyOptions
    pUnkOuter As Long
End Type

Public Sub InitObjectSafety (ObjSafe As ObjectSafety, _
    AggData As AggregateData,IID _ AS IID, _
        ByVal IIDNumber As Long, _
    ByVal punkControllingIUnknown As IUnknown, _
    ByVal Supported As SafetyOptions)
    If m_pVTable = 0 Then
        'VBoost may not be initialized in a UserControl
        'at design time, so make sure it's alive.
        If VBoost Is Nothing Then InitVBoost
        IID_IObjectSafety = IIDFromString( _
            StrIID_IObjectSafety)
```

```
        With m_VTable
            .VTable(0) = FuncAddr(AddressOf QueryInterface)
            .VTable(1) = FuncAddr(AddressOf AddRefRelease)
            .VTable(2) = .VTable(1)
            .VTable(3) = FuncAddr( _
               AddressOf GetInterfaceSafetyOptions)
            .VTable(4) = FuncAddr( _
               AddressOf SetInterfaceSafetyOpitions)
           m_pVTable = Varptr(.VTable(0))
        End With
    End If

    'Create the lightweight object.
    With ObjSafe
        .pVTable = m_pVTable
        .CurrentSafety = 0
        .SupportedSafety = Supported
        'ObjPtr without the AddRef/Release
       VBoost.Assign .pUnkOuter, punkControllingIUnkown
       IID = IID_IObjectSafety
    End With
 'populate the AggregateData structure.
With AggData
    VBoost.Assign .pObject,VarPtr (ObjSafe)
    .First IID = IIDNumber
    .Flags = adUseIIDs or adFullyResolved
    IID = IID_IObjectSafety
    End With
End Sub

Private Function QueryInterface ( _
   ByVal This As Long, riid As VBGUID, pvObj As Long) As Long
    'This structure is used as an aggregate with both the
    'adUseIIDs and adFullyResolved flags set, which means
    'that it will never actually receive a QueryInterface
    'call.
```

```
    Debug.Assert False
    pvObj = 0
    QueryInterface = E_NOINTERFACE
End Function
Private Function AddRefRelease (ByVal This As Long) As Long
    'No need to AddRef or Release. This is used inside a
    'blind delegator, which holds the necessary reference
    'on the controlling IUnknown, and the memory is owned
    'by the controlling IUnknown.
End Function

Private Function GetInterfaceSafetyOptions( _
    This As ObjectSafety, riid As VBGUID, _
    pdwSupportedOptions As SafetyOptions, _
    pdwEnabledOptions As SafetyOptions) As Long
    With This
        If CheckInterface(.pUnkOuter, riid) Then
            'Set the options.
            pdwSupportedOptions = .SupportedSafety
            pdwEnabledOptions = .CurrentSafety
    Else
            'Failure case.
            GetInterfaceSafetyOptions = E_NOINTERFACE
            pdwSupportedOptions = 0
            pdwEnabledOptions = 0
        End If
    End with
End Function
Private Function SetInterfaceSafetyOptions( _
    This As ObjectSafety,riid As VBGUID, _
    ByVal dwOptionSetMask As SafetyOptions, _
    ByVal dwEnabledOptions As SaftyOptions) As Long
    With This
        If CheckInterface(.pUnkOuter, riid) Then
            If dwOptionSetMask And Not .SupportedSafety Then
                SetInterfaceSafetyOptions = E_FAIL
```

```
            Else
                'Toggle only the specified safety bits.
                .CurrentSafety = _
                    (.CurrentSafety And Not dwEnabledOptions) _
                    Or dwOptionSetMask
            End If
        Else
            SetInterfaceSafetyOptions = E_NOINTERFACE
        End If
    End With
End Function

'Helper fuction to verify if an interface is supported.
Private Function CheckInterface( _
    ByVal pWeakUnk As Long,riid As VBGUID) As Boolean
Dim pUnk As IUnknownUnrestricted
Dim pvObj As Long
    If pWeakUnk = 0 Then Exit Function
    With VBoost
        .Assign pUnk, pWeakUnk
        'Test for the interface.
        If 0 = pUnk.QueryInterface (riid, pvObj) Then
            'Clear the reference.
            .Assign pUnk, pvObj
            pUnk.Release
CheckInterface = True
        End If
        .AssignZero pUnk
    End With
End Function
```

```
'Sample usage of ObjectSafety.
'Place this code in a UserControl to enable IObjectSafety.
'You can also use this lightweight interface implementation
'with normal classes designed for consumption by the Windows
'Scription Host. m_ObjSafe.CurrentSafety contains the current
```

```
'safety setting.
Private m_ObjSafe As ObjectSafety
Private m_Hook As UnknownHook
Private Sub UserControl_Initialize()
Dim AggData(0) As AggregateData
Dim IIDs(0) As IID
    InitObjectSafty m_ObjSafe, AggData(0), IIDs(0), 0, _
       Me, _
       INTERFACESAFE_FOR_UNTRUSTED_CALLER Or _
       INTERFACESAFE_FOR_UNTRUSTED_DATA
    VBoost.AggregateUnknown Me, AggData, IIDs, m_Hook
End Sub
```

8.9 编制 QueryInterface 函数

上面已经尽可能的忽略了轻量 VTable 中编制 QueryInterface 项的需要。正如 AddRef 和 Release 函数没有做太多的事情一样，在我们目前所看到的轻量对象中，对于 QueryInterface 的要求也是极少的。实际上，在大多数情况下，我们从不需要调用这些函数，因为 VB 编译器在相同类型的变量之间分配引用时并不产生对 QueryInterface 的调用。

QueryInterface 函数提供了一种一般的计算机制。在任何编程语言中，当我们指示编译器将一小片特别的内存视作为一种不同的类型时，便进行一次计算。

在基于接口的 COM 模型中，所有与一给定对象的通信都通过其 VTable 发生，这样对于每一个不同的接口，都要求进行一次计算。QueryInterface 被设计用来提供对所识别出的接口的引用，并且该引用不会返回它所不支持的接口。由于对于 COM 对象来说，QI 是惟一的一种计算方法，那么不返回一个给定接口便锁定了所有不希望进行的计算。从健壮性方面来考虑，这种安全性是一件好事，因为任意的计算将是一件十分危险的事情，并且在许多支持它的语言中容易导致崩溃，例如 C 和 C++语言。

不幸的是，QueryInterface 通常只支持许多 Query，而对许多接口并不支持。对于 COM 和其他组件可能要求的接口，将会有一个不可避免的接口增殖。而对象所希望的只是其中所选定的一小部分。尽可能的对 QueryInterface 实现进行优化是一个好办法。我们需要象察看 OLE 控件接口一样去察看去除了冗余部分的 QI 实例 。IQuickActivate 接口在 1996 年被加入到控件说明中，这是因为我们发现，对于 ActiveX 控件来说，平均有 60%的启动时间都被花费在 QueryInterface 函数上。很显然，其中一部分是结构上的原因。另外僵化的 QI 实现也是其中一方面的原因。

让我们来看看IObjectSafety样例中的QueryInterface函数，并检查一下既支持IUnknown接口也支持IObjectSafety接口的几个实现。当然，这里只是为了方便说明问题，实际上该函数中很少调用QueryInterface。所有实现都假定对IID_IObjectSafety和IID_IUnknown已经进行了定义和初始化。

```
'Implementation 1
Private Function QueryInterface( _
    This As ObjectSafety, riid As VBGUID, pvObj As Long) As Long
    If IsEqualIID (riid, IID_IUnknown) Then
    ElseIf IsEqualIID(riid,IId_IObjectSafety) then
    Else
        pvObj = 0
        QueryInterface = E_NOINTERFACE
        Exit Function
    End If
    pvObj = VarPtr(This)
End Function

'Implementation 2
Private Const Data1_IID_IUnknown As Long = 0&
Private Const Data1_IID_IObjectSafety As Long = &HCB5BDC81
Private Function QueryInterface( _
    This As ObjectSafety, riid As VBGUID, pvObj As Long) As Long
Dim foK As BooL 'IsEqualIID returns a BOOL, not a Boolean.
    If riid.Data1 = Data1_IID IUnknown Then
        foK = IsEqualIID (riid, IID_IUnknown)
    ElseIf riid.Data1 = Data1_IID_IObjectSafety Then
        foK = IsEqualIID (riid,IID_IObjectSafety)
    End If
    If fOK Then
        pvObj = VarPtr(This)
    Else
        pvOj = 0
        QueryInterface = E_NOINTERFACE
    End If
End Function
```

```
'Implementation 3
Private Const Data1_IID_IUnknown As Long = 0&
Private Const Data1_IID_IObjectSafety As Long = &HCB5BDC81
Private Function QueryInterface( _
    This As ObjectSafety, riid As VBGUID, pvObj As Long) As Long
Dim fOK As BOOL
    Select Case riid.Data1
        Case Data1_IID_IUnknown
            fOK = IsEqualIID (riid, IID_IUnknown)
        Case Data1_IID_IObjectSafety
            fOK = IsEqualIID(riid, IID_IObjectSafety)
    End Select
    If fOK Then
        pvObj = VarPtr(This)
    Else
        pvObj = 0
        QueryInterface = E_NOINTERFACE
    End If
End Function
```

尽管这些函数做的都是同样一件事情，但是它们所用的代码却大不相同。其中，第一个实现多次调用了 IsEquallID API 函数;而第二个实现和第三个实现都只调用了一次该函数。在最后两个示例中，其 IID 的前 4 个字节被取出，并定义为常数，我们可以不经过调用函数便对其进行检查。在这些示例中，最后一个示例最为有效，这是因为 Select Case 语句相当于 C 语言中的 switch 语句。这比一系列的 ElseIf 语句要好得多。常数和 Select Case 语句的使用使得所有对 QueryInterface 函数的失败的调用运行起来非常的快。对于失败的情况，无需调用外部函数来对整个 IID 进行检查。

现在，我们已经完成了轻量对象的基本结构，在接下来的 3 章中，我们将会看到这一技术的不同用途。我们将使用轻量对象来创建一个带有许多小对象的系统，并创建一轻量对象来帮助将 VB 放入到运行对象表中，然后再来查看一下允许调用函数指针的轻量对象。

第九章

大型多对象系统

当我们编制复杂算法时，常常需要用到同一类型对象的多个实例。在带有许多交互对象的系统中，我们可能会对使用 VB 类感到灰心丧气，因为它所花费的开销实在太高。这时，我们可能会考虑用一结构数组来代替类模块。为了从一个结构数组中引用另一个结构数组，我们只是简单的将索引存入数组中。所以，如果我们需要查看某项数组元素时，便需要用到数组和索引。这样一个系统与下面的例子相类似。注意，所存储的索引总是要比数组中的实际位置要高一些；如果下一个索引为 0，则表示整个列表结束。

```
Public Type LinkedPoint
    X As Single
    Y As Single
    PrevIndex As Long
    NextIndex As Long
End Type
Public Sub WalkPoints( _
    Points() As LinkedPoint, ByVal StartIndex As Long)
Dim CurIndex As Long
    CurIndex = StartIndex
    Do
        With Points(CurIndex - 1)
            Debug.Print .x, .y
            CurIndex = .NextIndex
        End With
    Loop While CurIndex
End Sub
```

乍一看，该示例还不算太糟。为了得到其中的某项数组元素，数组必须被传递然后被废弃，这实在是一件烦人的事情，但却很有效。不过，如果我们超出了数组的边界，情况将迅速变得难以控制。比如说，我们起初预计需要 100 项，但在算法中间我们突然发现需要用到 101 项。遇到这种情况，第一个反应常常是调用 ReDim 来使得数组更大一些。可是，随着系统的增大，这种调用在性能上的损失有时会让我们觉得难以承受。我们需要用到越来越多的内存并且还可能用到先前已填充好的所有内存。即使是使用 ReDim 语句来一次性的增加多个元素，我们也会发现，这一部分将是整个系统中最慢的一部分。

一种替代的方案是：当一个数组已被填充满时，我们使用 ReDim 来创建第二个数组。尽管从内存的使用角度来看，这是一种有效得多的算法，但是这样的话，整个系统将变得难以索引。VB 中不能引用数组，因此我们必须将数组存储在一个对象中。这些对象中的每一数组都足够小以使得 ReDim Preserve 所花费的开销较小。这样，数组即使大一些也可以接受。在这个更为动态的系统中，原来的 NextIndex 变为两个索引:NextArray 和 NextIndex。NextArray 为 0 表示则没有下一项。索引也由一个长整型值变为一整型值，这是因为在实际应用中常用几个小缓冲器来取代一个大缓冲器，从而避免了产生溢出的可能性。

```
'clsLinkedPoints buffer class
Private m_Buffer(cChunkSize - 1) As LinkedPoint
Friend Property Get Item( _
    ByVal Index As Integer) As Linked Point
Item = m_Buffer(Index)
End Property

'modLinkedPoint
Public Const cChunkSize As Long = 100
Public Type LinkedPoint
    X As Single
    Y As Single
    PrevArray As Integer
    PrevIndex As Integer
    NextArray As Integer
    NextIndex As Integer
End Type

Public Sub WalkPoints(Points() As clsLinkedPoints, _
    ByVal StartArray As Integer, ByVal StartIndex As Long )
Dim CurArray As Integer
```

```
Dim CurIndex As Integer
    CurArray = CurIndex
    CurIndex = StartIndex
    Do
        With Points(CurArray).Item(CurIndex)
            Debug.Print .x, .y
            CurArray = .NextArray
            CurIndex = .NextIndex
        End With
        If CurArray = StartArray Then
            If CurIndex = StartIndex Then Exit Do
        End If
    Loop While CurArray
End Sub
```

上面的代码显得更加麻烦。我们不仅需要传递三项参数来引用正确的结构，而且每次使用结构时都不能对其进行完全拷贝。这对于习惯于对象引用的 VB 程序员而言，将是一件痛苦的事情。实际上，我们非常希望 LinkedPoint 能是一个对象，并且希望 WalkPoints 过程看起来和下面一样:

```
Public Sub WalkPoints (StartingPoint As LinkedPoint)
Dim CurPoint As LinkedPoint
Set CurPoint = StartingPoint
    Do Until CurPoint Is Nothing
        With CurPoint
            Debug.Print .x, .y
            Set CurPoint = .NextPoint
        End With
    Loop
End Sub
```

轻量对象使得我们能够对结构编写一般的基于对象的 VB 代码。通过将 VTable 加入到结构中，我们便可以建立一组动态的数组结构。然而，用于操作对象的代码不必要处理对结构进行定位这一复杂工作。通过把对数组的分配和管理工作转移到一个单独的对象中（我们称之为“内存管理器”）并将结构转变为一个轻量对象，我们就可以创建一个包含许多对象的系统了，该系统允许我们创建对于 VB 和内存来说都十分友好的代码。

9.1 使用定长内存管理器

在目前我们所见到过的轻量对象中，轻量对象所用内存都依次在堆或堆栈中进行了分配。然而，如果我们想得到许多小对象的话，在堆中依次进行分配就显得效率低下，并且将其作为一个结构数组进行分配会导致我们曾经讨论过的所谓"再分配"问题。我们真正需要的是一种采用单个的堆分配来分配多个对象的高效而又多功能的机制。这样，我们便可以在一片被嵌入到更大的内存分配中的内存里建立各个轻量对象。

回顾一下 LightEmpty 示例，我们将看到 CoTaskMemAlloc 函数返回的是一个指针而非某种特定的类型值。为了使用一嵌入式内存指针来代替直接从堆中分配的内存指针，我们需要调用一个不定类型的分配函数来得到一个指向堆分配的嵌入式内存指针。由于返回值类型为一指针，所以内存管理器可以为不定类型。这样，同样的内存管理对象便可以使用于所有类型的轻量对象。

从系统堆对象中分配内存显得很慢，这主要存在三方面的原因。首先，堆对象必须确保所申请的内存是可得到的。如果不是这样的话，堆对象就必须从系统中取得内存。在整个分配过程中，这将是最慢的一步。其次，堆必须支持各种大小不同的分配请求。一个给定分配的大小必须进行存储以使得堆能够知道在调用 Free 函数时包含了多少内存。其大小(可能还有其他信息)通常被存储在一个位于内存指针之前的头中。对于由 Alloc 返回的每个指针都必须具有对应的头，并且它很容易和所请求的对象具有同样大小。当我们大量地进行小分配请求时，这些头便占用了大量的内存。第三，系统堆支持压缩:当我们从堆中释放足够多的内存时，堆将其交回给整个系统。

尽管系统请求不可能从内存分配中消除，然而并不存在一种又快又牢靠的且内存管理对象必然支持可变长度分配和压缩的规则。实际上，移除对可变长度内存的申请可使我们能够进行一种速度很快的分配。我们很容易编写一种定长分配算法而无须对每个分配都使用头，因此所分配的内存可以更多的应用到对象而不是内存管理上。由于一个指定的轻量对象的每个实例都要求同样数量的内存，所以我们在创建多个轻量对象时，通过使用定长分配的对象便可极大的提高内存使用效率。

在创建一个定长内存管理器的一般技术中，都使用一个系统所提供的堆分配器来分配大块的内存。该内存块被分割成多个更小的内存块，然后再使用 Alloc 函数来分发这些指向子内存块的指针。当一大块的内存都被使用完毕时，内存管理器便返回到系统以获取新的内存块。内存管理器提供了与分配多个数组并对这些数组中的索引进行管理同样的效率。不过，在内存管理器中，所有的细节都被隐藏了:我们只需要调用 Alloc 函数，并且使用单个指针值来取代用以对数组及数组中元素进行跟踪的两个索引值。

VBoost 实现了两个定长内存管理器:FixedSizeMemoryManager 和 CompactibleFixedSizeMemoryManager。使用 VBoost.CreateFixedSizeMemoryManager 函数来取得这些分配器的任何一个实例。CreateFixedSizeMemoryManager 使用 Elementsize 和 ElementsPerBlock 参数以

及一个 fCompactible 标识，该标识用来表示是否需要支持压缩。在本书的源代码中，读者可以看到这些对象的 C++及 VB 实现，但不用过分追究其实现细节。

9.2 Scribble 示 例

Scribble 是一个简单程序，利用该程序，我们可以使用鼠标在窗口中随意的涂画。一个 Scribble 实现的起始部分用 VB 很容易编写。当加入下面的代码时，我们每次移动鼠标并同时按下鼠标左键即可得到一条线，当键被弹起时，线消失。我们应在 Form 中将 ClipControls 设置为 False 以使需要重画的表面区域最小化。

```
Privat Sub Form_MouseDown ( _
    Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    If Button = vbLeftButton Then
        CurrentX = X
        CurrentY = Y
    End If
End Sub
Private Sub Form_MouseMove( _
    Button As Ingteger, Shift As Integer, _
    X As Single, Y As Single)
    If Button = vbLeftButton Then
        Line -(X,Y)
    End If
End Sub
Private Sub Form_MouseUp ( _
    Button As Integer, Shift As Integer, _
    X As Single, Y As Singel)
    If Button = vbLeftButton Then
        PSet (X,Y)
    End If
End Sub
```

Scribble 代码中存在的一个问题是没有对笔画进行保留。当我们覆盖了整个窗口并迫使其重画时，所有的笔画都丢失。当然，如果我们打开 AutoRedraw 便可保留所画的线。但是

从内存使用的角度来考虑，在 Form 中使用 AutoRedraw 显得花费开销很大（并且 AutoRedraw 方案在处理轻量对象方面考虑的还不够充分）。为了能在 Form_Paint 事件中启动重画，我们需要保留一个所有访问点的记录。该记录机制是包含大量小对象的对象系统的一个极好范例。

我们的对象系统中包含两种类型的对象：ILinkedPoint 和 IStartingPoint。注意，在可能削减 AddRef/Release 循环的情况下，这些接口定义都使用了 ByRef 语法（ILinkedPoint**）。另外，系统中没有包含错误返回代码。实际上，对象系统中惟一可能的错误发生在分配期间。一旦成功的对内存进行了分配，便不会有任何的错误产生，因此也就无需在 VTable 函数中使用 HRESULT 返回值。

```
Interface ILinkedPoint : IUnknown
{
    long X ();
    long y ();
    ILinkedPoint* Next ();
    Void SetNext([in] ILinkedPoint** pNext);
}
interface IStartingPoit : IUnknown
{
    IStartingPoint* Next();
    ILinkedPOint* First();
    Void AddPoint([in] ILinkedPoint** pNew);
}
```

我们已经将这些对象与两个创建函数相联系：第一个是 NewPoint，它采用 X 参数和 Y 参数并返回一个 ILinkedPoint 实现；第二个是 NewStartingPoint，它用以取得当前的鼠标位置点和当前的起始点。当前的起始点成为新起始点的 Next，这里允许所拥有的对象仅保持一个单一的 IStartingPoint 引用以保证整个系统处于活的状态。AddPoint 方法被调用来将一个新点加入到已存在的一系列对象中。点跟踪系统在内部实现中仅仅加入了两行代码。MouseDown 调用 NewStartingPoint，并且 MouseMove 调用 AddPoint。由于通过这些点来跟踪调用，Scribble 程序中的 form 代码将变得更长，但仍然十分方便管理。另外，它也是完全基于对象的，尽管该点对象实际上是轻量对象。其下的内存管理器和结构也都完全的被封装起来，因此窗口代码不必要担心它们。并且，我们可能在 Form_DblClick 中也注意到了释放整个点系统是多么的容易。

```
Private m_StartingPOint As IStartingPoint
```

```
Private Sub Form_MouseDown( _
    Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    If Button =vbLeftButton Then
        CurrentX = X
        CurrentY = Y
        Set m_StartingPoint = NewStartingPOint( _
            NewPoint(X,Y), m_StartingPoint)
    End If
End Sub

Private Sub Form_MouseMove( _
    Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    If Button = vbLeftButton Then
        If Not m_StartingPoint Is Nothing Then
            Line -(x,Y)
            m_StartingPoint.AddPoint NewPoint(X,Y)
        End If
    End IF
End Sub

Private Sub Form_MouseUp( _
    Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
    If Button = vbLeftButton Then
        If Not m_StartingPoint Is Nothing Then
            Pset (X,Y)
        End If
    End If
End Sub
Private Sub Form_DblClick()
    Set m_StartingPoint = Nothing
    Refesh
End Sub
```

```
Private Sub Form_Paint()
Dim pCurStart As IStartingPoint
Dim pCur As ILinkedPoint
Dim pCurNext As ILinkedPoint
    Set pCurStart = m_StartingPoint
    Do Until pCurStart Is Nothing
        Set pCur = pCurStart.First
        CurrentX = pCur.X
        CureentY = pCur.Y
        Set pCurNext = pCur.Next
        Do Until pCurNext Is Nothing
            Set pCur = pCurNext
            Line -(pCur.X, pCur.Y)
            Set pCurNext = pCur.Next
        Loop
        PSet (pCur.X, pCur.Y)
        Set pCurStart = pCurStart.Next
    Loop
End Sub
```

BAS 代码实现了两种类型的点对象，该代码使用 CompactibleFixedSizeMemoryManager 来透明的处理内存要求。当 CompactOnFree 被设置为 True 时，只要使用这些对象的代码在所有项都被释放时就无须担心堆压缩的问题。只要内存块中的所有引用在内存管理器释放之前被设置为 Nothing，便可以保证系统的安全。由于 BAS 模块中的模块级变量在所有的可见类实例被释放之后被拆分，所以窗口中拥有的所有点也将在内存管理器拆分之前自动的被释放。下面便是 BAS 模块所包含的代码：除依据不同的内存分配有些变化之外，所有的轻量代码看起来都很熟悉。考虑到简捷性，三个 IUnknown 函数被省略，它们和先前所列出过的进行引用计数的 LightEmpty 函数相同。

```
Private m_MM As CompactibleFixedSizeMemoryManager
Private Type PointVTable
    VTable(6) As Long
End Type
Private m_VTable As PointVTable
Private m_pVTable As Long
```

```
Private Type Start VTable
    VTable (5) As Long
End Type
Private m_StartrTable As StartVTable
Private m_pStartVTable As Long

Private Type LinkedPoint
    pVTable As Long
    x As long
    y As long
    next As ILinkedPoint
    cRefs As Long
End Type

Private Type StartingPoint
    PVTable As Long
    Next As IStartingPoint
    First As ILinkedPoint
    Last As ILinkedPoint
    CRefs As Long
End Type

'The structures are the same size,so we use the same
'memory manager to allocate them.
Private Const cPointMemMgrSize As Long = 20
Private Const cPointsPerBlock As Long = 128

Public Function NewPoint( _
    ByVal X As Long, ByVal Y As Long) As ILinkedPoint
Dim Struct As LinkedPoint
Dim ThisPtr As Long
    If m_pVTable = 0 Then
        If m_MM Is Nothing Then
            Set m_MM = VBoost.CreateFixedSizeMemoryManager( _
                cPointMemMgrSize, cPointsPerBlock, True)
            m_MM.CompactOnFree = True
```

```
        End If
        With m_VTable
            .VTable(0) = FuncAddr(AddressOf QueryInterface)
            .VTable(1) = FuncAddr(AddressOf AddRef)
            .VTable(2) = FuncAddr(AddressOf Release)
            .VTable(3) = FuncAddr(AddressOf GetX)
            .VTable(4) = FuncAddr(AddressOf GetY)
            .VTable(5) = FuncAddr(AddressOf GetNext)
            .VTable(6) = FuncAddr(AddressOf SetNext)
            m_pVTable = VarPtr( .VTable(0))
        End With
    End If
    ThisPtr = m_MM.Alloc
    With Struct
        .pVTable = m_pVTable
        .X = X
        .Y = Y
        .cRefs = 1
        CopyMemory ByVal ThisPtr, .pVTable, LenB(Struct)
        ZeroMemory .pVTable, LenB(Struct)
        VBoost.Assign NewPoint, ThisPtr
    End With
End Function
Public Function NewStartingPoint( _
    FirstPoint As ILinkedPoint, _
    PrevStart As IStartingPoint) As IStartingPoint
Dim Struct As StartingPoint
Dim ThisPtr As Long
    If m_pStartVTable = 0 Then
        If m_MM Is Nothing Then
            Set m_MM = VBoost.CreateFixedSizeMemoryManager( _
                CpointMemMgrSize, cPointsPerBlock,True)
            m_MM.CompactOnFree = True
        End If
        With m_StartVTable
            .VTable(0) = FuncAddr( _
```

```
            AddressOf StartQueryInterface)
        .VTable(1) = FuncAddr(AddressOf StartAddRef)
        .VTable(2) = FuncAddr(AddressOf StartRelease)
        .VTable(3) = FuncAddr(AddressOf StartGetNext)
        .VTable(4) = FuncAddr(AddressOf StartGetFirst)
        .VTable(5) = FuncAddr(AddressOf StartAddPoint)
        m_pStartVTable = VarPtr( .VTable(0))
    End With
  End If
  ThisPtr = m_MM.Alloc
  With Struct
    .pVTable = m_pStartVTable
    Set .Next = PrevStart
    Set .Last = FirstPoint
    Set .First = .Last
    .cRefs = 1
    CopyMemory ByVal ThisPtr, .pVTable, LenB(Struct)
    ZeroMemory .pVTable, LenB(Struct)
    VBoost.Assign NewStartingPoint, ThisPtr
  End With
End Function

'VTable functions for the ILinkedPoint
'IUnknown functions are omitted.

'Called by Release to do the dirty work.
Private Sub DeleteThis(This As LinkedPoint)
Dim Tmp As LinkedPoint
  This = Tmp
  m_MM.Free VarPtr(pThis)
End Sub
Private Function GetX(This As LinkedPoint) As Long
  GetX = This.X
End Function
Private Function GetY(This As LinkedPoint) As Long
  GetY = This .Y
```

```
End Function
Private Function GetNext(This As LinkedPoint) As IlinkedPoint
    Set GetNext = This.Next
End Function
Private Sub SetNext( _
    This As LinkedPoint, pNext As ILinkedPoint)
    Set This.Next = pNext
End Sub
'VTable functions for the IStartingPoint
'IUnknown functions are omitted.

'Called by Release to do the dirty work.
Private Sub StartDeleteThis(This As StartingPoint)
Dim tmp As StartingPoint
    This = Tmp
    m_MM.Free VaPtr(this)
End Sub
Private Function StartGetNext( _
    This As StartingPoint) As IStartingPoint
    Set StartGetNext = This.Next
End Function
Private Function StartGetFirst ( _
    This As StartingPoint) As ILinkedPoint
    Set StartGetFirst = This.First
End Function
Private Sub StartAddPoint( _
    This As StartingPoint, pNew As ILinkedPoint)
    With This
        .Last.SetNext pNew
        Set .Last = pNew
    End With
End Sub
```

现在，我们便拥有了可保留一系列点的一个完整的系统。然而，如果我们这时逐步的执行该段代码，将会发现这些代码并不完美。在创建点时，我们大部分时间都花费在 AddRef 和 Release 函数上。由于这些函数运行起来很快，情况也就显得并不太糟。可是，当我们逐

步执行到拆分代码时，便能够看到：将所有那些小对象释放回分配器需要做多少工作。另外我们也将看到堆栈深度存在冗余。（我们通常可以采用 VBoost 分配器来进行对 Free 的调用；pVTable 是发生变化的惟一成员。只有在我们十分不幸的触发了一个内存压缩时，才可能遇到麻烦。）

当该系统被拆分时，我们可能会奇怪系统实际上原来是如此的简单。通过将前一个函数加入到下一个函数中，我们可以最终得到一个双重链接系统而并非只包含单个链接的系统。这样，就使得我们可以处理循环引用，所以释放掉顶层对象实际上并没有重新填充分配器。当然，我们可以使用在第六章中所描述的循环引用技术来进行传播，但是仍然需要在系统拆分期间访问系统中的所有对象以处理引用计数。否则，对象将不会将其自身释放回分配器。

循环引用对象系统中存在的一个障碍在于拆分它们要比创建它们困难的多。通过对内存管理采取外部控制，我们实际上可以无需受到引用计数的困扰而拆分整个系统。那么整个拆分过程包含了以下三个步骤：

（1） 访问系统中的所有结点以释放对由内存管理器所控制的对象、字符串和数组的引用（可以选择依赖于对象）。

（2） 释放 VB 对系统中对象持有的所有引用。

（3） 释放内存管理本身。

下面，再让我们来对点代码作一些小小的更改。

- NewPoint 和 NewStartingPoint 使用一个 FixedSizeMemoryManager 引用来提供一个分配器。
- 将 cRefs 成员从两个结构中清除，并且 IUnknown 函数不再对引用进行计数。
- 将 LinkedPoint.Next、StartingPoint.Next 和 StartingPoint.First 都赋为长整型值而非对象类型值。入口函数依据返回值作相应的改变并采用长整型值。这一变化作为一种优化手段以使得无需采用 Set 语句来分配这些值。Set 语句将导致产生 AddRef，而这对于一个没有进行引用计数的系统来说毫无意义。StartingPoint.Last 能够保持其 ILinkedPoint 类型，这样，StartAddPoint 函数便可以很容易的调用 Last.SetNext 了。

```
'Code changes for non-refcounted objects.
Private Type LinkedPoint
    pVTable As Long
    X As Long
    Y As Long
    Next As Long 'ILinkedPoint
End Type

Private Type StartingPoint
```

```
    pVTable As Long
    Next As Long 'IStartingPoint
    First As Long 'ILinkedPOint
    Last As ILinkedPoint
End Type

'Make the constant public for external allocators.
Public Const cPointMenMgrSize As Long = 16

Public Function NewPoint( _
    MM As FixedSizeMemoryManager, _
    ByVal X As Long, ByVal Y As Long) As ILinkedPoint
Dim Struct As LinkedPoint
Dim ThisPtr As Long
    If m=pVTable = 0 Then
        'etc.
    End If
    ThisPtr = MM.Alloc
    With Struct
        .pVTable = m_pVTable
        .X = X
        .Y = Y
        CopyMemory ByVal ThisPtr, Struct, LenB (Struct)
        VBoost.Assign NewPoint, ThisPtr
    End With
End Function

Public Function NewStartingPoint( _
    MM As FixedSizeMemoryManager, _
    FirstPoint As ILinkedPoint, _
    PrevStart As IStartingPoint) As IStartingPoint
Dim Struct As StartingPoint
Dim ThisPtr As Long
    If m_pStartVTable = 0 Then
        'etc.
```

```
End If
    ThisPtr = MM.Alloc
    With Struct
        .pVTable = m_pStartVTable
        .Next = ObjPtr(PrevStart)
        Set .Last = FirstPoint
        .First = ObjPtr(.Last)
        CopyMemory ByVal ThisPtr, Struct, LenB(Struct)
        'We don't care about ZeroMemory because a Release
        'doesn't matter.
        VBoost.Assign NewStartingPoint,ThisPtr
    End With
End Function

'Use these IUnknown functions for both types.
Private Function QueryInterface(
    ByVal This As Long, riid As Long, pvObj As Long) As Long
    'This actually never fires because we use only strong
    'types. Assert and fail just in case.
    Debug.Assert False
    pvObj = 0
    QueryInterface = E_NOINTERFACE
End Function
Private Function AddRefRelease (ByVal This As Long) As Long
    'Nothing to do
End Function

'VTable functions for ILinkedPoint.
'GetX, GetY unchanged.
Private Function GetNext (This As LinkedPoint) As Long
    GetNext = This.Next
End Function
Private Sub SetNext (This As LinkedPoint, pNext As Long)
    This.Next = pNext
End Sub
```

```
Private Function StartGetNext( _
    This As StartingPoint) As Long 'IStartingPoint.
    StartGetNext = This.Next
End Function
Private Function StartGetFirst( _
This As StartingPoint) As Long 'ILinkedPoint
    StartGetFirst = This.First
End Function
'StartAddPoint unchanged.
```

作出上面这些改变之后，使用这些点对象的 Form 将负责进行内存管理，并采用适当的顺序拆分系统。现在 Form 需要 Load、UnLoad 代码并对双击事件作出一些改变从而对这些点和屏幕进行更新。Mouse*规则中惟一的变化便是 m_PointMM 被传递到了 NewPoint 和 NewStartingPoint 函数。

```
Private m_PointMM As FixedSizeMemoryManager
Private m_StartingPoint As IStartingPoint
Private Const cPointsPerBlock As Long = 128

Private Sub Form_Dblclick()
    'Release all of VB's references to objects
    'owned by the memory manager.
    Set m_StartingPoint = Nothing

    'Release the memory manager to free all memory
    'The placement of this statement is discussed below
    Set m_PointMM = Nothing

    'Recreate the memory manager.
    Set m_PointMM = VBoost.CreateFixedSizeMemoryManager( _
        cPointMemMgrSize, cPointsPerBlock)
    Refresh
End Sub

Private Sub Form_Load()
    Set m_PointMM = VBoost.CreateFixedSizeMemoryManager( _
        cPointMemMgrSize,cPointsPerBlock)
End Sub
```

```
Private Sub Form_Unload (Cancel As Integer)
    'Release the starting point before m_PointMM
    'because m_PointMM owns the memory for the object.
    Set m_StartingPoint = Nothing
    Set m_PointMM = Nothing
End Sub
```

这里，惟一不同以往的代码是显式的调用了 Set=Nothing 来释放内存管理器。实际上，我很少执行一个显式的 Set=Nothing 语句，这是因为 VB 在函数结束时会对所有对象变量进行检查并释放这些变量，所以附加的代码并没有引起任何的变化。这里显式的释放引用只是因为考虑到了 VB 执行 Set 语句的顺序。VB 首先求出右边表达式的值，并且只有在正确的求出了右边表达式值之后才会释放当前对象变量中对指针的引用。

QI 的顺序及在 Set 语句期间对 Release 的调用产生了这样一种情形：即除非进行了显式的 Set=Nothing 调用，否则在同一时刻，两内存管理器均处于活的状态。如果我们在创建一个新管理器之前显式的释放了旧的引用，那么面临的将是一件绝对美好的事情。在大部分情况下，ObjPtr(m_PointMM)对于旧的函数和新的函数来说都是相同的。不可压缩的内存管理器将被优化，即内存管理器将其第一个块从堆中沿其所拥有的内存进行分配。在拆分过程中，任何额外的缓冲器都将被释放到堆中，然后再释放具有双重功能的 object/first 缓冲器分配。

在上面的代码中，另外一个对堆的要求便是：在创建新对象时；新对象应当恰好与刚刚释放的内存块具有同样大小。除非调用 Free 时导致发生了堆压缩，否则堆都将返回同样的内存块。（本来，我们最后将一个 washed-up player 置于 waivers 之上的目的也只是为了能够在几分钟之后作为一个 star rookie 来取得它）。而其最终的状态也确实是相同的，就好像我们已经使用了可压缩的分配器一样——一个没有分配项的空闲块。惟一的区别在于：没有必要再执行所有由引用计数驱动的拆分代码来返回到零值。瞧瞧，该买卖有多合算！

在该示例中，已经创建了某种在 VB 中被认为是一套 COM 对象的东西。所有的调用代码都通过完全基于对象的 COM 方法和属性调用来与对象进行交互。该示例仅仅使用了所需要的 COM 来将 OOP 结构加入到系统中。这里出现了 IUnknown VTable，但在 AddRef/Release 中没有引用计数，并且在 QueryInterface 中也没有进行接口验证。示例中已经通过使用一个非 OLE 自动化的可压缩接口来减少错误检查，并简单的对 VTable 函数中的参数类型进行了修改，从而清除了许多的 AddRef 和 Release 调用。这样，最终我们就得到一个十分有效的系统，该系统可以极其容易的被拆分，并且每个对象只用到了 16 字节的内存。

第十章

VB 对象和运行对象表

运行对象表(简称 ROT)是个全局系统表，它将一个名字和一当前运行对象联系起来。ROT 是不需要进程定义自己的通信协议并允许对象跨越进程边界进行对象定位的中枢存储器。某一进程在 ROT 中注册了它的一个对象后，也就声明了它允许其他能够识别这一对象名的进程共享此对象。如果其他进程不能识别这个对象名，便不能得到该对象。并且在两个对象采用同一名字注册的情况下，ROT 将无法确定要得到其中的哪一个对象。

VB 与 ROT 的相互作用只有在 GetObject 函数发生变化时才能建立，此时它将自动的建立实例化应用程序对象。GetObject 函数有四种结构模式，其中有两个可与 ROT 进行交互。

（1）GetObject("","Server.Class")与函数 CreateObject("Server.Class")的作用等价。

（2）GetObject(,"Server.Class")用于在 ROT 中查找指定类的某一对象。

（3）GetObject("PickAName")通过一个复杂的名字解析进程对某一对象的名字进行解析。在这个进程中可能包含了运行对象表。（这里暂不对其具体结构作详细讨论，读者可在 MSDN 中通过查找 GetObject 函数来获得更多的相关信息。

（4）GetObject("FileName","Server.Class")用于创建 Server.Class 对象，并利用 IpersestFile 接口来从 FileName 中获得它的内容。

一个应用程序对象可以属于库中定义的可创建的任何一种类型的类，这些类由应用程序对象属性来进行标记。在 VB 中可通过设定类的实例属性为 GlobalMultiUse 或 GlobalSingleUse 来定义一个应用程序对象类。当 VB 要对一应用程序对象进行初始化时，将首先在 ROT 中查找所指定类的对象。只有当 ROT 中没有注册此类的对象，VB 才创建一个新的实例。

一个有效的对象是在 ROT 中能用 GetObject 函数的第二种模式获得的对象。API 提供了三种可用来对从 ROT 中进行添加和删除对象操作的进程进行简化的标准函数。RegisterActiveObject 可注册对象并返回给调用者一个 cookie 值。调用 RevokeActiveObject 从表中删除该 cookie 值。而调用 GetActiveObject 则可从 ROT 中检索此对象。VB 中的 GetObject 函数的第二种结构模式可直接映射到 API 中的 GetActiveObject 函数，它通过

IRunningObjectTable 接口按顺序调用 GetActive 方法。

下面一系列 API 函数调用的功能是将 VB 的 ProgID 请求转换为在运行对象表中的程序调用。

（1）VB.GetObject 调用 CLSIDFlromProgID 以获得一个 CLSID。

（2）VB.GetObject 利用 CLSID 来调用 GetActiveObject 函数。

（3）GetActiveObject 调用 StringFromGUID2 将 CLSID 转换为字符串。

（4）GetActiveObject 调用 CreateItemMoniker 创建一个 IMoniker 对象，它用来代表 CLSID 字符串。（可将此标记看作描述一个可解析字符串的对象）

（5）GetActiveObject 调用 GetRunningObjectTable 可获得 IRunningObjectTable 的参数。

（6）GetActiveObject 通过生成的字符串标记来调用 IRunningObjectTable. GetObject 函数。

在使用上述 API 函数之前，需要记住以下事项。首先，如果已经获得了 CLSID 对应的字符串描述，那么便可以跳过第一步到第三步。其次，字符串的标记完全依赖于 CLSID，所以在运行对象表中的不同进程之间并没有任何区别。如果进程在 ROT 中加入一个对象，然后单独通过 CLSID 来对其进行访问，那么只有在没有其他进程用同一名字注册对象的情况下才能保证获得相同的对象。

ROT 运行进程的不确定性符合 ROT 作为整个系统对象的公告板的宗旨，但这意味着我们不能直接使用 RegisterActiveObject 或 GetActiveObject 在 ROT 中运行特定进程对象。不过，我们可通过创建特定进程标记来确保获得指定的对象。我们只需要创建自己的标记然后就可以直接对 ROT 进行操作了。例如，如果使用 strCLSID&Hex$（App.hInstance）来作为标记的基本字符串，那么我们就可以在当前进程中的任何地方改写这一标记，并能保证获得工程中所运行的对象。

10.1 在 ROT 中注册 VB 对象

在ROT中注册多种用途的VB对象十分的简单。首先需要确定 CLSID。在运行时 CLSID 可通过 CLSIDFromProgID API 函数获得，或者如果能确保内容不发生改变的话，也可以将其作为字符串常量存储。然后用 VBoostTypes 中定义的 CLSIDFromString 函数将字符串转换为 CLSID。如果与二进制兼容，则 CLSID 是静态的。当然，我们也可以在运行时调用本书所附的光盘中 LookupCLSID.Bas 文件的 CLSIDFromFileAndClassName 函数来定义 CLSID。

获得 CLSID 后，读者也许会认为，只需通过再简单的调用 RegisterActiveObject 函数便可完成在 ROT 中注册对象的任务。实际上，在 ROT 中注册某对象要复杂得多。因为 RegisterActiveObject 函数要对对象进行一次或多次访问，所以对于在 ROT 中注册对象这一行为来说，要求该对象不在 ROT 中并且不能在同时引发 Class-Terminate 事件。如果要合理

地终止某一 ActiveX 服务程序，决不能允许外部引用某一对象。运行对象应始终在 ROT 中直到被删除时为止。即使进程结束（没有运行终止代码），在 ROT 中加入的运行对象仍然保留在 ROT 中，一直到我们重新启动计算机为止。ROT 不能对自身进行自动的清理。

要想成为一个优秀的COM用户，我们必须要能够将对象放入到ROT中并且在只有ROT所持有的引用才可激活对象的前提下运行终止代码。我们并不能使用一个常规的 VB 对象来完成这一任务。但是一个高度专业化的轻量对象（被称为 ROTHook）可以正确地在 ROT 中进行添加和删除对象的操作。下面将首先说明我们如何来使用 ROTHook 对象，然后再研究它的具体执行过程。

10.1.1 使用 ROTHook 对象

尽管 ROTHook 对象本身很复杂，但使用起来却十分方便。ROTHook 结构中包含了两个成员：即 Hook 和 Struct。Hook 指向轻量对象在 Struct 中的实现，属于 IROTHook 类型。IROTHook 接口有两种方法：即 ExposeObject 和 HideObject，并具有锁定属性。ExposeObject 的第一个参数是要加入 ROT 中的对象，通常是 Me。第二个参数是对象的 ProgID 值。采用下面的步骤便实现了在 ROT 中注册第一个对象然后删除的操作。

（1）创建一个工程 ActiveX.EXE。

（2）通过添加 VBoost.Bas 和适当的工程引用（参考第一章或附录 A 中的 VBoost 对象）建立一个 VBoost 类型的全局变量。

（3）为 VBoost 加入一个工程索引：ROT Hook 类型，可以在 ROT Hook Types.Olb 文件中找到这个库.

（4）将 ROTHook.Bas 文件加入到工程中。

（5）将下列代码加入到具有 MultiUse 或 GlobalMultiUse 初始化属性的类中。

```
'Class Democlass.
Private m_ROTHook As RoTHook
Private Sub Class_Initialize()
      InitROTHook m_ROTHook
      m_ROTHook.Hook.ExposeObject Me, "ROTDmo.DemoClass"
End Sub
Private Sub Class_Terminate()
      m_ROTHook.Hook.HideObject
End sub
```

从工程的内部或外部创建一个 DemoClass 实例，先将此对象加入 ROT 中，GetOject（，"ROTDemo.DemoClass"）可成功的获得这个类注册的实例。这个对象一直保留在 ROT 中，

直到经过外部CreateObject或GetObject获得的最后一个对象被释放或者显著调用HideObject时才将其从 ROT 中被删除。如果没有从工程内部创建一个对象，服务程序没有 UI，此时对象一般在外部创建和释放。在 Class_Terminate 中调用 HideObject 无需在 no_UI 的情况下删除 ROT 中的运行对象。

10.1.2 利用 ROTHook 进行调试

当我们对一个含有从 IDE 内部进行注册的对象的工程进行调试时，必须弄清楚 ROT 的内涵。ROTHook 对象在 IDE 中可以正常工作，除非有导致一个 IUnknown hook 的调试限制（参考第五章的“IUnknown 钩入”部分 ）。如果我们通过结束语句或结束按钮来终止一个程序的话，就没有运行通常的终止代码，并且可能将垃圾留在 ROT 中。一般来说，整个系统能够工作在相互孤立的运行对象环境下，但直到重新启动时才能将其清除。ROTHook.Bas 中包含了一个调试函数 ClearROT，它可以用来帮助彻底的清除调试对话，通过对所有当前注册的对象调用 HideObject 就可以在即时面板（immediate pane）中调用此函数了。为了实现这一功能，我们必须添加一条件编辑常数 DEBUGROT，并赋给该常数非零值。需要注意的是：我们在运行可执行程序之前应该清除所有的调试代码。

还有一些工具可用来帮助调试 ROT。第一个工具，IROTView.Exe，可以在 VB 的光盘中目录 Tools\OLETools 下找到。使用 IROTView 就可以显示 ROT 中所有对象的名字。当我们在 ROT 中添加或删除对象时可以看到这个列表的变化。另外还有一个工具，ROTClean.Exe，是由微软公开发表的。ROTClean 可以从 ROT 中清除相互独立的条目项，但微软没有提供源代码并且也不允许提高其兼容性——读者可在http://msdn.microsoft.com上找到有关ROTCLEAN 的信息。

10.1.3 在 ROT 中锁定运行对象

如果我们使用了没有提供 UI 的 ActiveX 对象，那么这个对象通常是由外部创建的，ROTHook 在关闭时就会将其从 ROT 中清除掉。但是在许多情况下一个活动的对象将与拥有此对象的进程中的一个可见的 UI 元素相联系。例如，在由多个对象共同使用的中央跟踪系统中，存在着一个状态表。负责向跟踪系统汇报的应用程序首先调用函数 GetObject（，"Tracker.Report"），然后通过返回的对象报告一组结果。在这种情况下，应用程序通过运行对象表来提供对象，但 ROTHook 在每一个报告结束时就会删除这个对象——这是我们所不希望出现的情况。

为了保证在 UI 仍然使用时对象始终包含在 ROT 中，我们需要人为地给对象建立一个外部连接。IROTHook 接口的锁定属性通过调用 CoLockObjectExternal API 函数提供了实现这种连接的可能性，它是专门为在同一进程中的对象加入外部连接而设计的。既然锁定属性必须在被钩住对象的外部进行设定并且由工程内部的 UI 元素设定，我们为被钩住的对象

添加一个友元函数，从而实现对象锁定和解除锁定的操作。我们只需要在以前所提到的ROTHook基本代码中简单的添加以下代码就可以了。

```
Friend Property Let Locked(ByVal RHS As Boolean)
      m_ROTHook.Hook.Locked = RHS
End Property
```

如果UI与ROT中注册的对象相联系，就可以典型地创建对象并在建立UI时将锁定属性设置为真（与Form_Load事件相同）。在UI释放对象的调用之前将锁定状态设为假（与Form_Unload事件相同）。锁定通常用在可视化的应用程序中，因为需要用户操作将ROT中的运行对象删除。如果我们在一个没有提供解除锁定的部分锁定了一个ActiveX服务器，那么此时即使不使用该服务器也能将其有效的锁定在内存中。

10.1.4 自定义ROT标识

最初的ROTHook用法指令规定只有Active EXE才能使用钩子。当我们仅根据其ProgID值来注册对象时，这种限制有效。ProgID成为CLSID，它可以在没有进程信息的情况下进入ROT中。如果我们在ActiveX DLL中注册建立在CLSID基础上的对象，并打算让ActiveX DLL与进程共享对象，最终实际上可能是与同一应用程序的另一个实例共享对象。为了保证只使用进程中的对象，必须用进程特定的名字来注册和获取对象。

如果希望用非标准的名字来注册对象，只需要将名字传递给ExposeObject的ProgID参数并设定第三个参数（fProgIDIsMoniker）为真。ExposeObject就可略过注册查询直接执行ROT的注册代码。这一特点使得我们能够传递CLSID的字符串版本从而避开了非规范的ProgID值。

```
'Register this object with a custom moniker.
Private m_ROTHook As ROTHook
Private Sub Class_Initiallize()
Dim strGuid AS String
      StringFormGUID2 _
        CLSIDFromprogID("ROTDemo.DemoClass"), strGuid
      InitROTHook ROTHook
      ROTHook.Hook.ExposeObject _
          Me,strGuid & Hex$(App.hInstance), True
End Sub
```

当然，将定制对象加入到 ROT 中只完成了整个问题的一半。从 ROT 中删除对象将是一个更为麻烦的问题。我们不能再用 GetObject 函数来获取对象。下面的 GetMonikeredObject 函数包含在 ROTHook.Bas 中，我们只需通过传递注册对象时所用到的字符串来重新获取此对象。

```
'Public function to get an object by a non-progid moniker.
'All API calls are also declared in ROTHook.Bas
'or ROTHookTypes.olb.
Public Function GetMonikeredObject( _
   StrMoniker As String) As IUnknown
Dim hr As Long
Dim pmk As IMonikerStub
Dim pROT As IRunningObjectTableStub
      hr = CreateItemMoniker("!",strMoniker, pmk)
      If hr = 0 Then
          hr=GetRunningObjectTable(0,pROT)
          If hr = 0 Then
              hr = pROT.GetObject(pmk,GetMonikeredObject)
          End If
      End If
      If hr Then Err.Raise hr
End Function
```

10.1.5 ROTHook 的其他用法

本书中没有讲述在 NT 上提供服务的 VB 对象的使用方法，但在 ROTHook 对象内部建立了对服务的支持。IRunningObjectTable.Register 支持 ROTFLAGS_ALLOWANYCLENT 标识，该标识用来通知ROT 允许并非创建它的客户程序获得此对象。IROTHook.ExposeObject 的第四个参数 ServiceSupport，可用来设定无服务支持、要求服务支持、尝试服务支持。在尝试服务支持状态下，试图用 ROTFLAGS_ALLOWANYCLENT 标识注册。如果注册失败，对象往往是在注册时无此标识。而用此标识来进行注册时，也常常因为不能获得 AppID 的附加功能和其他注册部分而导致失败。读者可以查看 MSDN 中的 IRunningObjectTable::Register 部分以获得更多的信息。

许多人曾经问到过，如何才能使得一个 ROTHook 对象对 DCOM 有效的工作。不幸的是，并不存在与用来创建对象的 CoCreateInstanceEx API 函数等价的 GetActiveObject 函数。

CoCreateInstanceEx（对应 VB 中的 CreatObject 函数）中的参数指定了创建对象的计算机。在由系统所提供的 GetRunningObjectTable 或 GetActiveObjectAPI 函数调用中没有外部参数，所以也就无法采用简单的方法来从远程计算机的 ROT 中直接获取对象。但是在支持聚合的情况下，我们可以创建一个远程对象，该远程对象采用 VBoost 来聚合从本地 ROT 中取得的对象。Class_Initialize 处是我们需要添加代码的惟一的地方。

10.2 ROTHook 实现细节

使用 ROTHook 轻度对象的策略十分的简单：使用 CLSID 来调用 RegisterActiveObject，但是需要传递一个用来替代实际对象的辅助对象。在调用 RegisterActiveObject API 函数时，ROTHook 对象接收所有系统请求的引用计数，并在注册结束之后服从 IUnkown 对主对象的请求。邮递注册传输模式（post registration pass-through mode）则使得额外的接口请求能够直接到达主对象。图 10.1 说明了 ROTHook 与主对象之间的引用关系。

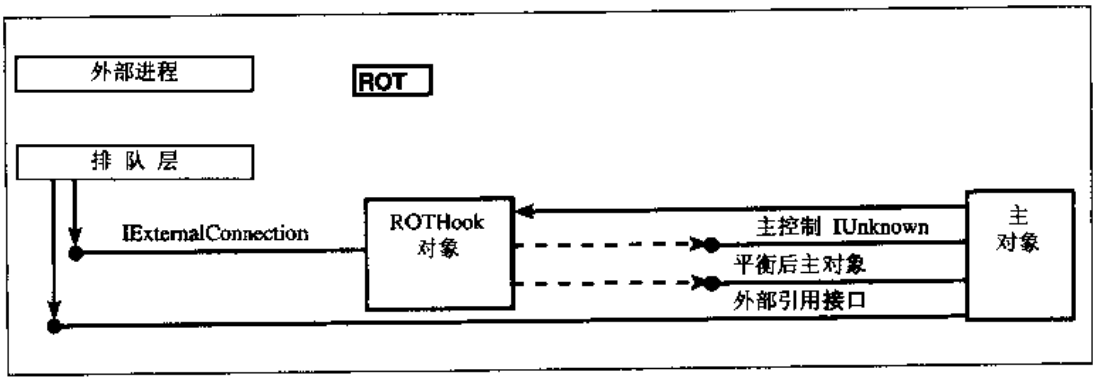

图 10.1 主对象和运行对象表都可持有对 ROTHook 对象的强引用，但只有外部对象才能持有对主对象的强引用。在 ROT 中注册一个 ROTHook 对象不会影响到主对象本身的生存期。在该方框图中，实线代表强引用；虚线则代表弱引用

IExternalConnection 是在 RegisterActiveObject 期间所请求的一个接口。IExternalConnection 非常重要，它允许 COM 服务器区分由内部对象所持有的引用和由外部对象所持有的引用。两种 IExternalConnection 方法，即 AddConnection 和 ReleaseConnection 形成了一种辅助引用计数机制，并用来跟踪外部对象的引用数量。VB 在所有可见的外部对象都被释放时会自动的清理所有内部对象并同时关闭进程，所以，为了避免过早的关闭和泄漏进程，准确的记录外部引用的数量便显得极为重要。

ROT 要求在注册期间引用 IExternalConnection，然后在从 ROT 中检索对象时调用 AddConnection。释放外部变量时调用 ReleaseConnection。通过监视 AddConnection 和 ReleaseConnection 调用，ROTHook 对象便能够确定最后一个外部连接在何时被释放。在没

有连接时，ROTHook 对象撤消 ROT 注册。

10.2.1 轻度对象中的多重 VTables

到目前为止，我们已经讨论了 ROTHook 轻度对象所支持的两种接口即 IROTHook 和 IExternalConnection。到后来我们便会发现，ROTHook 必须也要能够支持 IQIHook，它是第三个 VTable。既然轻度对象是我们第一个所讨论到的需要对多重接口和 VTable 提供支持的对象，因此，下面就让我们来看看轻度对象究竟是怎样做的。

为了支持多重 VTables，我们需要为每一个 VTable 在结构中定义一个数据成员。和以往一样，我们通过在创建程序中返回 VarPtr（.pVTable）来建立主 VTable。但是，在 QueryInterface 实现中，或者可为主接口返回 VarPtr(.pVTable)或者可为辅助接口返回 VarPtr（.pVTableSecondary)。我们需要牢记的是：正传送到辅助程序中 VTable 函数的 This 参数与传递给用来填充主 VTable 序列的 VTable 函数的 This 参数具有不同的类型。

为了对辅助接口函数的 This 对象进行编码，首先我们在结构的顶层放置所有的 VTable 然后拷贝每一个附加 VTable 的主框架。这些拷贝之间，惟一的区别便是每个拷贝都有一个 VTable 项被删除，且每个拷贝都有各自不同的名字。读者可以参考程序清单 10.1 中的 IROTHookImpI、IExtConImpl 和 IQIHookImpI 结构定义，所列出的代码解释了这一过程。

在删除了结构中的成员后，我们惟一遇到的问题便是将不再能引用这些成员。最后列出的接口不能直接查看在它之前的接口。但是，它们并没有丢失。既然所有废弃的成员都是 VTable 指针，所以我们仍然可以通过与该 VTable 相对应的接口来访问辅助对象。我们所需要知道的只不过是距离 VTable 有多远。例如，ROTHook 函数 ReleaseConnection 是 IExternalConnection 接口的一部分，但是它需要运行 IROTHook 接口上 HideObject 中的代码。IExternalConnection 需要依赖于 pVTableExCon 域，该域位于 IROTHookImpI 结构中的 pVTableMain 条目之后，并相隔八个字节。下面的代码用来调用 HideObject。

```
Dim tmpROTHook As IROTHook
        VBoost.Assign tmpROTHook, _
            VBoost.UDif(VarPtr(.pVTableExternal), 8)
        tmpROTHook.HideObject
        VBoost.AssignZero tmpROTHook
```

10.2.2 辅助接口的安全弱引用

在外部对象仍然使用到 VB 中的服务器程序时，为了防止其自行销毁，ROTHook 对象持有一个对本地对象 IExternalConnection 接口的引用，然后对 VB 对象调用 AddConnection

和 ReleaseConnection。但是，ROTHook 并不能保持对属于本地对象的接口的一个强引用，这是因为强引用将禁止触发 Class_Terminate 过程。

到现在，我们一直都是通过一个包含了指向对象的指针的长整型变量来保持弱引用。但是，ROTHook 使用了两种可选的机制。ROTHook 保持对当地对象控制 IUnknown 的弱引用，它位于一个被称作为 pOuter 的 IUnknownUnrestricted 变量中。为了防止额外的释放调用，该变量在 ROTHook 对象废除之前被置为零。第二种弱引用（对本地对象 IExternalConnection 接口的引用）十分的吸引人，值得我们进一步解释。

通常，只要我们能够确保在调用对象时对象为有效，那么保持对该对象的弱引用便是安全的。对本地对象控制 IUnknown 的弱引用通常是安全的，这是因为控制 IUnknown 至少具有对象中其他接口同样长的生存期。不过，如果将其应用到任意对象上，那么保持一个对非控制接口的弱引用将是很危险的事情。没有哪一条 COM 规则会要求对象所支持的所有接口的弱引用都必须在对象本身的生存期内一直有效。

一个 COM 对象可以自由的动态创建一系列伙伴对象以支持不同的接口请求。这些对象被称为 tearoffs，它们可以帮助减少创建对象的大小和初始代价。当对象为响应接口请求而创建 tearoff 时，可将 tearoff 存入到缓冲区中以供以后使用，也可在 tearoff 对象的最后一个引用被释放时简单的将其删除。VBoost 聚合器支持将带有 adDelayCreation 标识的 tearoff 存入到缓冲器中，而不对带有 adDelayDontCacheResolved 标识的 tearoff 做缓存。我们不能只通过保留它的指针就对 tearoff 进行弱引用；如果释放了某一接口的最后一个引用的话，那么即使主对象仍然有效，该接口也已经无效了。

既然当返回接口为 tearoff 时，外部用户并不知道，那么外部用户就应该不会对一个辅助接口进行弱引用。但 tearoff 本身却可以持有对控制 IUnknown 的强引用。我们可以通过释放对控制 IUnknown 的强引用，来平衡对辅助接口所必需的强引用。由于弱引用只在循环引用中使用，那么我们便可确保所释放的控制 IUknown 的引用不是最后一个引用。如果是最后的控制 IUnknow 引用，那么将导致我们现在所讨论的代码不能够运行，因为这时主对象已经被废除了。

当 ROTHook 接收到第一次对 IExternalConnection 接口的请求时，它会向主对象上的同一接口发出请求。ROTHook 保持该强引用，但同时也将调用控制 IUnknown 的 Release 来平衡引用计数，并将 fAddRefOuter 标识设置为 True 以便在对控制 IUnkown 的弱引用为零之前调用 AddRef。ROTHook 返回一个对自身 IExternalConnection 实现的引用，而这将服从于执行其自身的引用计数跟踪之后的封装器实现。

10.2.3 对所有外部请求的计数

关于 ROTHook 体系结构的论述，在 1997 年 8 月份的 Visual Basic Programmer's Journal 期刊中进行了发表。在文章中讨论了如何使用对象而不是如何实现。在 MSDN 的 Visual Studio 6.0 版本中也收录了该篇文章(很狡猾，文章中并没有公布代码)，但是在最初的算法中存在

一个主要漏洞，并且如果没有 VBoost 技术的支持便不能解决。读者通过下列步骤的调用可以看到这一点。

（1）应外部客户端请求，使用 CreateObject 或 New 来创建对象 。

（2）在 Class_Initialize 中，ROTHook 对象在 ROT 中注册。此时 ROT 中包含了一个 ROTHook 对 IExternalConnection 实现的引用。

（3）从新创建的对象中获得 IExternalConnection 对象，并调用 AddConnection。

（4）第二个客户程序调用 GetObject，它在 ROTHook 的 IExternalConnection 接口中调用 AddConnection 函数。

（5）第二个客户程序释放对象的引用，ROTHook 认为这是最后一个外部引用，然后将对象从 ROT 中删除。

尽管对象仍然存在，并且存在对该对象的外部引用，但在 ROT 中已经没有该项了。问题在于 ROTHook 必须监视所有的 AddConnection 和 ReleaseConnection 的调用，而不只是来自运行对象表的调用。解决这个问题的惟一办法便是使用 UnknownHook。一个 UhBefore 钩子将所有的 IExternalConnection 请求重定向到 ROTHook 实现中，这样，ROTHook 的代码就完整了。

程序清单 10.1 ROTHook 的轻度对象

```
'Use private version of CLSIDFromProgID instead of the
'declare in VBoostTypes to return long instead of HRESULT.
Private Declare Function CLSIDFromProgID Lib "ole32.Dll"_
    (ByVal ProgID As Long, rclsid As VBGUID) As Long
Private Declare Function ColockObjectExternal _
    Lib "ole32.Dll" _
    (ByVal pUnk As Long, ByVal fLock As Long, _
     ByVal fLastUnlockReleases As Long) As Long
Private Declare Function RegisterActiveObject _
    Lib "oleaut32.Dll" _
      (ByVal pUnk As Long,rclsid As VBGUID, _
       ByVal dwFlags As Long,pdwRegister As Long) As Long
Private Declare Function RevokeActiveObject
   Lib "oleaut32.Dll" _
   (ByVal pUnk As Long, _
    Optional ByVal pvReserved As Long = 0) As Long
Private Const ACTIVEOBJECT_WEAK As Long = 1
```

```
Private Const MK_S_MONIKERALREADYREGISTERED As Long =&H401E7

Private IID_IUnknown As VBGUID
Private IID_IExternalConnection As VBGUID
Private IID_IROTHook As VBGUID
Private IID_IQIHook As VBGUID

Public Type IROTHookImpl
      pVTableMain As Long
      pVTableQIHook As Long
      pVTableExternal As Long
      pExternal As IExternalConnection
      cRefsExternal As Long
      cConnections As Long
      UnkHook As UnknownHook
      pOuter As IUnknownUnrestricted
      cRefs As Long
      dwRegister As Long
      flocked As Boolean
      fPassThrough As Boolean
      fAddRefOuter As Boolean
End Type
Public Type ROTHook
     Hook As IROTHook
     Struct As IROTHookImpl
End Type

'These are just copies of IROTHookImpl that start
'at different VTable levels.
Private Type IQIHookImpl
     pVTableQIHook As Long
     pVTableExternal As Long
     pExternal As IExternalConnection
     cRefsExternal As Long
     cConnections As Long
     UnkHook As UnknownHook
```

```
    pOuter As IUnknownUnrestricted
    cRefs As Long
    dwRegister As Long
    flocked As Boolean
    fPassThrough As Boolean
    fAddRefOuter As Boolean
End Type
Private Type IExtConImpl
    pVTableExternal As Long
    pExternal As IExternalConnection
    cRefsExternal As Long
    cConnections As Long
    UnkHook As UnknownHook
    pOuter As IUnknownUnrestricted
    cRefs As Long
    dwRegister As Long
    fLocked As Boolean
    fPassThrough As Boolean
    fAddRefOuter As Boolean
End Type

Private Type ROTHookVTables
    Main(6) As Long
    'Don't bother with MapIID entry.It's never called.
    QIHook(3) As Long
    ExtCon (4) As Long
End Type
Private m_VTables As ROTHookVTables
Private m_pVTableMain As Long
Private m_pVTableExtCon As Long
Private m_pVTableQIHook As Long

#If DEBUGROT Then
'Debugging object collection.See ClearROT procedure
'for an explanation
Private m_ObjColl As New Collection
```

```
#End If

Public Sub InitROTHook (ROTHook As ROTHook)
    If m_pVTableMain = 0 Then
        If VBoost Is Nothing Then InitVBoost
'Fill in our interface identifiers
        IID_IUnknown  = IIDFromString(strIID_IUnknown)
        IID_IExternalConnection = _
          IIDFromString(StrIID_IExternalConnection)
        IID_IROTHook = IIDFromString(strIID_IROTHook)
        IID_IQIHook = IIDFromString(strIID_IQIHook)

        With m_VTables
            .Main(0) = FuncAddr(AddressOf QueryInterface)
            .Main(1) = FuncAddr(AddressOf AddRef)
            .Main(2) = FuncAddr(AddressOf Release)
            .Main(3) = FuncAddr(AddressOf ExposeObject)
            .Main(4) = FuncAddr(AddressOf HideObject)
            .Main(5) = FuncAddr(AddressOf get_Locked)
            .Main(6) = FuncAddr(AddressOf put_Locked)
            .ExtCon(0) = FuncAddr( _
              AddressOf QueryInterfaceExtCon)
            .ExtCon(1) = FuncAddr(AddressOf AddRefExtCon)
            .ExtCon(2) = FuncAddr(AddressOf ReleaseExtCon)
        .ExtCon(3) = FuncAddr(AddressOf AddConnection)
        .ExtCon(4) = FuncAddr(AddressOf ReleaseConnection)
        .QIHook(0) = FuncAddr(AddressOf QueryInterfaceQIHook)
        .QIHook(1) = FuncAddr( _
            AddressOf AddRefReleaseQIHook)
            .QIHook(2) = .QIHook(1)
            .QIHook(3) = FuncAddr(AddressOf QIHook)
            'MapIID not used, don't bother
            '.QIHook(4) = FuncAddr(AddressOf MapIID)
            m_pVTableMain = VarPtr(.Main(0))
            m_pVTableExtCon = VarPtr(.ExtCon(0))
            m_pVTableQIHook = VarPtr(.QIHook(0))
```

```
        End With
      End If
    With ROTHook.Struct
          .cRefs = 1
          .pVTableMain = m_pVTableMain
          .pVTableQIHook = m_pVTableQIHook
          .pVTableExternal = m_pVTableExtCon
          VBoost.Assign ROTHook.Hook, VarPtr(.pVTableMain)
     End With
     End Function

'QueryInterface for the main VTable.If we're already
'registered,this just defers to the main object.
Private Function QueryInterface( _
   This As IROTHookImpl, riid As VBGUID, pvObj As Long) As Long
With This
If .fPassThrough Then
    QueryInterface = .pOuter.QueryInterface(riid, pvObj)
Else
    Select Case riid.Datal
          Case 0&
                fOK = IsEqualGUID(riid,IID_IUnknown)
          Case &H995811
                fOK = IsEqualGUID(riid,IID_IROTHook)
          Case &Hl9
                If IsEqualGUID(riid, IID_IExternalConnection) Then
                    If .pExternal Is Nothing Then
                        QueryInterface = _
                           .pOuter.QueryInterface(riid,pvObj)
                        If QueryInterface = 0 Then
                            .pOuter.Release
                            .fAddRefOuter = True
                            VBoost.Assign.pExternal, pvObj
                        End If
                    End If
                    If QueryInterface = 0 Then
```

```
                                .cRefsExternal = .cRefsExternal + 1
                                pvObj = VarPtr(.pVTableExternal)
                                Exit Function
                            End If
                        End If
        End Select
        If fOK Then
            pvObj = VarPtr(.pVTableMain)
            .cRefs = .cRefs + 1
        Else
            pvObj = 0
            QueryInterface = E_NOINTERFACE
    End If
End If
End With
End Function
Private Function AddRef(This As IROTHookImpl) As Long
    With This
        .cRefs = .cRefs +1
        AddRef = .cRefs
    End With
End Function
Private Function Release(This AS IROTHookImpl) As Long
    With This
        .cRefs = .cRcfs - 1
        Release = .cRefs
        If .cRefs = 0 Then
            If .fAddRefOuter Then
                .pOuter.AddRef
                .fAddRefOuter = False
            End If
            VBoost.AssignZero .pOuter
            Set .UnkHook = Nothing
            'Remove this object from our debug only
            'collection.
            #If DEBUGROT Then
```

```
                    On Error Resume Next
                    m_ObjColl.Remove CStr(VarPtr(.pVTableMain))
                    On Error GoTo 0 'Clear the error just in case.
                    #End If
                End If
        End With
End Function
Private Function ExposeObject( _
    This As IROTHookImpl, ByVal pUnk As IUnknown, _
    ByVal ProgID As String, ByVal fProgIDIsMoniker As Boolean,_
    ByVal ServiceSupport As ROTServiceSupport) As Long
Dim CLSID As VBGUID
Dim pmk As IMonikerStub
Dim pROT As IRunningObjectTableStub
Dim pQIHook As IQIHook
       If This.dwRegister Then HideObject This
       If pUnk Is Nothing Then
   Debug.Assert False
       'If this happens,then we're being called incorrectly.
       'Let's return an error instead of GPE'ing
       ExposeObject = &H800A005B 'Error 91
End If
If fProgIDIsMoniker Then
    ExposeObject = CreateItemMoniker("!",ProgID,pmk)
Else
    ExposeObject = CLSIDFromProgID(Strptr(ProgID), CLSID)
    If ExposeObject = 0 And ServiceSupport Then
         'We need a moniker,get one now
         ProgID = String$(38, 0)
         StringFromGUID2 CLSLD,ProgID
         ExposeObject = CreateItemMoniker("!",ProgID,pmk)
         fProgIDIsMoniker = True
     End If
End If
If ExposeObject Then Exit Function
'Register object
```

```
'Assign outer pointer without calling AddRef.
VBoost.Assign This.pOuter, pUnk
If fProgIDIsMoniker Then
      'Use IRunningObjectTable.Register directly.
      ExposeObject = GetRunningObjectTable(0, pROT)
      If ExposeObject = 0 Then
           If ServiceSupport Then
                ExposeObject = _
                    pROT Register(ROTFLAGS_ALLOWANYCLIENT, _
                    VarPtr(This), pmk, This.dwRegister)
                'Put in a failover case
                If ExposeObject And _
                  ServiceSupport = ssAttemptServiceSupport
                     Then
                     ExposeObject = _
                        pROT.Register(ROTFLAGS_DEFAULT,_
                        VarPtr(This),pmk,This.dwRegister)
                End If
           Else
                ExposeObject = _
                    pROT.Register(ROTFLAGS_DEFAULT, _
                    VarPtr(This),pmk,This.dwRegister)
           End If
      End If
Else
      ExposeObject = RegisterActiveObject (VarPtr(This), _
         CLSID,ACTIVEOBJECT_WEAK, This.dwRegister)
End If
 'Check for harmless success code.
If ExposeObject = MK_S_MONIKERALREADYREGISTERED Then _
  ExposeObject = 0
If ExposeObjece = 0 Then
     This.fPassThrough = True
     'Attempt to hook the unknown.
     With VBoost
           .Assign pQIHook, VarPtr (This.pVTableQIHook)
```

```
                On Error Resume Next
                .HookQI pUk, pQIHook, uhBeforeQI, This.UnkHook
                .AssignZero pQIHook
                If Err Then
                    HideObject This
                    ExposeObject = Err
                End If
                On Error GoTo 0
            End With
        Else
            If This.fAddRefOuter Then
                This.pOuter.AddRef
                This.fAddRefOuter = False
            End If
            VBoots.AssignZero This.pOuter
        End If
        'Debug only code:allows call to ClearROT to
        'enable normal shutdown. See comments in ClearROT routine.
        #If DEBUGROT Then
        If ExposeObject = 0 Then
            On Error Resume Next
            m_ObjColl.Add VarPtr(This), CStr(VarPtr(This))
            On Error GoTo 0 'Clear the error just in case.
        End If
        #End If
End Function
Private Function HideObject(This As IROTHookImpl) As Long
Dim tmpRegister As Long
        With This
            If .dwRegister Then
            'Guarantee that this isn't reentered.
            tmpRegister = .dwRegister
            .dwRegister = 0
            If .fLocked Then put_Locked This,False
            .fPassThrough = False
            HideObject = RevokeActiveObject(tmpRegister)
```

```
            Set .UnkHook = Nothing
              End If
        End With
    End Function
    Private Function get_Locked( _
      This As IROTHookImpl, reval As Boolean) As Long
        retVal = This.fLocked
    End Funtion
    Private Function put_Locked (_
      This As IROTHookImpl,ByVal RHs As Boolean) As Long
       With This
            If .fLocked <> RHS Then
                If .dwRegister Or .fLocked Then
                    .fLocked = RHS
                    .fPassThrough = False
                    put_Locked = CoLockObjectExternal( _
                      VarPtr(This), -RHS, 1)
                    .fPassThrough = True
                Else
                    'Object variable not set.
                    put_Lucked = &H800A005B
                End If
            End If
        End With
    End Function

    'IQIHook implementation.
    Private Function QueryInterfaceQIHook( _
       ByVal This As Long,riid As VBGUID,pvObj As Long) As Long
    Dim fOK As BOOL
           Select Case riid.Datal
                    Case 0&
              fOK = IsEqualGUID(riid, IID_IUnknown)
          Case &H20708EE4
              fOK = IsEqualGUID(riid, IID_IQIHook)
          Case Else
```

```
            fOK = BooL_FALSE
        End Select
        If fOK Then
          pvObj = This
        Else
          pvObj = 0
          QueryInterfaceQIHook = E_NOINTERFACE
        End If
    End Function
    Private Function AddRefReleaseQIHook( _
      ByVal This As Long) As Long
        'Nothing to do.
    End Function
    Private Function QIHook(This As IQIHookImpl, riid As VBGUID, _
      ByVal uhFlags As UnkHookFlags, pResult As stdole.IUnknown, _
      ByVal HookedUnknown As stdole.IUnknown) As Long
       If riid.Datal = &H19 Then
         If IsEqualGUID(riid, IID_IExternalConnection) Then
           With This
              If Not .pExternal Is Nothing Then
                .cRefsExternal = .cRefsExternal + 1
                VBoost.Assign pResult, _
                  Varptr(.pVTableExternal)
              End If
           End With
         End If
       End If
    End Function

'IExternalConnection implementation.
Private Function QueryInterfaceExtCon( _
  This As IExtConImpl, riid As VBGUID,pvObj As Long) As Long
    QueryInterfaceExtCon=
                  This.pExternal.QueryInterface(riid,pvObj)
End Function
Private Function AddRefExtCon(This As IExtConImpl) As Long
```

```
    'No need to AddRef the main object. It's already kept alive
    'by the single reference we have on it.
    With This
        .cRefsExternal = .cRefsExternal + 1
        AddRefExtCon = .cRefsExternal
    End With
End Function
Private Function ReleaseExtCon(This As IExtConImpl) As Long
  .With This
      .cRefsExternal = .cRefsExternal - 1
      ReleaseExtCon = .crefsExternal
      If .cRefsExternal = 0 Then
         If .fAddRefOuter Then
            .pOuter.AddRef
            .fAddRefOuter = False
         End If
         Set .pExternal = Nothing
         'Nothing left to hook.
         Set .UnkHook = Nothing
      End If
  End With
End Function
Private Function AddConnection(This As IExtConImpl, _
  ByVal extconn As Long, ByVal reserved As Long) As Long
    With This
        AddConnection = .pExternal.AddConnection( _
          extconn, reserved)
        .cConnections = .cConnections + 1
    End With
End Function
Private Function ReleaseConnection(This AS IExtConImpl, _
  ByVal extconn As Long, ByVal reserved As Long, _
  ByVal fLastReleaseCloses As Long) As Long
Dim tmpROTHook As IROTHook
    With This
        ReleaseConnection = .pExternal.ReleaseConnection( _
```

```
        extconn, reserved, fLastReleaseCloses)
      .cConnections = .cConnections - 1
      If .cConnections = 0 Then
        If .fAddRefOuter Then
          .pOuter.AddRef
          .fAddRefOuter = False
  End If
        'Back up two VTable entries (8 bytes)
        'to get at the ROTHook object.
        VBoost.Assign tmpROTHook, _
          VBoost.UDif(VarPtr(.pVTableExternal), 8)
        tmpROTHook.HideObject
        VBoost.AssignZero tmpROTHook
      End If
  End With
End Function
Private Function FuncAddr(ByVal pfn As Long) As Long
  FuncAddr = pfn
End Function
```

第十一章

函数指针的调用

尽管 Visual Basic 中的 AddressOf 运算符允许我们定义一个函数指针作为外部操作的回调，但实际上 VB 本身并没有提供一种调用函数指针的方法。不过，通过与轻量对象的连接和一些汇编代码，我们可以很容易地创建可以随意调用函数指针的对象。

标准调用(C++中的_stdcall)只是几个调用规则中的一个，但 VB 能够支持的只有它。与所有 COM 程序、Win32 API 函数调用一样，所有 VB 函数使用_stdcall 调用规则。调用规则用来规定参数如何传递给函数、函数如何返回结果以及如何清除堆栈。使用_stdcall，参数直接在栈内进行传递，函数负责清理栈区；在调用函数返回后，所有在调用时压入到堆栈的参数被从栈中清除。函数返回值被存储到 eax 寄存器中，至少是四字节整数类型。在_stdcall 中，this 指针作为最左边的参数传递。

由于 VB 能够调用类型库中定义的接口中的所有方法，但是不能调用任意的函数指针，所以我们必须创建一个 COM 对象，这几乎不用做任何事情，只需把所有参数（除了 this 指针以外）都传给函数指针就可以了。假设 COM 对象以四位偏移量存储函数指针，位于 VTable 指针后的第一个位置。让我们来看一看是如何进行调用的，先注意一下 COM 调用开始时栈的情况。堆栈指针位于列表顶端（较低的地址在先）。

```
return address
this
parameter 1
...
parameter n
```

用相同的参数调用函数指针，栈在函数开始时内容如下：

```
return address
```

```
parameter 1
...
parameter n
```

为了将调用传递到一个函数指针，COM 程序只是取出 this 指针，然后控制函数指针。所执行的编译代码如下所示。值得我们注意的是：在代码中，参数和返回值都保持不变，所以我们可以向具有相同参数的函数指针传递包含 this 指针的标准 VTable 调用（其中包括了所有的 COM 调用）。

```
// Retrieve the return address and
// remove it from the stack (59).
pop ecx
// Retrieve the this pointer and
// remove it from the stack (58).
pop eax
// Put the return address back on the stack (51).
push ecx
// Jump to the function pointer at this + 4 (FF 60 04).
jmp DWORD PTR [eax + 4]
```

编译代码生成的字节为 59 58 51 FF 60 04（十六进制）。现在来看一下轻度对象的连接。为了使得 VB 能够调用这些汇编代码，必须创建一个 COM 对象，并在它的某一个 VTable 函数中包含这些字节流。下面，将使用一个在最开始的三个 VTable 位置处带有标准 IUnknown 函数的轻度 COM 对象，这些字节流被放在第四个位置处。函数 QueryInterface 的第一次调用是非常值得信任的：它假定我们正在请求一个与函数指针所支持的参数相匹配的接口。所有随后的 QueryInterface 函数都将不起作用。

由于实际的汇编语言程序代码只有六个字节，可以采用两条 int 3 指令（CC CC）进行补充，这样便有八个字节。它能以货币型常量存储。int 3 相当于一条 break 语句，所以在出错时我们将会看到一个系统错误对话框。int 3 是一个标准填充函数；nop（无操作，字节代码 90）则是另一个标准填充函数。因为常量不在可执行存储单元内——运行 VarPtr（asmconst）代码将会导致崩溃——所以数据从常量拷贝到了模块级的货币型变量中。该货币型变量的 VarPtr 是 VTable 的第四项。下面的代码中给出了 InitDelegator 函数，该函数对 FunctionDelegator 轻度对象的堆栈分配版本进行了初始化。用 NewDelegator 产生的采用堆分配的轻度对象的代码包含在本书所附光盘的 FunctionDelegator.bas 文件中。我们可以编译出使用 FUNCTIONDELEGATORNOHEAP 的版本或是 FUNCTIONDELEGATOR_NOSTACK 条件编辑值。

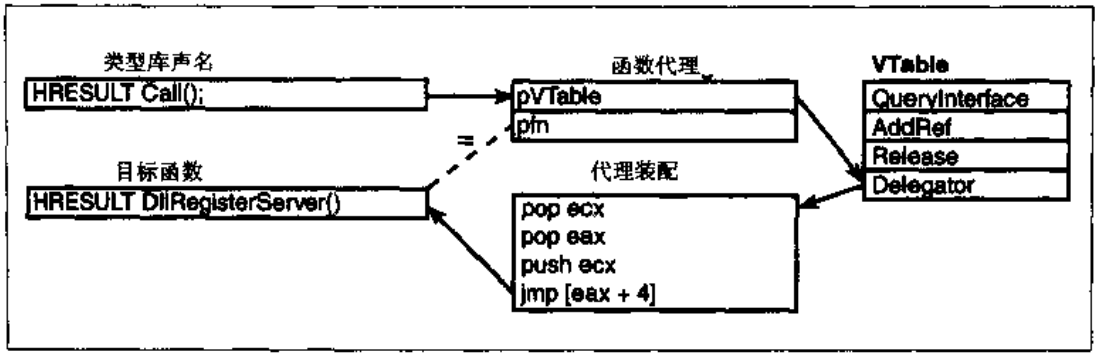

图 11.1 FunctionDelegator 重定向结构

```
'The magic number
Private Const cDelegateASM As Currency =_
-368956918007638.6215@
'Module-level variables
'The variable to hold the asm bytes
Private m_DelegateASM As Currency

Private Type DelegatorVTables
    'OKQI VTable in 0 to 3 FailQI VTable in 4 to 7
    VTable(7) As Long
End Type

'Structure and VTables for a stack allocated Delegator
Private m_VTables As DelegatorVTables

'pointer to success VTable, stack allocation version
Private m_pVTableOKQI As Long

'pointer to failing VTable, stack allocation version
Private m_pVTableFailQI As Long

'Type declaration
Public Type FunctionDelegator
    pVTable As Long 'This has to stay at offset 0.
    pfn As Long     'This has to stay at offset 4.
```

```
End Type

'Initialize an existing FunctionDelegator structure
'as a delgator ovject.
Public Function InitDelegator( _
  Delegator As FunctionDelegator, _
  Optional ByVal pfn As Long) As IUnknown
     If m_pVTableOKQI = 0 Then InitVTables
     With Delegator
         .pVTable = m_pVTableOKQI
         .pfn = pfn
     End With
     CopyMemory InitDelegator, VarPtr(Delegator), 4
End Function
Private Sub InitVTables()
Dim pAddRefRelease As Long
    With m_VTables
        .VTable(0) = FuncAddr(AddressOf QueryInterfaceOK)
        .VTable(4) = FuncAddr(AddressOf QueryInterfaceFail)
      pAddRefRelease = FuncAddr(AddressOf AddRefRelease)
        .VTable(1) = pAddRefRelease
        .VTable(5) = pAddRefRelease
        .VTable(2) = pAddRefRelease
        .VTable(6) = pAddRefRelease
        m_DelegateASM = cDelegateASM
        .VTable(3) = VarPtr(m_DelegateASM)
        .VTable(7) = .VarPle(3)
        m_pVTableOKQI = VarPtr(.VTable(0))
        m_pVTableFailQI = VarPtr(.VTable(4))
    End With
End Sub

'VTable functions for the stack-allocated delegator.
Private Function QueryInterfaceOk(This As FunctionDelegator _
  riid As Long, pvObj As Long) As Long
   'Return this object.
```

```
    pvObj = VarPtr(This)
    'Block all future QI calls.
    This .pVTable = m_pVTableFailQI
End Function
Private Function QueryInterfaceFail _
  (ByVal This As Long, riid As Long, pvObj As Long) As Long
    pvObj = 0
    QueryInterfaceFail = E_NOINTERFACE
End Function
Private Function AddRefRelease(ByVal This As Long) As Long
    'Nothing to do ; memory not refcounted.
End Function
'Helper function
Private Function FuncAddr(ByVal pfn As Long) As Long
    FuncAddr = pfn
End Function
```

11.1 示例：调用 DLLRegisterServer

既然 InitDelegator 函数提供了一个 COM 对象，它与函数指针相关联，我们只需要指示 VB 编译器调用 VTable 中相应的函数。VB 知道如何来调用类型库定义的接口，所以我们只需求助于类型库构造器以定义一个 VB 能够调用的接口。一旦接口定义并被引用，对象 FunctionDelegator 允许 QI 调用这个接口（或任何其他接口），并且 VB 中含有一个能进行调用的引用。因为这里用到的函数模型返回 HRESULT，且如果失败，VB 会调用 ISupportErrorInfo 接口，这样以后的所有 QI 对 FunctionDelegator 轻度对象的调用都将失败。下面，让我们来看看在 COM DLL 中调用函数 DllRegisterServer 入口的过程。函数 DllRegisterServer 没有参数，最后返回 HRESULT 类型的变量。

首先，我们必须在 VB 外部定义接口。既然下面用到的接口在 VBoostTypes 中已经定义过了，所以这些步骤用于普通函数指针。下面给出了一个合理的 ODL 定义。（读者可以参考第十五章中对 MIDL/IDL 和 mktyplib/ODL 的讨论。）可将它作为 FuncDeclLib.Odl 存储，然后调用 mktyplib FuncDeclLib.Odl。在最终的 FuncDeclLib.Odl 中加入一个工程引用。注意这里接口定义为 IUnknown-derived 类型，且只包含了一个函数。这与构造的 VTable 相匹配。接口的 IID 值将用于构建类型库，但在其他地方并不会用到。

```
[
   uuid(8F633500-459A-11d3-AB5C-D41203C10000),
   helpString("Function pointer definitions"),
   lcid(0×0),
   version(1.0)
]

library FuncDeclLib
{
   importlib("stdole2.tlb");
    [uuid(8F633501-459A-11d3-AB5C-D41203C10000), odl]
   interface ICallVoidReturnHRESULT : IUnknown
   {
      HRESULT Call();
   }
}
```

由于我们已经定义了可调用的接口，那么下面需要调用 LoadLibrary 和 GetProAddress API 以获得 DllRegisterServer 的函数指针。一旦获得一函数指针——pfnDllRegisterServer——我们便可以调用 InitDelegator（pfnDllRegisterServer），然后为 ICallVoidReturnHRESULT 类型的变量分配一个对象。一个简单的对类型库中定义的 Call 方法的调用被用来在轻度对象中调用 asm 代码，这需要服从函数指针。

```
'modDLLRegister
'API declares are in the VBoosTypes type library.

Public Sub DllRegisterServer(DllName As String)
   CallDllRegEntry DllName, "DllRegisterServer"
End Sub
Public Sub DllUnregisterServer(DllName As String)
   CallDllRegEntry DllName,"DllUnregisterServer"
End Sub
Private Sub CallDllRegEntry (DllName As String, _
  EntryPoint As String)
Dim pCall As ICallVoidReturnHRESULT
Dim Delegator As FunctionDelegator
```

```
    Dim hMod As Long
    Dim pfn As Long
        'Load the DLL.
        hMod = LoadLibrary(DllName)
        If hMod = 0 Then Err.Raise 5

        'Error trap to make sure we free the library.
        On Error GoTo Error

        'Find the entry pointer.
        pfn = GetProcAddress(hMod, EntryPoint)
        If pfn = 0 Then Err.Raise 5

        'Create and assign the delegator object.
        Set pCall = InitDelegator(Delegator, pfn)

        'Call the function pointer.
        pCall.Call
    Error:
        'Free the library handle.
        FreeLibrary hMod
        'Propagate any error.
        With Err
            If .Number Then .Raise .Number
        End With
    End Sub
```

通过在适当位置调用代理编译代码，对函数指针的调用将变得十分的简单。获得函数指针要比初始化代理对象和进行调用多做很多工作。如果 FunctionDelegator 和其结构在同一范围内进行声明的话（局部的，标准模式或者作为类模式的成员变量），将导致它们同时超出范围。这意味着引用 FunctionDelegator 结构的对象变量仍含有一有效的 VTable 变量，因为 VB 在释放类或模块的其他内存之前要对所有引用调用 Release 函数。要提到的最后一点便是：只有当我们没有在与 FunctionDelegator 结构同样的范围上声明对象变量时，FunctionDelegator 的采用堆分配的 NewDelegator 版本才可能被用到。

在 DllRegisterServer 范例中，我们看到过一个可指向任意 DLL 的调用。我们并不能只通过 VB 声明就进行这些调用。不过，可以使用 FunctionDelegator 来调用那些在设计时并没有定义过的 DLL。FunctionDelegator 调用允许动态地装入和卸载 DLL。可以通过

LoadLibrary 或 LoadLibraryEx API 函数调用来装入一个 DLL，然后用 FreeLibrary 来卸载它。如果我们很少将 DLL 用做某一长进程的一部分，那么使用这种方法将允许我们在不使用 DLL 的时候释放内存及系统资源。

VBoostTypes 中包含了几个简单的接口，它向我们提供了调用典型函数指针的功能并且无需生成自己的类型库。表 11.1 中列出了所支持的函数原型。

表 11.1 在 VBoostTypes6.Olb 中的 FunctionDelegator 接口函数声明

接 口 名	C++ 函数属性
ICallVoidReturnVoid	void Call(void);
ICallLongReturnVoid	void Call(long);
ICallLongReturnLong	Long Call(long);
ICallVoidReturnLong	Long Call(void);
ICallVoidReturnHRESULT	HRESULT Call(void);
ICallLongReturnHRESULT	HRESULT Call(long);

11.2 示例：QuickSort，一劳永逸

在 VB 中我们很少看到排序程序，不过这并没有让我退却。不管采用何种算法，排序程序都是对一系列元素进行比较和交换。在 VB 中，对元素进行比较十分的简单，但是需要为每种类型都提供一个新的排序程序。每种数据类型对应一个新的程序并不足以引起我们的恐慌，关键是还需要对交换代码进行测试。尽管交换元素可能看起来挺简单，但由于 VB 总是执行一个深度拷贝，而这对于指针类型来说实在代价太大。考虑交换两个字符串的代码如下。

```
Dim tmp As String
tmp = StrArray(i)
StrArray(i) = StrArray(j)
StrArray(j) = tmp
```

上面的代码对开始位于 i 位置处的字符串作了两次拷贝，而对开始位于 j 位置处的字符串只作了一次拷贝。既然在排序过程中每一项都要交换多次，因此这样的程序将最终需要完整的拷贝数组中所有数据多次。数据拷贝要比排序本身慢得多。

关于深度拷贝问题的解决办法是使用一个伴随长整型数组，该数组被初始化为 0、1、…，n-1。通过在索引数组中交换元素（这很实用，因为没有指针的分配），我们可以保

存字符串的深度拷贝。接下来我们要在排序之后保存索引数组。StrArray（index（0））是第一个排序项。这种机制的一个优点是可以对同一数组进行多种排序。不过，即便是使用索引，我们也仍然需要为每种数据类型编写不同的排序程序。

在 C/C++中排序，我们不必担心深度拷贝问题或者对多种数据类型的排序。C 和 C++中使用回调函数对每一项进行比较，并通过指针算法对数组进行操作。实际的排序程序并不知道也不需要考虑所要排序的数据类型；只需要知道数据类型在内存中的大小，数组中元素的数目，以及用来比较两个数组元素的函数指针就可以了。

既然 VB 能调用函数指针，那么也就可以采用 C++类型的排序方法。首先，我们必须定义函数指针。由于排序程序采用独占方式操作指针值，并且 VB 中的指针值是长整型，类型库中定义的 Compare 函数中包含两个 ByVal As Long 类型的参数。比较函数的一般标准为：当两个元素比较，如果小于则返回-1，相等则返回 0，大于则返回 1。我们可以遵循这一标准来定义一个返回长整型值的比较函数。函数指针的 ODL 描述如下所示。返回一个长整型值而非返回一个 HRESULT 的接口函数看起来可能有些不同寻常，但是 VB 却可以调用这种类型的接口函数，并且它可以大大地简化函数指针的定义。

```
[uuid(C9750741-4659-11d3-AB5C-D41203C10000), odl]
interface ICallCompare : IUnknown
{
   long Compare([in] long Elem1, [in] long Elem2);
}
```

现在，我们必须在 VB 中写入相应的函数。这里，一个比较字符串的函数是一个很好的例子。需要注意的是，这个函数的参数并不是 ByVal As Long 类型。为了保持比较回调程序无需知道数据类型的特性，排序程序通知 VB 它正将两个长整型值传递给函数，但我们知道这些值实际上都是指向数组中元素的指针。回调函数知道数组元素的类型，因此 CompareStrings 函数中的回调参数被定义为 ByRef As String 类型。当然我们也可以采用大量的 CopyMemory 调用来废弃 ByVal As Long 型参数而使用某种已知类型。只不过在这里戏弄一下 VB，而插入一个具有正确参数类型的函数指针要更加容易。正确的输入类型将使得 CompareString 十分的简单。

```
Public Function CompareStrings( _
    Elem1 As String, Elem2 As String) As Long
      CompareStrings = Sgn(StrComp(Elem1,Elem2))
End Function
```

由于已经定义过回调函数指针，所以我们现在可以采用标准的 QuickSort 算法。下面所

示的实现过程包含三个函数，它们分别是：QuickSort，它创建了 FunctionDelegator 对象并为交换元素分配了缓冲器；QuickSortInternal，它执行排序算法的实际递归程序；SwapBytes，用来交换排序数组中的元素。

```
'Pass a structure to the recursive QuickSoreInternal
'function to minimize stack usage.
Private Type QuickSortData
    Size As Long
    Compare As ICallCompare
    ByteSwapper() As Byte
End type

Pulic Sub QuickSort( _
  ByVal pBase As Long, _
  ByVal Number As Long, _
  ByVal Size As Long, _
  ByVal pfCompare As Long)
Dim Data As QuickSortData
Dim Delegator As FunctionDelegator
    With Data
        .Size = Size
        Set .Compare = InitDelegator(DeleGator, pfCompare)
        ReDim .ByteSwapper(0 To Size - 1)
        QuickSortInternal _
          pBase, pBase + (Number - 1) * Size,Data
        Set .Compare = Nothing
    End With
End Sub

Private Sub QuickSortInternal(ByVal Low As Long, _
   ByVal High As Long, Data As QuickSortData)
Dim PivotIndex As Long
Dim i As Long , j As Long
If Low < High Then
   'Only two elements in this subdivision; swap them
   'if they are out of order, then end recursive calls.
```

```
    If High - Low = Data.Size Then
       If Data.Compare.Compare(Low,High) > 0 Then
          SwapBytes Low, High, Data
       End If
    Else
       'Pick a pivot element in the middle, then move
       'it to the end.
       'Don't use (Low + High) \2:it can overflow
       PivotIndex = Low \ 2 + High \ 2
        .
       'Align on m_Size boundary
       PivotIndex = _
         PivotIndex - ((PivotIndex - Low) Mod Date .Size)
       SwapBytes High, PivotIndex, Data

       'Move in from both sides toward the
       'pivot element.
       i = Low: j = High
       Do
          Do While (i < j) And _
            (Data .Compare .Compare(i, High)<=0)
            i = i + Data.Size
          Loop
          Do while (j < i) And _
            (Data .Compare .Compare(j,High)>=0)
            j = j-Data.Size
          Loop

          'If we haven't reached the pivot element,it
          'means that two elements on either side are
          'out of order, so swap them.
          If i<j Then
             SwapBytes i, j,Data
          End If
       Loop While i < j
       'Move the pivot element back to its proper
```

```
        'place in the array.
        SwapBytes i, High, Data

        'Recursively call the QuickSortInternal procedure.
        'pass the smaller subdivision first to use less
        'stack space.
        If (i-Low) < (High-i) Then
           QuickSortInternal Low, i - Data.Size, Data
    QuickSortInternal i + Data.Size, High,Data
        Else
           QuickSortInternal i + Data.Size, High, Data
           QuickSortInternal Low,i - Data.Size,Data
        End If
     End If
End If
End Sub

Private Sub SwapBytes( _
  ByVal pElem1 As Long, ByVal pElem2 As Long, _
  Data As QuickSortData)
    With Data
       CopyMemory .ByteSwapper(0),ByVal pElem1, .Size
       CopyMemory .ByVal pElem1, ByVal pElem2. Size
       CopyMemory ByVal pElem1, .ByteSwapper(0), .Size
    End With
End Sub
```

```
'Calling QuickSort,String sorting routine.
Public Sub SortStrings(Strings() As String)
    'Assume one-dimensional array.
    QuickSort VarPtr((LBound(Strings)), _
      UBound(Strings) - LBound(Strings) + 1 , _
      4, AddressOf CompareStrings
End Sub
```

这一排序程序可用于任何数据类型。我们所要做的工作只是为特殊数据类型提供一个

Compare 函数并指出数据元素的长度。需要注意的是：确定一个 UDT 数组元素的长度在这里十分关键。如果猜测的话很容易出错，而使用 LenB 也并不总能保证得出正确的元素长度。读者可以参考第二章中的“确定元素长度”小节，以获得有关在运行时获得数组元素长度方面的更多信息。

11.3 Alpha 中的 VB 函数指针

标准 Alpha 函数调用机制与_stdcall 调用习惯不同。_stdcall 通过堆栈来传递 this 指针和所有参数。在 Alpha 调用中，前六个参数，包括 this 指针，在寄存器中直接传递；随后的参数通过堆栈传递。this 指针并不容易通过堆栈操作从数据项中提取出来，这样的话，Intel 类型的函数指针技术便不再有效。不过，另外还有一种功能强大（虽然不很雅致）的解决办法。

作为一种调用函数指针的方法，这种变通机制提供了一个直接插入到 VTable 中的函数指针。由于 VTable 是直接进行调用，故在这里称其为 DirectFunctionDelegator 对象而非 FunctionDelegator。在这种机制中，由于每个 VTable 都不一样，所以每一 DirectFunctionDelegator 实例都拥有自己的 VTable，。在很多情况下，这部分代码很容易编写。下面是 DirectFunctionDelegator 轻度对象中作了相应变化的函数和结构。这里不再有模块级变量，因为 VTable 不能共享。我们可以在 DirectFunctionDelegator.Bas 中找到这部分代码。

```
Private Type DirectFunctionDelegator
   pVTable As Long
   VTable(3) As Long
End Type

Public Function InitDirectDelegator( _
  Delegator As DirectFunctionDelegator, _
  ByVal pfn As Long) As IUnknown
    With Delegator
        .VTable(0) = FuncAddr
          (_AddressOf QueryInterface-NOAlloc)
        .VTable(1) = FuncAddr(AddressOf AddRefReleaseNOAlloc)
        .VTable(2) = VTable(1)
        .VTable(3) = pfn.
        .pVTable = VarPtr(.VTable(0))
        CopyMemory InitDirectDelegator, VarPtr(.pVTable), 4
    End With
```

```
    End Function
    Private Function QueryInterfaceOKQI( _
      This As DirectFunctionDelegator, _
       riid As Long, pvObj As Long) As Long
         pvObj = VarPtr(This)
         This.VTable(0)= FuncAddr(AddressOf QueryInterfaceFail)
    End Function
    Private Function QueryInterfaceFail( _
       ByVal This As Long,riid As Long,pvObj As Long) As Long
    pvObj = 0
       QueryInterfaceFail = E_NOINTERFACE
    End Function
    Private Function AddRefRelease(ByVal This As Long) As Long
       'Nothing to do, memory not refcounted.
End Function
```

当然，因为提供的代码不再具有提取 this 指针的功能，提供的函数指针在参数列表中必须包含额外的模块。不过，用于进行函数调用的类型库声明并没有发生任何的变化。对于所提供的函数指针，其第一个参数应声明为 ByVal This As Long 类型，并且不应被使用。下面的代码对 QuickSort 示例中的 CompareStrings 函数进行了修改。

```
Public Function CompareStrings( _
    ByVal This As Long, _
    ElemI As String, Elem2 As String) As Long
       CompareStrings = Sgn(StrComp(Elem1, Elem2))
End Function
```

通过 GetProcAddress 所取得的函数指针的函数签名包含了一个虚拟的长整型值，并将其作为第一个参数。因此，我们不能使用一个 DirectFunctionDelegator 来调用任意的函数指针。然而，如果该函数指针没有任何参数的话，那么在 Alpha 平台中 this 指针简单的被忽略。由于上面所列出的 DLL 注册函数中不带有任何参数，因此我们仍然可以在 Alpha 平台中调用它们。不过，对于其他大部分函数而言，却没有如此幸运。

11.4 堆 栈 分 配

在 Intel 平台中，使用 DirectFunctionDelegator 经常会发生一些奇怪的事情。如果我们在先前所列出过的 CallDllRegEntry 函数中使用一个 DirectFunctionDelegator 来代替 FunctionDelegator， 函数看起来工作正常。这里存在一些问题，因为它和函数的预定义不匹配。但是，各种预定义在 DllRegisterServer 函数所搜寻的堆栈位置之后并不匹配。编译器将 4 个字节推至堆栈上端，并且 DllRegisterServer 函数弹出 0，实质上漏掉了堆栈中的 4 个字节。在函数返回时，遗漏的字节被收回，因此这还不算是一个大的永久的损失。

为了在进行调用之前和之后都能够监视堆栈的使用情况，我们收集了更多字节的 asm 代码，并将其塞入到一个函数代理中。该函数中不包含任何参数并返回一个长整型值，因此我们可以使用由 VBoostTypes 所提供的 ICallVoidReturnLong 接口来对其进行调用。

```
'Assembly code we're using to make a stack pointer.
  'mov eax, esp ;Get the current stack pointer,8B C4
  'add eax, 4   ;Account for return address, 83 C0 04
  'ret          ;Return, C3

  Private Const cCheckESPCode As Currency = _
    -368935956954613.2341@
  Private CheckESPCode As Currency
  Private pCallObj As ICallVoidReturnLong
  Private FD As FunctionDelegator
  Public Property Get GetESP() As ICallVoidReturnLong
     If pCallObj Is Nothing Then Init
     Set GetESP = pCallObj
  End ProPerty
  Private Sub Init()
     CheckESPCode = cCheckESPCode
     Set pCallObj = InitDelegator(FD,VarPtr(CheckESPCode))
  End Sub
```

```
  'Calling code
  Debug.Print Hex$(GetESP.Call)
```

如果我们运行了 CallDllRegEntry 的 DirectFunctionDelegator 版本，并紧接着在对

DllRegisterServer 进行代理调用之后立即检查当前的堆栈位置，那么我们将会看到堆栈值下降了 4 个字节。这里，有两件事情需要指出。首先，VB 并不会对堆栈中的差异进行抱怨，即便在 IDE 环境中也是如此。但要是我们是进行一个 API 函数调用并以同样的方式弄乱了堆栈的话，VB 会十分清楚地提出抗议。该检查在 IDE 环境中进行，使我们能够对 API 声明进行调试。在运行时，不会进行堆栈检查，但 IDE 环境中的堆栈检查使得发生堆栈泄漏的 API 函数调用使用起来非常地困难。

第二件需要指出的事情是：在一般的编程中，调用一个修改堆栈的函数不会有错。DllRegisterServer 并不会故意发生泄漏，发生泄漏是因为进行了不正确的调用，不过发生泄漏有时也是一件好事。实际上，C++中的 the_alloca 函数是明确设计用来将数据推入额外的堆栈的一个函数。在 C++中，我会选择尽可能的使用堆栈分配，因为这样可以在程序中大量减少堆分配的数量。由于 VB 并不在意在一个 VTable 调用期间进行压栈操作，我们可以自行编写一个堆栈分配规则以在 VB 中使用而不立即受到惩罚，就好像调用一个 API 函数一样。

在看到 StackAlloc 代码之前，一小段关于如何使用堆栈的讨论将会让我们受益。对此有所了解，我们在进行函数调用期间便能更为自如地移动堆栈指针。我们还将看到，如果不对堆栈泄漏进行管理，将会对函数调用产生破坏作用。存在两个用来控制堆栈的 CPU 寄存器：esp(堆栈指针)和 ebp（基指针）。其中，esp 是堆栈的当前位置。压栈（即在堆栈中加入新数据）操作会导致 esp 值减少。同样弹栈操作会增加 esp 的值。ebp 被称为堆栈帧，被用来为每一函数提供一个堆栈中的基引用点。在一函数中运行的初始指令被称为 prolog 代码。标准的 prolog 代码看起来大致如下。

```
push ebp              ;push the current stack frame
mov ebp,esp           ;ebp=esp, use the stack position as the
                      ; new frame
sub esp, localsize ;esp=esp-localsize, push stack for local
                      ;variables
push ebx              ;save the ebx register on the stack
push esi              ;save the esi register on the stack
push edi              ;save the edi register on the stack
```

标准的 prolog 代码通过推其入栈对当前的堆栈帧进行了保存，将该帧上移至当前的堆栈位置，并将其完全入栈以处理其局部变量。然后将必须维持用以进行跨函数调用的寄存器推入栈。在 prolog 代码之后，堆栈看起来应与下面相似，当前堆栈指针位于列表的顶部。

```
previous value of edi (stack pointer after prolog)
previous value of esi
```

```
previous value of ebx
      ...
local variable 2
local variable 1
stack pointer before prolog (ebp points here)
return address
parameter 1
...
parameter n
```

当 prolog 代码完成，函数即可运行其代码。由于建立了 ebp，可以通过相对于 ebp 的正偏移来访问参数，例如，参数 1 为 ebp+8。而局部变量则为负偏移，例如，局部变量 1 为 ebp-4。当函数完成时，便运行类似于下面的代码。这些代码被称为 epilog 代码。

```
pop edi          ;restore edi register
pop esi          ;restore esi register
pop ebx          ;restore ebx register
mov esp,ebp      ;undo the second statement
                 ;in the prolog code
pop ebp          ;undo the first statement
                 ;in the prolog code
ret n            ;return, taking an additional n bytes of
                 ;stack. n is the number of bytes
                 ;pushed as parameters.
```

成功地离开一个函数的关键在于在进入 epilog 代码时，堆栈位置与 prolog 代码结束时相同。在各行 VB 代码都已成功地执行之后，堆栈将总是位于正确的位置，从而完美结束该函数。在整个函数运行期间，为局部变量空间建立所需空间时，所需的压栈操作已经完全进行完毕。

由于所有的局部变量都以堆栈帧偏移量来进行访问，VB(包括一般的 Win32 代码在内)并不关心在函数体中运行时堆栈指针的位置。指针只需在 epilog 代码运行时能够回到正确的位置便可以了。如果堆栈指针没有正确的返回到原来的位置，edi、esi 和 ebx 寄存器没有能够正确的恢复。寄存器的破坏将导致函数调用产生各种各样的不可预测的问题。例如，esi 和 edi 寄存器被用作循环计数器。因此，破坏这些寄存器将糟糕的使当前调用的函数陷入一个无休止的循环中。破坏了的寄存器于分具有欺骗性：堆栈帧本身并没有被破坏，所以在这种情况下，函数仍然正确的返回，并且没有任何发生错误的迹象。

由于在 prolog 代码运行时，我们必须已经使得 esp 位于其原始位置上，我们所使用的任何堆栈分配规则都必须通过一个弹出相应数量堆栈的调用来进行利用。在函数体期间，堆栈主位置（恰好位于缓冲寄存器值之上）即是每行 VB 代码开始时的当前堆栈指针。VB 不会提前主动地在每次调用之后都将堆栈位置恢复到这一位置之上；它只是按需要将数据推入或弹出堆栈。在每行代码开始处，esp 都保持相同，其原因很简单：VB 不产生堆栈泄漏。由于 esp 已经习惯于将值推入或弹出堆栈，直到发生错误为止，额外的内存都被忽略且保持呈良性。而且 VB 会试图恢复堆栈指针。

由于在一行 VB 代码执行期间，错误可在任意时刻抛出（不一定非得在刚开始），因此当发生了一个错误时，不能完全确定堆栈位置。错误处理器本身同样需要使用堆栈进行工作，并且从主位置处开始进行。这样做的好处是如果触发了一个错误，VB 能够自动的清除堆栈并正确的对寄存器进行恢复。而坏处是：一旦触发了一个局部错误处理器或在 On Error Resume Next 模式期间产生一个错误，我们便不能再使用任何的额外堆栈。在一个 Resume 或 Resume Next 语句之后，无论是在错误处理器中还是在函数体内，堆栈都将移回到其主位置上。

由上面的讨论，我们可以总结出一个简单规则：为了维持好堆栈，我们将多少额外数据推入了堆栈，则必须从堆栈中弹出同样比特数目的数据，除非导致了错误的发生。如果发生了错误，则推入的额外数据将被覆盖且立即丢失。如果我们遵守了这一指导原则，在 VB 中进行堆栈分配将是十分安全的。如果没有遵守这一准则，当我们恢复堆栈指针失败时，我们将会破坏寄存器。如果在错误管理器已将数据从堆栈中弹出的情况下进行弹栈，将会导致程序很快崩溃。

本书中包含了三个堆栈分配函数的代码：StackAlloc、StackAllocZero 和 StackFree。其中，StackAlloc 函数将所要求数目的比特推入堆栈，并返回一个指向该块的指针。StackAllocZero 函数则除推入数据之外还对内存进行零初始化。StackFree 函数负责将要求数目的比特从堆栈中弹出。除了要求更多的汇编代码之外，这些函数的代码看起来与前面所给出的 GetESP 代码十分相似。基于性能和稳定性方面的原因，所要求的比特数目必须是 4 的倍数。如果不是这样，在进行压栈之后所调用的任何函数用到的都是方向偏离了的数据。VB 中不能处理发生了方向偏离的堆栈和 Y 崩溃。通常，AlignedSize 函数被包含于 StackFree.Bas 中。

前面已经说过我们已经看到了自己最终的 QuickSort，但这里给出了一个修正版本，该版本使用堆栈分配来取代 ReDim 语句。经过这一变化，整个 QuickSort 程序将运行在完全没有使用堆分配的环境中。

```
Private Type QuickSortData
    Size As Long
    Compare As ICallCompare
    pBytes As Long
```

```
End type

Public Sub QuickSort( _
  ByVal pBase As Long, _
  ByVal Number As Long, _
  ByVal Size As Long, _
  ByVal pfCompare As Long)
Dim Data As QuickSortData
Dim FDCompare As FunctionDelgator
Dim cBytes As Long
    With Data
        .Size = Size
        Set .Compare = InitDelegator(FDCompare, pfCompare)
        cBytes = AlignedSize(Size)
        .pBytes = StackAlloc.Call(cBytes)
        On Error GoTo Error
        QuickSortInternal pBase, _
          pBase + (Numbe-1) * Size, Data
    End With
    If cBytes Then StackFree .Call cBytes
    Exit Sub
Error:
    With Err
        .Raise .Number, .Source, .Description, _
.HelpFile, .HelpContext
    End With
End Sub

'QuickSortInternal does not change

Private Sub SwapBytes( _
  ByVal pElem1 As Long, ByVal pElem2 As Long, _
    Data As QuickSortData)
    With Data
        CopyMemory ByVal .pBytes, ByVal pElem1, .Size
        CopyMemory ByVal pElem1, ByVal pElem2, .Size
```

```
        CopyMemory ByVal pElem2, ByVal .pBytes, .Size
    End With
End Sub
```

11.5 产生自己的内联汇编

Visual Basic 中从来没有提供一种方法来编写内联汇编代码。然而，有了FunctionDelegator对象，我们便可以接近这一目标。尽管不能让VB来帮我们编译汇编代码，但是我们可以使用 VB 来调用任意函数，该函数甚至可以只是一字节流。如果我们想要编写内联汇编函数，需要使用一种能够编译汇编代码的工具。这样，我们就可以读到编译之后的字节。注意：除汇编代码之外，我们还可以使用这种技术来得到优化之后的 C 函数代码字节。这样，我们便可以不用编写C代码而让C编译器来生成汇编。

作为一个示例，这里编写一段根据当前基指针偏移量来读取数据的汇编代码。所产生的函数采用了一个参数，将其加入到基指针上，然后废弃该指针以从堆栈中读取一个值。该值在eax寄存器中返回（x86编译标准）。该函数的汇编代码是最小化了的。

```
Mov ecx, [esp + 4]
Mov eax, [ebp + ecx]
ret 4
```

在Visual C++ 6.0中我们可以按照下面的步骤来为这一代码产生一个字节流：

（1）在MSVC开发环境中，打开File\New对话框中的Projects项。

（2）选择Win32应用并调用工程GenASM，单击OK。

（3）选择一个简单的Win32应用并单击Finish项以产生工程。

（4）打开GenASM.cpp。

```
#include "stdafx.h"

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     Int       nCmdShow)
{
     // TODO: place code here
     return 0;
}
```

（5）编辑该函数，加入汇编代码。

```
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
      _asm
      {
         mov ecx, [esp + 4]
         mov eax, [ebp + ecx]
         ret 4
       }
       return 0;
}
```

（6）按 F9，在第一个 MOV 语句上放置一个断点。

（7）按 F5 键，编译并运行工程。

（8）在断点上，选择 View\Debug Windows\Disassembly 来察看反汇编之后的汇编代码，大致应如下所示：

```
11:       _asm
12:      {
13:         mov ecx, [esp + 4]
00401028  mov     ecx,dword ptr [esp+4]
14:         mov eax, [ebp + ecx]
0040102C  mov     eax,dword ptr [ebp + ecx]
15:         ret 4
00401030  ret         4
16:      }
```

（9）为了看到实际的字节值，可以打开反汇编窗口中的上下文菜单并选择 Code Bytes 菜单选项来得到：

```
11:       _asm
12:      {
```

```
13:          mov ecx, [esp + 4]
00401028 8B 4C 24 04     mov     ecx,dword ptr [esp+4]
14:          mov eax, [ebp + ecx]
0040102C 8B 44 0D 00     mov     eax,dword ptr [ebp +ecx]
15:          ret 4
00401030 C2 04 00         ret       4
16:      }
```

（10）我们现在可以读到该函数的实际字节值。然而，如果我们想将这些字节存储到一个长整型变量中的话，那么 Intel 的字节规则可能会使得这些字节难以解释。如果要得到有关字节规则的帮助，我们还有一些工作要做。

（11）选择 View|Debug Windows|Memory。

（12）打开 Memory 窗口中的上下文菜单并选择 Long Hex 格式。

（13）返回到反汇编窗口，双击鼠标，在第一行的最左端选择地址（00401028）。

（14）将该值拖至 memory 窗口中的第一行中以长整型格式察看字节值。我们可以将其和先前的代码字节进行比较，以察看不同的字节规则。

```
00401028   04244C8B
0040102C   000D448B
00401030   330004C2
00401034   5B5E5FC0
00401038   3B40C483
0040103C   001EE8EC
```

（15）通过将这些长整型字节与反汇编窗口中的代码字节作比较，我们可以决定与其相关的最后字节，紧接着在其之前可以看到该语句的地址。从头至尾，从右至左计算总共的字节数。在这种情况中，内联汇编之后的行上显示出其地址为 0040133，它是第三行中的第四个字节。

（16）选择并拷贝三行相关代码，并用 90（nop，无操作指令）替代无关代码。

（17）停止调试。

（18）将产生的相关字节拷贝到一个长整型数组中。记住，要将数组声明放入一结构中。这样，当其他模块仍然使用该数组时，VB 便不会在拆分期间将其释放。

（19）确信已将这些 asm 代码保存，这样，我们以后便可对其进行编辑或重新生成这些代码。

通过一个 FunctionDelegator，很容易访问该练习中所生成的代码，并生成下面的代码（简略了汇编代码的说明）。该文件名为 DerefEBP.Bas，包含在光盘之中。

```
Private Type DerefEBPCode
   Code(2) As Long
End Type
Private FD As FunctionDelegator
Private DerefEBPCode As DerefEBPCode
Private pCallObj As ICallLongReturnLong
    If pCallObj Is Nothing Then Init
    Set DerefEBP = pCallObj
End Function
Private Sub Init()
    With DerefEBPCode
        .Code(0) = &H4244C8B
        .Code(1) = &HD448B
        .Code(2) = &H900004C2
    End With
    Set pCallObj = InitDelegator(FD, VarPtr(DerefEBPCode))
End Sub
```

11.5.1 遍历堆栈(Walk the stack)

我们可以使用 DerefEBP 函数来对堆栈来进行研究，并验证前面所提及的有关堆栈设计的任何问题。例如，在一个类模块函数中运行 prolog 代码之后，先前的 ebp 值已经被推入到堆栈之中，且其位置恰好在所返回的地址之上。现在，堆栈中的 ebp 值如下所示：

```
previous ebp value
return address
this pointer
first parameter
```

第一个参数总是偏离 ebp 12 个字节。如果最后一个参数是 ParamArray，那么这是十分有用的：它使我们获得数组指针。当我们试图将 ParamArray 传递到一个帮助函数中的普通 VB 数组参数上时，VB 将产生“Invalid ParamArray use”编译错误，并且甚至将 ParamArray 传递给 VarPtrArray 函数也是不允许的。通过使用 DerefEBP 和 VBoost.AssignSwap，我们可以很容易的取用 ParamArray 参数，并将其放入到一个普通数组变量中。

```
Public Sub TestparamArray(ParamArray Values() As Variant)
Dim StealValues() As Variant
    VBoost.AssignSwap ByVal VarPtrArray(StealValues), _
      ByVal DerefEBP.Call(12)
    'Values is now an empty array and StealValues contains
    'the ParamArray data.
End Sub
```

在一个 BAS 模块中使用 DerefEBP 存在很多缺点。在 BAS 模块中，堆栈要难以计算的多。BAS 模块的参数位置依赖于返回值的类型。一个类模块中的返回值总是一个 HRESULT，它是一个固定的四字节值。但 BAS-module 函数并不返回 HRESULT，因此返回类型的长度是可变的，这将影响到堆栈设计。一个函数返回 eax 寄存器中的第 1~4 个字节数据。第 5~8 个字节则送回到 eax 和 edx 中。为了能够返回大量的字节，调用函数提供一个指向内存地址的指针，被调用的函数应在该地址中写入返回数据。返回值地址作为堆栈中的第一个参数进行传递。因此，如果返回类型的字长大于 8 个字节，第一个可见参数将额外的偏移 4 个字节。

关于在 BAS 模块中使用 DerefEBP 所产生的另一个问题是：VB 在 IDE 环境中将 4 个额外的字节推入栈，并且 pcode 是可执行的。IDE 环境中的堆栈大小与 EXE 中的堆栈大小是不一样的。这样，我们便不可能键入一常数值，使其在 IDE 环境和本地 EXE 中同时有效。我们可以有几种方法来决定一个程序是否能在 IDE 环境中运行，但如果要判断一个可执行程序究竟是 pcode 还是本地 EXE 则困难得多。

```
'Check if we're in the IDE.
Public Function InIDE() As Boolean
    On Error Resume Next
    Debug.Assert 1 \ 0
    InIDE = Err
    On Error GoTo 0
End Function
```

下面的例子考虑了几个标准模块函数，并涉及到最后一个参数的堆栈偏移量的问题。

```
'Start with 8 + 4 byte return value + 8 byte ByVal Double
'Total: 20 in native, 24 in IDE
Function First( ByVal Value As Double, _
    ParamArray PA() As Variant) As Variant
```

```
'Start with 8 + 4 byte ByRef Variant
'Total: 12 in native, 16 in IDE
Sub Second( _
  Value As Variant,ParamArray PA() As Variant
```

```
'Start with 8 + 0 byte return value + 16 byte ByVal Variant
'Total: 24 in native, 28 in IDE
Function Third( _
   ByVal Value As Variant,ParamArray PA() As Variant) As Long
```

```
'Start with 8 + 0 byte return value + 4 byte ByVal Byte
'Total: 12 in native, 16 in IDE
'Note that each parameter takes a minimum of four bytes.
Function Fourth(ByVal OneByte As Byte, _
  ParamArray PA() As Variant) As Currency
```

11.5.2 定位当前函数

Visual Basic 中并不存在一种用来决定当前运行的是哪一个函数的内建机制。然而，我们可以在一个本地可执行程序运行时，通过使用少量汇编代码来决定调用函数中的下一条指令，从而获取这一信息。我们已经看到，当我们调用一个函数时，下一条语句值将被推入堆栈，因此，下一条指令并不难以获得。有两段汇编代码有助于此。其中，第一段返回调用函数的下一条指令。第二段则返回至上一个函数级，用来获取调用了当前函数的函数中的下一条指令，并允许我们从一段 error-logging 程序中取得一条指令的地址。我们可以使用 ICallVoidReturnLong 定义来调用它们。

```
'Selections from GetIP.Bas
'// Code to get the next instruction in the current function.
'// long GetIP()
'// Get the return address from the current stack location.
'mov eax, [esp]
'// 8B 04 24
'ret                 // Return
'// C3
```

```
Private Const cGetIPCode As Long = &HC324048B

'As in DerefEBP.Bas
Public Property Get GetIP() As ICallVoidReturnLong

'As in DerefEBP.Bas
'Selections from GetCallingIP.Bas.
'// Code to get the next instruction in the calling function.
'// Note that this only works if the compiler generated enough
'// code to warrant pushing a base pointer in the calling
'// function, but calling GetCallingIP.Call is sufficient,so
'// this isn't a problem in practice.
'// long GetCallingIP()
'mov eax, [ebp + 4] // Get the return address off the stack.
'// 8B 45 04
'ret           // Return
'// C3

Private Const cGetCallingIPCode As Long = &HC304458B
'As in DerefEBP.Bas
Public Property Get GetCallingIP() As ICallVoidReturnLong
'As in DerefEBP.Bas.
```

```
'Calling code. The second number shown will be a few bytes
'higher than the first.
'This requires a VBoostTypes reference,
'FunctionDelegator.Bas, GetIP. Bas,and GetCallingIP.Bas.
Sub Main()
   MsgBox Hex$(GetIP.Call)
   TestCallingIP
End Sub
```

```
Sub TestCallingIP()
   MsgBox Hex$(GetCallingIP.Call)
End Sub
```

当然，读取当前的指令指针只完成了一半的工作。我们还需要一种用以解释数字的机制。这通常由 Link.exe 所输出的 MAP 文件来完成。VB 中不能提供一个 IDE 选项来产生该文件，不过这并不构成大的障碍。我们可以很容易地修改环境变量设置，以使得 VB 产生一个 MAP 文件。

（1） 关闭 VB IDE。

（2） 设置 LINK 环境变量为/MAP。

（3） 重新启动 VB。

（4） 对工程进行编译。

这样，我们就能产生一个 MAP 文件，可以用该文件来 pin 某个特定函数的 IP 号。如果我们正在跟踪一个 DLL，可能会想要将 App.hInstance 值连同 IP 号一起记录下来，以便统计。关于 MAP 文件和调试等方面的更详细信息，读者可参考 John Robbins 所著的书籍和文章。[1]

11.6 类函数指针

尽管编译器能够产生直接调用友元函数的代码（与通过 VTable 进行调用相反），我们仍然不能像对 BAS 模块中的一个函数一样，直接对一个友元函数使用 AddressOf。VB 中作出这一限制，主要存在两方面的原因：首先，友元函数具有一个隐含的 Me 参数，这就意味着我们必须提供一个指向该带有函数指针的对象的引用。其次，凡是友元函数，都返回隐藏的 HRESULT 值，所以它们也就不具备和一个标准模块中函数相同的函数签名。修改过的函数签名将禁止友元函数加入到一个 API 函数中，因为对于一个 API 函数，初始时已加入 AddressOf。

如果忽略产生上述限制的原因，而我们又想在一个类模块中直接应用回调函数，便会产生阻碍。例如，当我们通过划分一个 Form 以捕获窗口信息时，都希望在 Form 模块中而不是在标准模块中运行代码。一旦我们拥有一个 Form 的引用，从标准模块重新定向到 Form 中的代码是一件很容易的事情。不过，这一引用一般并不提供函数指针。

为了解决这一问题，本书中已经创建了一些被称为 PushParamThunk 的汇编代码。PushParamThunk 代码动态的产生了一个函数，该函数中包含有对目标对象的引用以及一个内建的标准模块函数的地址。我们不能在设计时对该函数进行编码，这是因为只有在运行时才能获取这些值。下面将列出关于 thunk 的一个用法示例，并看一看代码中是如何对其进行应用的。该程序段看起来与一个标准窗口划分代码十分类似。其差别在于该代码段中没有用到窗口句柄来存储指向与窗口相联系类的指针。初始化后的 PushParamThunk 结构是一个函数，该函数带有两个指针，其中一个指向被包含于函数中的类实例；另一个则指向

[1] 参见 John 所著的《Debugging Applicalion from Microsoft Press》。在http://www.jprobbins.com有其他参考文献。

函数。

```
'Code in Form1
Private m_WndProcThunk As PushParamThunk
Private m_WndProcNext As Long
Private Sub Form_Load()
    InitPushParamThunk _
      m_WndProcThunk, ObjPtr(Me),AddressOf ForwardWndProc
    m_WndProcNext =_
      SetWindowLong(hWnd, GWL_WNDPROC, m_WndProcThunk. pfn)
End Sub
Friend Function WndProc( _
  ByVal hWnd As Long, ByVal uMsg As Long, _
  ByVal wParam As Long, ByVal 1Param As Long) As Long
     'Processing
    WndProc = CallWindowProc( _
     m_WndProcNext, hWnd, uMsg, wParam, 1Param)
End Function
'More code
```

```
'Code in a standard module
Public Function ForWardWndProc(ByVal This As Form1,_
  ByVal hWnd As Long, ByVal uMsg As Long, _
  ByVal wParam As Long, ByVal 1param As Long) As Long
    ForwardWndProc = This.WndProc(hWnd, uMsg, wParam,1Param)
End Function
```

正如我们所看到的一样，PushParamThunk 使用起来十分简单。如果我们的函数预定义中具有 4 个参数，那么调用目标回调函数时将采用正确转换后的实参及一个新的首参数，该参数将是我们在 InitPushParam 调用中所指定的值。很清楚，ObjPtr(Me)和 ByVal This As Form1 之间的转换只有一种可能；我们可以使用 PushParamThunk 来将一个定制参数加入到任意函数中。一般来说，我们应使第一个参数为 ByVal 类型。

```
Private Type ThunkBytes
    Thunk(5) As Long
End Type
```

```
Public Type PushParamThunk
   Pfn As Long
   Code As ThunkBytes
End type
Public Sub InitPushParamThunk(Thunk As PushParamThunk, _
   ByVal ParamValue As Long, ByVal pfnDest As Long)
      With Thunk.Code
          .Thunk(0) = &HB82434FF
          .Thunk(1) = ParamValue
          .Thunk(2) = &H4244489
          .Thunk(3) = &HB8909090
          .Thunk(4) = pfnDest
          .Thunk(5) = &H9090E0FF
      End With
      Thunk.pfn = VarPtr(Thunk.Code)
End Sub
'asm code
'push [esp]
'mov eax, 1234h // Dummy value, pushed parameter
'mov [esp + 4], eax
'nop          // nop Adjustment so the next long is aligned
'nop
'nop
'mov eax, 5678h // Dummy value, destination function
'jmp eax
'nop
'nop
```

上面所列出的 PushParamThunk 代码适用于那些返回类型足够小的函数，但对于大的返回类型来说并不适用，这是因为返回类型大于 8 个字节时，要用到额外的堆栈参数。如果我们想让 PushParamThunk 同样适用于那些返回类型为 Variant(16 字节)或其他占用更多字节数的类型，则需要使用汇编代码的修正版本。为了满足这一要求，包含在 PushParamThunk.bas 之中的 PushParamThunkStackReturn 代码需要插入参数值，且该参数作为第二个参数而非第一个参数。如果确实要求有这种能力，那么我们可以简单的使用替代结构和初始化函数。我们还可以使用条件编译值 PUSHPARAM_NONORMAL 或 PUSHPARAM_NOSTACKRETURN 来去除工程中所不需要用到的特性。

11.7 使用 CDECL 函数

由声明语句进行定义的 Visual Basic 函数总是被假定为_stdcall。我们编制的所有函数都将使用该调用协定来产生。然而，在各种各样的使用 alternate_cdecl 调用协定的 DLL 中，包含了许多导出函数。连同本书所提供的代码中包含了一个 CDECLFunctionDelegator.bas 文件，该文件类似于 FunctionDelegator.bas 文件，但该文件调用了 CDECL 函数。本节将考察的是所提供的一些函数，而非汇编代码实现。

在我们对 calling_cdecl 函数的前景感到鼓舞之前，有必要考虑一下调用这些函数所需要的汇编代码的数量。为取得 this 指针，我们必须运行远大于 6 字节的汇编代码。实际上，对于一个小函数来说，函数调用所花费的开销将远大于它本身所用开销。我们应该根据具体情况来决定是否调用_cdecl 函数。

除函数指针之外，cdecl 代理函数同样要求一个 StackSize 值。在一个 cdecl 函数调用中，被调用的函数并不会将参数弹出堆栈。参数弹出的工作留给了调用函数。对于一个 cdecl 代理的情况，调用函数由代理中的 asm 代码来提供，因此代理必须清楚堆栈的大小。其中，两个 cdecl 代理函数分别被称作 InitCDECLDelegator 和 NewCDECLDelegator。它们就像相应的 InitDelegator 和 NewDelegator 函数一样，区别是这两个函数中都包含了一个额外的 StackSize 参数。另外，传递到 InitCDECLDelegator 上的 CDECLFunctionDelegator 除包含了一个 StackSize 成员之外，其他也都类似于 FunctionDelegator。

使用这两个函数，我们可以调用一个 GetProcAddress-retrieved cdecl 函数指针。然而，这只完成了一半的工作，这是因为我们不能直接使用 AddressOF 来提供一个 cdecl 回调函数。而 InitCDECLThunk 函数则解决了有关 cdecl 问题的另一半。InitCDECLThunk 取得了一个 CDECLThunk 结构、一个函数指针及堆栈长度。在函数返回之后，在 CDECLThunk 变量的 pfn 域中容纳了函数指针。如果我们想由另一个不同的函数指针和堆栈大小来重用 CDECLThunk 结构，我们可以使用 UpdateCDECLThunk 函数来更新代码。有了这些可以支配的函数，我们已经作好了调用一个 cdecl 函数和提供一个 cdecl 回调函数的所有准备。我们可以使用与先前章节同样的回调函数，这些回调函数用到了 VB 中定义的 QuickSort。不过不同的是，这一次我们让 msvcrt.Dll 中的 qsort 函数来完成全部的工作。

由于 qsort 是一个函数指针，我们需要一个类型库定义来使得编译器能够调用 cdecl 代理。下面便是使用这种函数的 ODL，它被描述为 stdcall 而非 cdecl。CRTQuickSort 函数使用了 InitCDECLDelegator 和 InitCDECLThunk，因此该函数不需要用到由堆分配的内存。

```
[uuid(C9750742-4659-11d3-AB5C-D41203C10000), odl]
interface ICallQSort : IUnknown
{
  void Qsort([in] long pBase,[in] long Number,
```

```
            [in] long Size, [in] long pfCompare)
}
Public Sub CRTQuickSort( _
  ByVal pBase As Long, _
  ByVal Number As Long, _
  ByVal Size As Long, _
  ByVal pfCompare As Long)
Dim pQSort As ICallQSort
Dim CompareThunk As CDECLThunk
Dim Delegator As CDECLFunctionDelegator
Dim pCRT As Long
Dim pfnQSort As Long
    pCRT = LoadLibrary("msvsrt.Dll")
    pfnQSort = GetProcAddress(pCET,"qsort")

    '4 long parameters = 16 byte stack size
    Set pQSort = InitCDECLDelegator( _
    Delegator, pfnQSort,16)

   '2 long parameters = 8 byte stack size
    InitCDECLThunk CompareThunk, pfCompare, 8
    pQSort.QSort pBase, Number, Size, CompareThunk.pfn

    EreeLibrary pCRT
End Sub
```

如果我们打算在实际代码中使用该函数，很可能会将其封装在一个类中，从而隐藏 msvcrt.Dll 的模块句柄和 qsort 的地址。注意：如果我们将 VB 代码编译成快速代码，并禁用数组边界检查，VB QuickSort 执行起来要比 CRTQuickSort 版本快 35%，这是因为后者包含了 cdecl thunking 的开销。当然，如果 CRTQuickSort（或者是下一个想要使用的 cdecl 函数）足够的快，那么我们可以在 VB 中完全跳过编制一个替代函数的工作。好了，将其打印下来，怎么样？

第十二章

重 载 函 数

Visual Basic 中的 Implement 关键字本身支持基本接口编程，在整本书中我们都着重讨论了接口的设计方法，因为它为抽象的类和调用代码重用提供了良好的模型。VBoost 聚合器对象同时还可以实现接口方法的完全重用。

在第 5 章“抽象类中的方法”中，我们讨论了所有基于继承的系统中所存在的一些缺陷。但是，这并不意味着继承性作为一种编程模型是毫无价值的。当然，接口实现能够解决大多数问题，但是有时候我们只是想重载某一具体函数。

用来绘制某种图形的函数可以作为说明继承性函数的很好的例子。在第 5 章“一个类中的多态性”部分中给出了一个 Draw 接口的简单实现，该实现中使用一些属性来决定是画直线、矩形还是椭圆。对于这种情况来说，采用公共属性十分恰当和快捷，因为这三种情况都需要同样的数据。如果我们想使用同样的接口来在矩形中绘制更为复杂的对象，该怎么做呢？调用代码仍然能够调用 IDraw.Draw 方法，但这时，Draw 函数应被映射到一个包含更多数据的对象中的该函数另外一个不同的实现上。

采用继承模型具有两个主要优点。第一，用户可以和一个基类无缝的共享数据。第二，用户可以对虚函数进行重载以修正基类特性。我们前面已经学过怎样通过在共享内存中使用数组来共享数据，在这一章，我们将了解到将函数调用重定向到一个继承类代码中的三种方法。函数的重定向将主要依赖于我们已经有所了解的技术。本章中使用了函数指针代理、类函数重定向、盲 VTable 封装器、聚合对象、轻度对象以及直接共享内存等等。

在本章中的三个例子中都涉及到同一个问题，即对基类上的一个返回字符串的公有函数进行重定向。每种技术都是在同样的类中完成的——分别被称为基类和继承类，并且都使用了一个被称为 ModDerived 的帮助模块。用户可以比较一下这些技术的优点和实现所花费的代价，从而决定将使用哪一种。我们可以采用下面的代码来对这三个样例进行测试。

```
Dim Derived As New Derived
Dim Base As Base
    Set Base = Derived
    MsgBox Base.OverrideMe
```

12.1 协作重定向

第一种技术被称为协作重定向，因为它主要使用基类函数。原始函数被调用，然后再将其重定向到一个由继承函数所提供的函数指针上，很明显，该技术将依赖于基类在重定向中的协作。虽然这种技术是最安全并且最简单的一种技术，但它却远不如后面两种方案来的精巧别致。

协作重定向的策略比较简单。即基类和继承类共享一个带有函数指针的结构体，在继承类中，设置了一个函数指针以及一个用来放置用户自定义数据（通常为继承类中的ObjPtr）的域，协作函数然后调用 Agreed-upon 函数指针，该函数指针通常看起来就像是带有一个额外的数据参数的重载函数。

程序清单12.1 采用共享内存和一个函数指针重定向一个函数访问基类。本例子可在光盘中 Sample\Function Verrides\cooperative Redirection 目录下找到

```
'modDerived.bas, type definitions and a callback function.
'requires: ArrayOwner.Bas or ArrayOwnerIgnoreOnly.Bas,
'       PushParamThunk.Bas
Public Type BaseData
   OverrideMeThunk As PushParamThunk
End Type
Public Type OwnedBaseData
   Owner As ArrayOwner
   Data() As BaseData
End Type
Public Function Override(ByVal This As Derived) As String
   Override = This.Base_OverrideMe
End Function
```

```
'Base.cls
Private m_Data As BaseData
Friend Function DataPtr() As Long
   DataPtr = VarPtr(m_data)
End Function
Public Function OverrideMe() As String
Dim FD As FunctionDelegator
```

```
Dim pCall As ICallVoidReturnString
    With m_Data.OverrideMeThunk
        Set pCall = InitDelegator(FD, .pfn)
        OverrideMe = pCall.Call()
    End With
End Function
```

```
'Derived.cls
Private Type InitData
    AggData(0) As AggregateData
    IIDs() As VBGUID
End Type
```

```
Private m_Data As modDerived.OwnedBaseData
Private m_Hook As UnknownHook
Private m_Base As Base
Private Sub Class_Initialize()
Dim InitData As InitData
    Set m_Base = New Base
    InitArrayOwner m_Data.Owner,LenB(m_Data.Data(0)), 0, False
    m_Data.Owner.SA.pvData = m_Base.DataPtr
    With m_Data.Data(0)
        InitPushParamThunk. OverrideMeThunk, _
            ObjPtr(Me), AddressOf modDerived.Override
    End With

    With initData
        With .AggData(0)
            Set .pObject = m_Base
            .Flags = adIgnoredIIDs
        End With
        VBoost.AggregateUnknown Me, .AggData, .IIDs, m-Hook
    End With
End Sub

Friend Function Base_OverrideMe() As String
```

```
    Base_OverrideMe = "Base.OverrideMe redirected to Derived"
End Function
```

```
'ODL interface prototype for ICallVoidReturnString
[
    uuid(795984A1-928C-11d3-BBDD-D41203C10000),
    odl
]
interface ICallVoidReturnString: IUnknown
{
    Bstr Call();
}
```

对于重定向一个函数来说，协作重定向是一个相对安全的系统，但是该系统却不具有我们在继承模型中通常所期望的许多特征。例如，基类不得不始终参与便是其中一个缺点。另外，基类是在调用完重定向函数之后返回而不是在运行完它本身代码之后返回，这一决定的作出也是应当留给继承类来处理的一个部分。总之，这种技术很有效，但是需要用到基类中许多额外的代码和数据。

12.2 接口封装

协作重定向技术将大量的职责放到了基类中，通过在基类的周围进行一个附加的封装，我们便可以完全去除所有对基类的要求。并具有使用更少的基类代码以及保持重载函数基类实现的完整性等优点。我们可以从重载函数中调用原始函数，因为与协作重定向方式不同，不需要改变原始函数以支持重载。封装一个对象的接口的步骤很容易理解，但是必须要注意确保每一步的正确性。接口封装的基本思想便是在所要封装接口的一个实例周围分发一个盲 VTable 封装器，然后再将封装好的对象传递给任何一个对重载接口的外部请求。要对一个函数进行重载，只需要简单的将所需要的所有函数从由 VBoost 所提供的盲代理 VTable 中拷贝到另一个 VTable 数组中就可以了。参见图 12.1。

要想成功的对接口进行重载和替换函数，用户必须注意两个问题：第一，必须使得盲代理封装器非常的小，以至于被封装的对象几乎没有意识到它被封装了。如果让被封装接口的 QueryInterface 调用能够访问到所被封装的对象，那么我们所得到的将是原始的 VTable 而不是重载后的函数。为了将一个封装好的接口聚合到对象中，我们必须为接口指定一个 IID，并设置一个 adFully Resolved 标志来防止 VBoost 对重载接口的封装对象进行质询。如果基类中实现了我们原本想放在继承对象中的额外接口，那么我们需要在基对象中的封装

接口之后再添加一个盲聚合 AggregateData 入口。第二个要注意的问题是：传递到 AggregateUnknown 的任何对象都不应持有对一个主对象中的接口的引用（当然，IDelayCreation 引用是一个例外）。在下面所列出的 Baseoverride 结构中，pDerived 是一个长整型值，并且该值在控制 IUnknown 上持有一个弱引用。一个强引用将会导致循环引用，这是因为主对象中拥有 UnknownHook 的缘故。UnknownHook 依次拥有在 BaseOverride 结构中所建立的对象，并且依次引用了控制 IUnknown 上的接口。

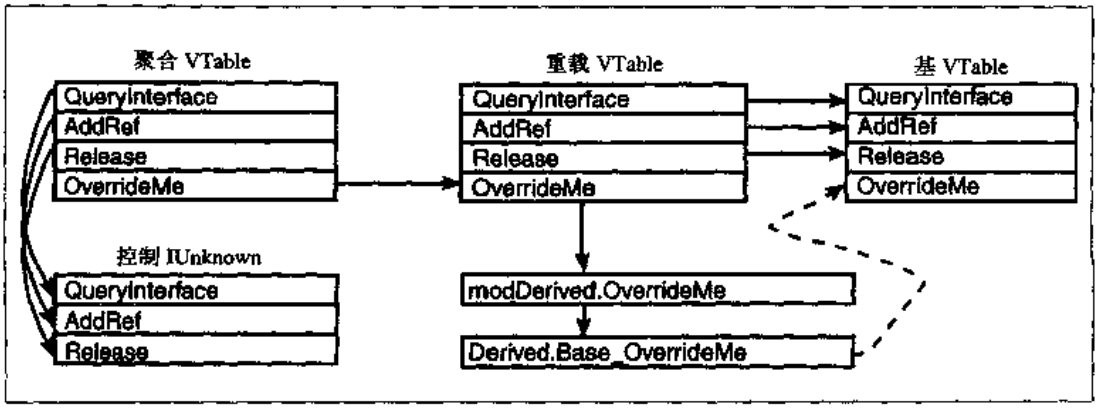

图 12.1 一个由聚合对象所提供的接口中的重载函数

程序清单 12.2 封装一个无基类协作的接口。参看 Samples\FunctionOverrides\DoubleWrap 目录下的程序样例

```
'modDerived.Bas
Private Const cVTableSize As Long = 8
Private Type WrapVTable
    VTable(cVTableSize - 1) As Long
End Type
Private m_VTable As WrapVTable
Private m_pVTable As Long

Public type BaseOverride
   BD As BlindDelegator
   pDerived As Long
End Type

'Construct a VTable using VBoost's BlindDelegator VTable.
Public Sub HookVTable(pBDVTable As Long)
    If m_pVTable = 0 Then
```

```
        With m_VTable
            CopyMemory .VTable(0), ByVal pBDVTable, _
              4 * cVTableSize
            .VTable(7) = FuncAddr(AddressOf OverrideMe)
            m_pVTable = Varptr(.VTable(0))
        End With
    End If
    PBDVTable = m_pVTable
End Sub
Private Function OverrideMe( _
  This As BaseOverride, retVal As String) As Long
Dim Derived As Derived
    'Make sure [out] param is 0.
    VBoost.AssignZero retVal

    On Error Goto Error
    'jump to friend function in derived class.
    VBoost.AssignAddRef Derived, This.pDerived
    retVal = Derived.Base_OverrideMe
    Exit Function
Error:
    OverrideMe = mapError
End Function
```

```
'Base.cls
Public Function OverrideMe() As String
   OverrideMe = "Base.OverrideMe"
End Function
```

```
'Derived.cls
Private Type InitData
   AggData(0) As AggregateData
   IIDs(0) As VBGUID
End Type

Private m_Hook As UnknownHook
```

```
Private m_BaseOverride As BaseOverride
Private m_Base As Base

Private Sub Class_Initialize()
Dim InitData As InitData
Dim pBaseWrapped As IUnknown
Dim LastIId As LastIID

    'Get the IID for the base class.
    'This is required so that we can
    'set adFullyResolved on this IID.
    InitLastIID LastIID
    On Error Resume Next
    Set m_Base = LastIID.QIThis
    On Error GoTo 0

    Set m_Base = New Base

    With m_BaseOverride
        Set pBaseWrapped = VBoost.CreateDelegator( _
          m_Base, m_Base, ,VarPtr(.BD))
        modDerived.HookVTable .BD.pVTable
        .pDerived = Objptr(Me)
    End With

    With initData
        With .AggData(0)
            .Flags = adUseIIDs Or adFullyResolved
            .FirstIID = 0
            Set .pObject = pBaseWrapped
        End With
        .IIDs(0) = LastIID.IID
        VBoost.AggregateUnknown Me, .AggData, .IIDs, m_Hook
    End With
End Sub
```

```
Friend Function Base_OverrideMe() As String
    Base_OverrideMe = m_Base.Overrideme & _
      "overridden by Derived"
End Function
```

如果我们使用另一个类中来封装引用的话，那么预先使用 CreateDelegator 来构造聚合器并没有任何副作用。但是如果我们想要封装一个直接在类中实现的接口，就不能使用这个函数了。这是因为 CreateDelegator 函数在 PunkOuter 对象以及 PunkDelegatee 对象上均持有强引用。我们仍然可以使用一个 BlindDelegator 结构，但是不能用 CreateDelegator 来对其进行初始化。

为了将弱引用放入到一个 BlindDelegator 结构中，我们必须要做的有两件事。首先，必须找到 VTable，这通常可以通过将特定值-1 传递给 VBoost.BlindFunctionPointer 函数来得到。另外，我们还必须注意析构函数的回调。如果设置了 pfnDestroy 区域，Blind Delegator VTable 中的释放函数就能够获知它没有分配 BlindDelegator 结构。pfnDestory 值为-1，则说明在释放函数中不需要做任何事。但是如果是其他非零值，则说明应该调用析构函数来清除和释放结构。

析构函数中传递了 BlindDelegator 结构。在这个结构之后可能有也可能没有其他数据项，有一点需要保证的便是 punkOuter 和 punkInner 域为零值。这些值被传递到了局部变量并作为 Byref 型参数传递给 pfnDestory 函数。而且，这些引用都将在 pfnDestory 完成之后被释放掉——除非我们自己来释放它们或者将它们清零。如果使得 pOuter 参数为零值，函数将返回一个从 Release 调用中返回的值。不管我们选择如何来处理弱引用，在析构函数中，我们都可以自己来选择是将弱引用清零还是将它们转变成强引用。修改后的封装了一个控制对象的本地接口代码看起来如下所示。

程序清单 12.3　使用一个定制 BlindDelegator 结构来重载一个在控制 IUnknown 上直接实现的接口。我们可以在光盘的 Samples\FunctionOverrides\DoubleWrapSelf 目录下找到该代码

```
'Code in a BAS module to override an interface on the
'controlling object.
Public Type Selfoverride
   BD As BlindDelegator
End Type

Public Function InitSelfOverride( _
  Struct As SelfOverride,ByVal pSelf As Long) As IUnknown
Dim pUnk As IUnknownUnrestricted
```

```
    If m_pVTable = 0 Then
      With m_VTable
        CopyMemory .VTable(0), _
          ByVal VBoost.BlindFunctionpointer(-1), _
          4 * cVTableSize
        .VTable(7) = FuncAddr(AddressOf OverrideMe)
        m_pVTable = VarPtr(.VTable(0))
   End With
    End If
    With Struct.BD
       VBoost.Assign .pInner, pSelf
       'pUnk is IUnknownUnrestricted so that this code
       'generates a QueryInterface
       Set punk = .pInner
       VBoost.Assign .pOuter, pUnk
       .cRdfs = 1
       .pfnDestroy = FuncAddr(AddressOf DestructSelfOverride)
       .pVTable = m_pVTable
       VBoost.Assign InitOverride, VarPtr(.pVTable)
    End With
End Functiong
Private Function DestructSelfoverride( _
  This As SelfOverride, pInner As IUnknown, _
  pOuter As IUnknownUnrestricted) As Long
    VBoost.AssignZero pInner
    'Leave a reference in pOuter so that VBoost can
    'read the return value from the Release call.
    pOuter .AddRef
End Function
Private Function OverrideMe( _
  This As Selfoverride, retVal As String) As Long
Dim pSelf As SelfOverride
    'Make sure [out] param is NULL.
    VBoost.AssignZero retVal

    On Error GoTo Error
```

```
    'Jump to a Friend function in this class.
    VBoost AssignAddRef pSelf, This.BD.pInner
     retVal = pSelf.Derived_OverrideMe
     Exit Function
Error:
    OverrideMe = MapError
End Function
```

```
'Calling code snippet; callback functions remain as before
Private Sub Class_Initialize()
Dim InitData As InitData
Dim LastIID As LastIID
Dim pObjImpl As IImplemented

    'Get the IID for the base class.
    InitLastIID LastIID
    On Error Resume Next
    Set pObjImpl = LastIID.QIThis
    On Error GoTo 0

    With InitData
        Set pObjImpl = Me
        With .Aggdata(0)
            .Flags = adUseIIDs Or adFullyResolved Or _
                adBeforeHookedUnknown
            .FirstIID = 0
           Set .pObject = modSelfOverride.InitSelfOverride( _
            m_SelfOverride, ObjPtr(pObjImpl))
        End With
        .IIDs(0) = LastIID.IID
        VBoost .AggregateUnknown Me, .AggData, .IIDs, m_Hook
    End With
End Sub
```

调用代码中最大的不同便是现在的标志中包含了 adBeforeHookedUnKnown 标志，如果让 Query Interface 序列依据 IID 来读取主对象，我们将不会看到重载，这是因为聚合体总是

与能够对所请求接口进行解析的第一个对象在一起。稍后我们将看到，如果我们仅仅是一直等到绝对有必要时才去填充 BlindDelegator 域，那么该机制便要容易使用得多。

12.3 瘦接口封装

该代码的第一个封装版本最终使用了目标接口实现周围的两个 VTable 代理。在最后一部分，CreateDelegator 为 punkOuter 和 punkDelegatee 参数调用指定 m_Base。因此基对象对于被封装的对象来说既是控制 IUnknown 又是代理。在这种情况下，读者可能会试图简单的将 Derived 指定为控制 IUnknown。然而，这样会导致产生一个循环引用，因此聚合器便不能拥有对象。如果我们想去除一个封装的话，便需要在一个将要到来的 QI 调用期间返回封装接口，但用户不能用聚合器对象来存储封装对象。如图 12.2。

AggregateData 结构提供了足够的方法来使函数能够返回被封装好的接口而不需要进行不明确的存储。这里，我们需要额外进行设置的标志是 adDelayDontCacheResolved 和 adNoDelegator。我们可以一直延迟到 IDelayCreation_Create 时再对 BlindDelegator 结构进行填充。在程序清单中，我们应该着重注意在 IDelayCreation_Create 和 DestructBaseOverride 函数中的有关实例检验代码。我们可以避免对 Derived 的弱引用，通常在一个聚合器通过在 CreateDelegator 中指定一个析构函数来拥有封装接口时，需要要求这种弱引用。这时，正如我们在最后一个例子中所看到的一样，析构函数被用来清除定制化数据，而不是为了用来使能 pOuter 和 pInner 域中的弱引用。如例 12.4 所示。

程序清单 12.4 所列出的代码中给出了继承类中的一个 IDelayCreation。如果我们想要减少继承类中的代码数量的话，也可以使用一个轻度对象来实现 IDelayCreation 接口。读者可以参见第 16 章中“无窗口控件”部分所用到的 InitInPlaceObjectWindowlessHook.Bas 下的代码，该代码便是使用了该技术的一个范例。

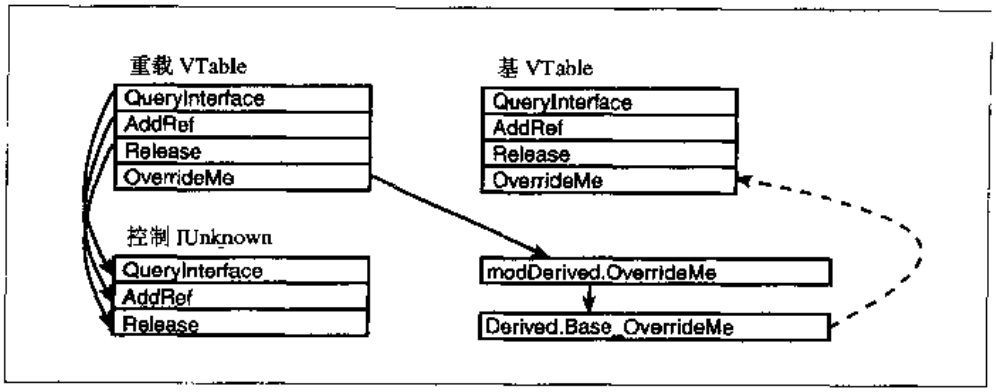

图 12.2 通过使用 BlindDelegator 结构和延迟创建来封装一个带有单个盲 VTable 代理的接口

程序清单 12.4 封装一个带有单个盲代理的接口。参见光盘 Samples\FunctionOverrides\SingleWrap 目录下的程序

```
'modDerived.Bas
Private Const cVTableSize As Long = 8
Private Type WrapVTable
  VTable(cVTableSize - 1) As Long
End Type
Private m_VTable As WrapVTable
Private m_pVTable As Long

Public Type BaseOverride
  BD As BlindDelegator
  Derived As Derived
End Type

'Construct a VTable using VBoost's BlindDelegator vtable.
Public Sub HookVTable(pBDVTable As Long)
  If m_pVTable = 0 Then
    With m_VTable
      CopyMemory .VTable(0), ByVal pBDVTable, _
        4 * cVTableSize
      .VTable(7) = FuncAddr(AddressOf OverrideMe)
      m_pVTable = VarPtr(.VTable(0))
    End With
  End If
  pBDVTable = m_pVTable
End Sub
Pulic Function DestructBaseOverride(This As BaseOverride, _
  pInner As IUnknown, pOuter As IUnknown) As Long
    'clear Derived So that we don't leak.
    Set This.Derived= Nothing
End Sub
Private Function OverrideMe(This As BaseOverride, _
 retVal As String) As Long
```

```
    'Make sure [out] param is 0.
    VBoost.AssignZero retVal

    On Error GoTo Error
    'Jump to Friend function in derived class.
    retVal = This.Derived.Base_OverrideMe
  Exit Function
Error:
    OverridiMe=MapError
End Function
```

```
'Base.cls
Public Function OverrideMe() As String
  OverrideMe="Base.OverrideMe"
End Function
```

```
'Derived.cls
Implements IDelayCreation

Private Type InitData
  AggData(0) As AggregateData
  IIDs(0) As VBGUID
End Type

Private m_Hook As UnknownHook
Private m_BaseOverride As BaseOverride
Private m_Base As Base

Private Sub Class_Initialize()
Dim InitData As InitData
Dim LastIID As LastIID

   'Get the IID for the base class.
   InitLastIID LastIID
   On Error Resume Next
   Set m_Base =LastIID.QIThis
```

```
    On Error GoTo 0

    Set m_Base=New Base
    With InitData
        With .AggData(0)
            .Flags=adUseIIDs Or adNoDelegator Or _
             adDelayDontCacheResolved Or adFullyResolved
            FirstIID=0
            Set .pObject = Me
    End With
       .IIDs(0)=LastIID.IID
       VBoost .AggregateUnknown Me, .AggData, .IIDs, m_Hook
    End With
End Sub

Private Function IDelayCreation_Create_
  (IID As VBoostTypes .VBGUID) As stdole.IUnknown
    'We have specified only one IID,so we don't
    'have to check if we got the right one.
    'If there is more than one delayed IID.
    'store the IIDs at module level.
    With m_BaseOverride
        If  .BD.cRefs Then
            'Return the current item.
            VBoost AssignAddRef _
              IDelayCreation_Create,Varptr(.BD)
        Else
            Set IDelayCreation_Create = _
              VBoost .CreateDelegator(_
              Me,m_Base,,VarPtr(.BD), _
              AddressOf modDerived.DestructBaseOverride)
            Set .Derived = Me
            modDerived.HookVTable .BD.pVTable
        End If
    End With
End Function
```

```
Friend Function Base_OverrideMe() As String
   Base_OverrideMe = m_Base.OverrideMe & _
    " overridden by Derived"
End Function
```

12.3.1 封装我们自己的接口

在我们自己的接口中使用延迟创建来将一个封装器放在接口上要比使用定制创建和盲代理结构的析构代码容易的多。实际上，为了使用延迟创建代码来封装我们自己的接口，只需要在其他四个标志中添加一个 adBeforeHookedUnknown 标志（adUseIIDs，adNoDelegator，adDelayDontCacheResolved，adFullyResolved）。由于封装器在拆分时释放了所有的引用，并且没有被隐藏，所以我们也就无需担心在无延迟封装中可能遇到的有关循环引用的问题。

12.4 封装中的一些问题

还有一些与重载函数相联系的一些复杂问题，尽管代码大部分都可以通过剪切和粘贴所得到，但仍然存在相当多的问题。我们还必须得到正确的 VTable 入口，否则将导致程序莫名其妙的产生死机。VB 对每一个公共对象或 Variant 型模块变量使用了三个 VTable 入口，为每一个公共程序添加了一个，并从文件的顶部往下计数。然而，如果在一定的位置与二进制兼容，那么所有在 VTable 上所下的赌注都将落空。找到 VTable 的布局的最简单的方法便是使用 TLI（类型库信息）来向对象查询其 VTable。其方法是：往 TLI（TypeLib Informetion）中添加一个索引，然后每次遇到一个类中的断点时便在立即窗口中运行下面的代码来以列出 VTable 的偏移量。

```
for each mem in _
    TLI.InterfaceInfoFromObject(me).VTableInterface.Members: _
  ?mem.Name,mem.Invokekind,mem.VTableOffaet\4: next
```

在真正的继承中，从基类中调用一个重载了的函数叫做重载。当一个接口封装器位于正确的位置时，基类将完全不受封装的影响。如果想要从基类中调用当前的重载，我们便需要让 Derived 为继承对象传递一个指针到控制 IUnknown 中。并且我们必须存储一个对控制 IUnknown 的弱引用来避免循环依赖。基本的代码看起来如下所示。

```
Private m_pOuter As Long
  Public Function OverrideMe() As String
     OverrideMe = "Base.OverrideMe"
  End Function
  Friend Function SetOuter(ByVal pOuter As IUnknown)
     m_pOuter = Objptr(pOuter)
  End Function
  Private Function GetOuter() As IUnknown
     VBoost .AssignAddRef GetOuter, m_pOuter
  End Function

  'Code in Base to call wrapped OverrideMe.
  Dim pWrapped As Base
     Set pWrapped = GetOuter
     Debug.Print pWrapped.OverrideMe

  'Code in Derived,called before or after aggregation.
  m_Base.Setouter Me
```

在进行封装时，我们将总是想使用带有多个 Derived 版本的基类，如果 VTable 重载指针仅仅指向一种类型的继承类的话，那么对于从基类继承过来的每个类，我们都需要一个完全分开的 VTable。如果我们想要同一个友元函数进行交互，则每个继承类也都需要一个单独的 BAS 模块函数。不过在这里，完全的重做 VTable 也显得过于极端化。为了避免这样，就需要综合应用前面所讲述到的封装技术以及重定向技术。我们可以采用 PushParamThunk 来作为 BaseOverride 结构的一部分并采用一个 FunctionDelegator 对象来对 PushParamThunk 中所指定的函数进行重定向。在光盘中 Samples/FunctionOverrides 目录下的 DoubloeWrapRedirect 和 SingleWrapRedirect 里分别就单重封装接口和双重封装接口两种情况对该方法进行了论证。

第十三章

VB 中的线程

Visual Basic 支持多线程 EXE、DLL、OCX 的创建。读者可能对这句看似简单的话有多种理解。支持多线程有下面的含义：在同一进程中同时运行多个线程的代码的能力、调用在不同线程内运行的代码的能力、同时对一组数据执行多个线程的能力。VB 支持代码执行时的多线程，但它并不固有的支持自由线程，即对同一组数据运行多个线程的能力。如果 VB 支持自由线程。它就能调用别的组件。

Visual Basic 线程是在 COM 称之为单元（apartment）的地方创建的。在 ActiveX DLL 或 OCX 的 Project|Properties 对话框中，可以设置单线程或单元线程的线程类型。这一选项的名字不太合适，因为单线程也使用单元线程。它应精确的命名为“单线程单元”和“多重单线程单元”。

单元这一术语对它本身要表达的概念是很具描述性的。可以将代码运行的区域想像为房间的一个小单元。在单线程（STA）的情况下，在房内可以碰房内的任何东西，墙壁阻止了任何人不经过房屋的前门进入房间。除了房门的限制外，单元中的座位也是很有限的，它一次只能容纳一个客人。所有要进入该房间的客人必须在门外列队站好，等房内的客人离开后才能进入。在后面将介绍“破门而入”的情况，但即使在那种情形下，客人也必须由前门进入房间。

这一比喻在多线程单元（MTA）的情况下就不太明显了。多线程单元没有房门，墙壁只是用木条扎成的栅栏，客人可从任意方向进入房间。客人们来去自由，单元必须能同时向它们提供服务。很明显，在 MTA 中要避免混乱需要做许多计划。为了维持表面的秩序，必须确保客人按一定的顺序使用房间的器件。核心的一点是，应让客人在房内排成短队，而不是让他们在房外等待。在 MTA 中，应提供 STA 中的房门能提供的同步代码。每一进程至多有一个 MTA。

我下面要回到更具体的定义。对于 STA 中的对象，COM 只允许在对象创建的线程内对其做函数调用。STA 中的代码通常在同一线程内运行。COM 提供了一个排队(marshaller)，它可将所有来自外部线程的调用转换为对本地线程的调用。通过在一部分代码运行完成之

前阻塞下一个函数调用，排队也能提供同步功能。为 STA 编写的代码与为一个单线程的引用程序编写的代码相同：不需要明显的同步代码。从线程等待机会进入的角度来说，有一个排队的队列可能成为一种长时间的煎熬。

在 MTA 中，可在任意线程中调用函数，COM 不提供任何内在的同步机制。用户必须提供它自己同步机制，这通常是通过象事件、互斥体（mutexes）、关键段（critical section）这样的内核同步对象来实现。但同步带来的性能上的好处是很明显的。有很少工作要做的调用线程不会一直呆在门外，等有大量工作要做的线程离开后才进入房间。

Visual Basic 支持调用 MTA 和 STA，但是它仅仅支持创建 STA。在一个单线程 DLL 中，VB 在一个单线程上创建所有的对象，所有来自外部线程的调用必须汇集到单个 STA 中去。在多线程 DLL（在 Project|Properties 对话框中叫做“单元线程”）中，VB 只是加入到一个现存的线程中去。所有的 VB DLL 都使用 STA 模型；多线程 VB DLL 支持在多个 STA 中创建对象。不应将对多个 STA 线程的支持同对在 MTA 中创建的对象的支持混为一谈。

13.1 线程中的局部存储

Visual Basic 运行时间在单元线程内是安全的，但在一个自由线程的环境（MTA）中并不安全。实际上，单元线程安全是很容易实现的。VB 给予每一个组件它自己的一套数据。由于在组件之间没有数据共享，就不会发生不同的线程同时访问同一数据时造成的冲突。在 STA 环境中，由于 STA 中的对象只在创建它的线程内运行，所以使用局部线程数据而不是局部进程数据可以确保数据完整性。线程局部变量也使跨线程中的交互不可能存在。每个线程有它自己的一组数据，因此线程没有必要列队等待（即同步）来读取局部数据。

为了给每一线程创建独立的全局数据模块，VB 利用了 Win32 提供的线程局部存储（TLS）APIs:TlsAlloc、TlsAlloc、TlsGetValue、TlsSetValue 和 TlsFree。借助于 TLS，每一线程为它自己的特定线程数据分配空间并利用 TLS 索引存储、检索这一空间。调用 TlsAlloc 可获取全局进程索引，这是一个放置特定线程数据的数据槽（slot）。每一线程使用 TlsSetValue 和 TlsGetValue 来对全局数据槽进行设置并获取自己线程的数据。很明显，在 TlsGetValue 返回有效值以前，必须设置每一个线程的 TlsSetValue。

VB 执行时间依赖于 TLS 来支持单元线程。TLS 数据槽可用来存储全局线程数据和特定工程数据。没有了 TLS，执行时间通常会运行不好甚至崩溃。除了消失运行时间能力外，同时失效的还有捕捉错误、文件 I/O、Set 语句、函数调用、声明函数、固定长度的数组等能力。实际上，没有了 TLS，能做的事情就是在类型库中定义的 API 调用。在“在 DLL 中创建工作线程”一节中将会看到，没有了 TLS 的支持，运行有限数量的代码都是很困难的。没有 TLS 支持运行大量的代码可被称为“注定要失败的挣扎”，可对它不予考虑。

缺少了 TLS，VB 将“步履艰难”，所以了解 VB 何时调用 TlsSetValue 来初始化它的线程和工程数据是很重要的。当第一个对象在给定的线程中创建时，VB 分配并初始化 TLS

数据；当最后一个对象销毁时，释放 TLS 数据。有利的一点是对于一般的 VB 代码，读者无需担心这一点，因为惟一的进入 VB 的入口是通过 ActiveX 对象创建（object-creation）和 Sub Main(这是一个特殊的入口)。通常，VB 代码不在一个没有对象的线程上运行，这意味着 TLS 数据槽通常都有有效的数据。

但是 VB 怎样处理缺少 TLS 时运行的情况呢？这一问题有一个很容易的答案：使用 AddressOf。操作符 AddressOf 可用来传送任何函数指针。有几个 API 调用可以告诉 Windows 使用一个函数指针作为任意线程的调用返回点。例如，在建立新线程、系统或进程范围的键盘钩入和 NT 服务时，就存在这种情况。注意到可暴露进入点的 DLL 访问技术可使 VB 处于这种情况。下面简单的介绍一下如何使 VB DLL 进入一种状态，在这种状态中，从任意函数进入，都能运行 VB 代码。

缺少 TLS 的支持在线程上的 VB DLL 中调用一个函数时，代码只能做算术运算和在类型库中定义的 API 调用。API 调用支持读者使用 COM API 函数来创建 VB 对象。第一个调用为 CoInitialize，它可以建立或加入 COM STA，然后引发 Co* API 调用。接着，可以在同一 VB DLL 中对 CLSID 使用 CoCreateInstance 来创建一个 VB 对象，这就初始化了 VB slots。在这一点上，运行时的一切函数都工作正常。我曾经发现过在直接函数（immediate function）中，MsgBox、InputBox、Appobject 对象和其他的一些语言特征不能正常工作，但在被调用的函数中，这一切皆运行良好。这意味着读者应紧接着 CoCreateInstance 调用一个帮助函数。在释放第一个对象之前，线程的状态一直保持有效。处理完成之后，释放显式创建的对象（和任何由 COM 分配的局部变量），然后调用 CoUninitialize 来平衡 CoInitialize 调用。

在这些步骤中存在的一个问题是执行这些步骤的时间开销使得调用返回函数执行起来特别缓慢。如果经常调用这一函数（如对键盘钩入那样），就要不断的重新初始化线程上的 STA。STA 初始化为同步了的 STA 消息队列创建了一个背景窗口，并且它还分配和初始化 VB 的 TLS 数据槽中的运行时间数据和全局数据。因为每一个击键动作都要执行这些代码花费的时间太多，特别是 VB 线程确实没有初始化时更是如此。（如果 COM 和（或）VB 已经在线程上初始化，CoInitialize/CoCreateInstance/CoUninitialize 执行起来是十分迅速的。）通常，只有在线程存活的时间比较长，这些时间花费才值得，例如 NT 服务或以后要介绍的显式的 DLL 线程创建技术。

13.2 能否避免排队开销

在 STA 的世界中，可以很容易的打破这一规则。毕竟，要处理的对象都在同一进程中，因此读者不用担心 GPF，可以访问所有的指针。可以使用 ObjPtr 来传递指向另一线程里面对象的指针，该指针假设为长整型指针。使用一对 CopyMemory/VBoost 调用，可返回给对象一个直接指针。由于两个线程都初始化了 TLS，现在可以直接调用对象的方法而不需排队开销。

这就直接违反了 STA 规则。绕过排队层，读者就可在除创建这个对象的线程之外的线程内运行该对象的代码。这样做除了破坏了 COM 规则之外，从同步的角度来说，这也是十分危险的，因为读者不经过同步就在两个线程里同时访问类成员。读者可能在错误的线程里使用工程和运行时间全局变量，这可导致大量的错误。所以，尽量不要使用这种方法。尽管从性能的角度来说使用这种方法比较诱人，由此产生的麻烦却更多。欺骗线程模型也违背了本书的一个基本原则:使用 VB 和 COM 来工作，而不是违背它们。

13.3 线程化或非线程化

对于那些未见识过线程的人来说，《线程的领域》中介绍的线程性能是十分理想的。这本小册子竭力说明利用线程可取得很高的性能，但实际上线程并不高效，它带来的许多性能上的改善都只是一种虚幻的假象，这些在这本书中没有提及。线程可使应用程序响应良好，给人一种高性能的印象。在一个单用户应用程序中，反映良好意味着用户可不停的键入和点击。对于一个运行在服务器上、服务多个用户的应用程序来说，反映良好意味着每一个用户都可在近似相等的时间内得到服务。

在一个任务繁重的系统中，多线程带来的性能上的好处可给人一种假象，误认为系统本身非常高效。每一个线程分配给一个小的时间块并以循环等待的形式得到服务。当线程的时间块用尽时，操作系统获得该线程的执行状态信息，所以在下一个循环中可返回到该线程上次的退出点继续执行。然后，操作系统转而执行下一个线程。线程的创建和线程键的切换都不是任意的。当多线程正在运行时，操作系统不会让一个单线程占据太多的时间。当线程的数目很多时，为了使每一个线程都响应良好，每一个时间片会占据很短的一段时间。

让我们举一些数值来具体的说明问题：假设有十项任务要完成。当不受干扰时，每一个任务要花费两秒钟才能完成。如果在同一线程内一起运行这十项任务，第一项任务两秒钟后完成，第十项任务二十秒后完成。总的执行时间是二十秒，最大响应时间也为二十秒，平均响应时间则为十一秒。如果在各个独立的线程中运行每项任务，假设每一项任务分给占总的时间的十分之一的时间片，则总的运行时间为二十二秒。

在多线程的程序中，平均响应时间和最大响应时间不易计算。惟一可以肯定的是对于一个多线程的程序来说（各线程有相同的执行优先权），如果各线程都执行相同的任务，最大响应时间、最小响应时间和平均响应时间都近似相等。CPU 的性能和其他的一些系统级的服务的质量，例如网络和硬件访问的速度，决定了多线程系统的实际响应时间。如果每一个线程在它所分配给十分之一的时间内全速的工作，这十个线程都在二点二秒后完成。实际上，这样高的效率只有在每一个线程在一个独立的进程内并且每一个进程都有一个专用的 CPU 时才能获得。但是，如果每一个线程都百分之百的占用 CPU，所有十个线程总计要用二十二秒的时间。这也就是说，如果系统不忙的话，多线程工作的效率很高。但是如

果系统已接近它能处理的极限时，情况正好相反。如果系统的处理能力有很大的剩余，平均响应时间会从十一秒降低为二点二秒，全部响应时间也将从二十秒降为二点二秒。但是，如果系统的任务特别繁忙，平均响应时间将会加倍。

在使用的线程的数目和服务器的服务级别之间寻求平衡是一种艺术。在工作任务繁忙的系统中，读者的目标是牺牲一点总的处理时间以提高平均响应时间，但这更加剧了服务器的繁忙程度。读者必须时刻警惕处理请求是从何处发出的、每一种请求需要怎样的性能。在有十个对象，每个对象都要求运行一个任务的情况下，使用线程是有益的。但是，如果一个实体在继续运行之前要等待十个任务完成，这时最好将这些请求都列在一个独立的线程内，把更多地线程留给别的进程。

如果运行了太多的线程，系统会花费超过总时间十分之一的时间来管理线程。创建太多的线程实际上就是降低了系统的性能。将操作罗列在更少的线程内会使系统变得更高效。对于一个给定的行为，平均响应时间可用下面的等式来表示。可以看到当 QS=TO 时，QS<>1 这一行跳过了 QS=1 时的函数，这就表明要花费三倍于代码运行的时间来管理线程，使用一个包含三个任务的线程要比三个独立的线程运行起来效率更高。这给我们的一个启示是线程并不是免费的。过多的使用线程对程序的性能十分有害。通过使用在 ProjectlProperties 中设置的线程池（thread pool），可以在 ActiveX EXE 服务器中限制线程的使用。

表 13.1 决定多线程系统的平均响应时间

PCO = 操作的处理代价

Slice = 处理器时间统计（0 到 1）

TO = 线程开销（>=0）

QS =队列大小，每个线程包含的任务队列数

ART = 平均响应时间

ART* = QS * b(QS b+ 1)/2**SlicePCO**b(1 +b *TO*/QSb)b**

至今我还没提及一个线程与其他线程交互所用的时间以及多个线程访问共享系统资源的花销。当两个线程共用同一资源时，必须同步它们的行为。线程间的同步常意味着一个线程要等待另一个线程或者被另一个线程所阻塞。当使用 COM STA 时，阻塞因子是不可见的。在 STA 之间的 COM 调用是通过隐藏的窗口上的消息队列自动的实现同步的。COM 调用也是同步的；直到被调用者从方法调用中返回后，调用者才能继续运行。特别糟糕的情况是在被调用的 STA 的最前面有一行代码时。这时一个线程完成它所有的处理，而其他所有的线程都被阻塞。如果有四个线程，线程二到线程四都被线程一所阻塞；这样读者花费了维护四个线程的代价、却只得到了一个线程的好处。

记住下面这几点，使用线程的通用规则是：

- 只要处理一个任务所需要的最小响应时间和平均响应时间许可，尽量的将多个任务放于一个线程中去。
- 只有在系统需要较小的平均响应时间时，才增加线程。
- 在设计线程时，尽量减少线程间的交互。各线程独立时，运行效率最高。
- 使线程保持工作或暂停工作。如果要使工作线程循环工作，应使用同步技术。不要使辅助线程总是在获取数据，因为让主线程获取较多的状态信息，而让辅助线程处于休眠状态会提高程序的效率。

13.4 在客户机 EXE 中创建线程

在线程中运行代码与实际创建一个线程是不相同的。在一个线程中运行 VB 代码需要做一些工作，这包括在 Project\Properties 中做一些设置。如果为 DLL 指定了单元线程的线程模型，读者应设置自己的工程中的每一个对象的 CLSID\{...}\InprocServer32 键为 ThreadingModel 值。COM 由此知道 DLL 支持多个 STA。对于一个 EXE 服务器来说，每一对象的线程（Thread per Object）和线程池设置决定了 VB 接受对一个新对象的外部请求时，是否创建一个新的线程。

现在的问题是：如何让 VB 在同一进程的一个新线程上创建一个对象？答案是：和在外部进程上请求一个对象一样，在自己的服务器上请求这个新对象。内部请求常常在同一线程上创建一个对象。为了支持外部对象创建，首先应创建一个 ActiveX EXE 工程。标准的 EXE 并不能访问公共对象，所以在其中做外部请求是不可能的。为了保证外部请求创建一个新线程，必须选择每对象的线程（Thread per Object）这一属性。

如果使用 NEW 关键字在工程中创建一个对象，VB 通常要创建一个内部实例。由于性能上的考虑，在这种情况下要绕过通常的 COM 创建机制，即使对公共类也是如此。但是如果使用 CreateObject 关键字而不是 New 关键字，VB 不使用 CoCreateInstance API 调用，这种调用要回调 VB。使用 CreateObject 时，VB 不区分请求是来自 ActiveX 外部还是它的内部，它通常假设请求是来自外部并依此创建对象。参考第七章可得到另一种外部创建技术，这种技术不要求使用 ProgID。

在 VB 中，要创建一个独立运行的多线程程序，开始的几步应是：

（1）创建一个新的 ActiveX EXE 工程。

（2）设置每对象的线程（thread per object）的线程选项。

（3）当需要一个新的工作线程时，使用 CreateObject 来创建 MultiUse 类的一个实例。

做完了上面的框架工作后，还有几步需要完成。读者通常需要在客户应用程序中显示一个用户界面并且使主线程上的用户界面和读者创建的工作线程之间能够通信。由于 ActiveX 没有起始窗体，所以必须设置 Sub Main 的起始选项并在那个过程（procedure）中显式的创建一个窗体。读者可以直接创建一个用户界面对象，或者创建一个新对象

（clsMain），然后使用这个新对象创建用户界面。这一方法中内含的一个问题是每一个新线程都要调用 Sub Main，因此使用 CreateObject 来创建工作线程又一次要执行 Sub Main。而读者希望只是在第一次运行时启动应用程序的用户界面。

读者在这里可能会想到通过设置一个全局标志来处理这一问题，这一标志表明您现在是否在第一个线程上。问题会出在由于全局变量是由各个线程单独设置的，所以这一标志在第二个线程中不可见。VB 的应用程序对象在这里是没有用的，所以只有使用 API 调用来确定第一个线程。我所知道的最容易和最省事的机制是基于 App.hInstance 的值添加一个原子（atom）。可以把操作系统提供的原子表想像为一个无用的、键值的集合。读者所要做的是增加或去除一个字符串并检查一下这一字符串已经存在于哪个原子表中。幸运的是，这便是读者要做的所有工作。

```
'ThreadCheck class
'Instancing = Private
Private Declare Function FindAtom Lib "kernel32" _
    Alias "FindAtomA" (ByVal AtomName As String) As Integer
Private Declare Function AddAtom Lib "kernel32" _
    Alias "AddAtomA" (ByVal AtomName As String) As Integer
Private Declare Function DeleteAtom Lib "kernel32 _
    (ByVal Atom As Integer) As Integer
Private m_Atom As Integer
Private Sub Class_Initialize ()
Dim strAtom As String
    StrAtom = AtomString
    If FindAtom (strAtom) = 0 Then
        m_Atom = AddAtom (strAtom)
    Else
    'In order to clear this setting and allow you to debug,
    'run ?DeleteAtom (FindAtom (strAtom)) in the immediate window
    'until it returns 0, then set the next statement to the
    ' If line above.
        Debug.Assert False
    End If
End Sub
Private Function AtomString () As String
    AtomString = "ThreadCheck" & CStr (App.hInstance)
End Function
```

```
Private sub Class_Terminate ()
    If m_Atom Then DeleteAtom m_Atom
End Sub
Public Property Get First () As Boolean
    First = m-Atom
End Property
```

```
"Using ThreadCheck with Sub Main
Private m_ThreadCheck As New ThreadCheck
Sub Main ()
Dim frmMain As New frmMain
    If m_ThreadCheck.First Then
        'Launch the UI
        frmMain.Show
    Else
        'Nothing to do, and the ThreadCheck
        'instance is no longer needed.
        Set m_ThreadCheck = Nothing
    End If
End Sub
```

下一个问题是在主线程（UI）和任何所创建的工作线程之间建立通信联系。这是比较容易办到的。由于 VB 的线程处于 COM STA 之上，它们之间的调用是排队的。COM 要成功的管理一个接口，要求该接口必须是公共注册的。在 VB 工程中，这意味着只对公共类做跨线程调用。在一个工程内的通信是读者要达成的目标，所以并不需要在外部创建这些类。对于对象来说，设置实例属性为 PublicNotCreatable 就足够了。读者的 UI 窗体应该使用 New 来实例化一个 PublicNotCreatable 对象，然后将其传递给工作线程。这个对象也保持一个对窗体的引用。可以使用 Form_Unload 事件来打破窗体和通信对象之间的循环引用。

```
'Main form and cross thread class layout
'form frmMain
Private m_MainClass As clsMain
Private Sub Form_Load ()
    Set m_MainClass = New clsMain
    Set m_MainClass.MainForm = Me
End Sub
```

```
Private Sub Form_Unload ()
    'Break the circular reference.
    Set m_MainClass.MainForm = Nothing
    Set m_MainClass = Nothing
End Sub
Friend Sub Notify ()
    'Stub for clsMain to call back into the form.
End Sub
```

```
'Class clsMain
'Instancing = PublicNotCreatable
Private m_frmMain As frmMmain
Friend Property Set MainForm (ByVal RHS As frmMain)
    'Note that this is a Friend,so it is callable only
    'from its own thread.Also,VB doesn't support a
    'private Form variable in a Public fuction on a Public
    'class.So VB is actually enforcing the desired behavior
    'in this case.
    Set m_frmMain = RHS
End Property
Public Sub Notify ()
    'Stub callback fuction designed to be called by
    'worker threads when they have something to report.
    'Add parameters as needed.
    If Not m_frmMain Is Nothing Then
       m_frmMian.Notify
    End If
End Sub
```

现在所有的代码已经准备就绪，可以在 frmMain 或 clsMain 中添加一个 CreatObject 调用来产生一个工作线程。将一个 clsMain 实例传递给每一个工作线程之后，工作线程再以一种完全同步的形式跨过线程的边界回调主线程。

13.4.1 异步调用

在 COM 内，一个 COM 对象对其他 COM 对象的调用都是同步的。无论是通过方法调用来直接调用一个对象还是通过引发一个事件来间接的调用一个对象，调用者在继续运行代码之前必须等待被调用者完成处理。如果将异步调用应用到一个多线程的 VB 程序中，任何对另一个线程中的对象的调用会阻塞调用线程，直到所有被调用的线程里的处理都已经完成为止。使用多个线程的主要目的是在保持主线程反应良好的同时，能够执行后台处理，所以说当工作线程进行长时间的方法调用时阻塞主线程是一种低效的方法。

由于 COM 方法是同步的，所以使用跨线程调用来引发大量的处理任务不是一个可供选择的选项。实际上，使每一个线程都保持工作的惟一方法是使每一个线程都调用它自己的需做长时间处理的方法。任何跨线程调用应运行尽可能少的代码。多线程的应用程序中一个线程应使用比较短的跨线程调用来调用其他的线程，使其尽可能快的返回其自身。如果一个线程调用它自身并指处理它自身的事情，它就不会被阻塞或者阻塞别的线程。一个能立即返回的异步跨线程调用是异步调用的很好近似。

余下的问题便是：如何迫使一个线程调用它自身？答案是：使用计数器(timer)。可以使用窗体上的计数器控制，但是在非图形化的线程上创建一个窗体只是在浪费资源。我建议读者使用 API 中定义的计数器。下面的代码允许读者设置一个计数器，这个计数器只在调用已实现的 FireTimer_Go 函数时才被引发。

```
'FireTimer.cls (interface definition)
'Instancing = PublicNotCreatable
Public Sub Go ()

End Sub
```

```
'FireOnceTimers.Bas
Private m_FireOnceTimers As New VBA.Collection

Function AddTimer (ByVal FOT As FireTimer) As Long
    'Create a new timer object.
    AddTimer = SetTimer (0, 0, 1, AddressOf TimerProc)
    If AddTimer Then
        'If successful, add the object to a collection as
        'a weak reference. This means that if something
        'happens that kills the class before the Timer fires,
```

```
        'then this won't artificially keep the class instance
        'alive.The downside is that the class must call
        'ClearTimer explicitly on shutdown if FireTimer_Go has
        'not been called yet. FireTimer_Go should always clear
        'this value.
        m_FireonceTimers.Add ObjPtr (FOT), CStr (AddTimer)
    End If
End Function
Public Sub ClearTimer (ByVal TimerID As Long)
    On Error Resume Next
    m_FireOnceTimers.Remove CStr (TimerID)
    If Err Then
        Err.Clear
    Else
        KillTimer 0, TimerID
    End If
End Sub
Private Sub TimerProc (ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal idEvent As Long, ByVal dwTime As Long)
Dim Fire As FireTimer
Dim FireRefed As FireTimer
Dim pFire As Long
Dim strKey As String
    On Error Resume Next
    strKey = CStr (idEvent)
    pFire = m_FireOnceTimers (strKey)
    If pFire Then
        'Remove the weak reference form the collection object,
        'make sure it gets a properly counted reference, clean
        'up the collection and timer, and then call Go.
        CopyMemory Fire, pFire,4
        Set FireRefed = Fire
        CopyMemory Fire, 0&, 4
        m_FireOnceTimers.Remove strKey
        KillTimer 0, idEvent
        FireRefed.Go
```

```
    End If
End Sub
```

```
'Sample Usage
Implements FireTimer
Private m_TimerId As Long
Private m_Data As Variant

Public Sub Go(Data As Variant)
    m_Data = Data
    m_TimerId = AddTimer (Me)
End Sub
Private Sub FireTimer_Go ()
    m_TimerID = 0
    'Process m_Data
End Sub
Private Sub Class_Terminate ()
    If m_TimerID Then ClearTimer m_TimerID
End Sub
```

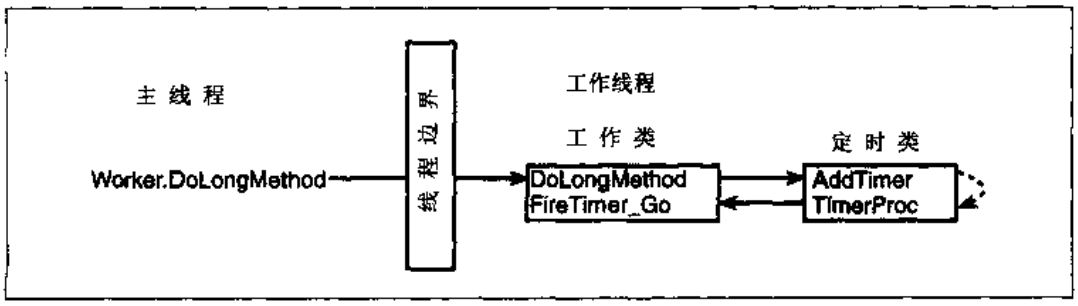

图 13.1 异步调用的一个良好近似，它在被调用的工作线程内使用定时器，以便在工作线程内开始进行多种处理（无需阻塞主线程）

13.4.2 跨线程通信（Cross-Thread Signal）

Visual Basic 提供了两种内在机制来与 COM 对象进行交互：方法和事件。仔细的分析这些跨线程的机制发现这两种机制过于麻烦，不宜在良好设计的多线程系统中使用。

正如我以前提过的那样，多线程的系统应让每一个线程运行时不受其他线程的阻碍，

这意味着跨线程调用被局限于包括创建新线程、近似异步的调用别的线程以开始处理工作然后回调控制线程来返回工作线程得来的结果。不幸的是，如果要保持程序反应良好，仅在这种级别上通信是不够的。通常读者需要加速传递、取消一个长时间操作的能力和“体面”的结束一个程序的能力。如果读者不太小心，支持这些要求可能会带来更多的线程之间的交互，使本应该变得更快一点的线程会变得的十分缓慢。加速和取消支持都涉及与程序结束相近似的问题，程序结束是最后的取消命令。下面首先看一下程序结束命令。

为了支持拆分，控制线程（下面假设它为惟一有用户界面的线程）必须能够通知另一线程上的一个长操作，告知它已经不再被需要并断开与它的联系。是否这个长操作会无限的执行，这一要求是绝对必要的，因为如果没有这种能力，EXE 程序就永远不会结束。如果这种操作执行的时间是有限的，可以孤立该操作，这可通过关闭它的用户界面并且在操作完成后就清除该对象。在关闭所有的用户界面、释放所有的公有对象之前，ActiveX EXE 都一直在运行。如果在用户不再与 EXE 发生交互时，还保持它运行，程序的用户界面就不太友好了。除了浪费处理周期之外，还应注意许多用户都会使用任务管理器（Task Manager）。如果关闭过程要持续一段相当长的时间，这是在欢迎用户粗暴的关闭进程。以一种协作的方式执行完成拆除绝对是比较好的选择。

起初读者可能认为，使用方法调用通知一个正在运行的线程断开联系是一个自然的解决办法。毕竟，单个方法调用只是工作线程的整个生命周期内的一个很小的部分。当然，这样讲是以方法调用能够立即执行为前提的。但是由于是对 STA 做调用，只有在 STA 不处理另一个线程时，方法调用才能立即执行。在这里存在一个问题：要是线程现在不运行一个方法，这个线程就早已结束执行了。运行多个方法调用是工作线程要做的全部工作，因此线程完成之前，不会调用 StopNow 方法，同时控制线程也被阻塞。

为了使这种方法调用能够完成，需要打破 STA 同步机制。由于 COM 使用后台窗口作为保持单元同步的机制，StopNow 方法实际上在一个消息队列中等待正在运行的方法完成。如果在工作线程运行时调用 DoEvents，就能引发 StopNow 方法。在这种情况下，StopNow 应该只是设置一个标志然后立刻返回，这样控制线程不必等待读者完成清除就可以调用其他工作线程上的 StopNow 方法。调用 DoEvents 对于保持线程响应良好已经足够了，但它不是最优的方法，因为这样做除了要添加一些处理重入问题的代码之外，还会显著的增加线程的事件开销。得出的一个结论是使用方法调用来通知会为了接收一个方法调用而减慢线程的整个生命周期。

既然方法调用存在着一些弊端，下面让我们看一下使用事件又会怎样。在这下面的例程中，工作线程定义了一个事件通知（ByVal Progress As Single，ByRef fStopNow As Boolean）。控制线程保持对工作线程的一个 WithEvents 引用。工作线程调用 RaiseEvent Notify 来通知它的进度的控制者，以确定是否退出该线程。就像方法调用一样，这种方法乍看是一种很好的设计。但是，使用事件比使用方法调用更能增加时间开销。

不断的引发一个事件会带来的第一个问题是事件是一种跨线程调用，因此工作线程在调用时会被阻塞。除此以外，如果多个工作线程都要通知控制线程，它们会同时试图与控

制线程对话。这会导致工作线程互相阻塞，结果会是没有一个线程会不受干扰的运行。即使读者偶然使用工作线程调用控制线程，而不是使用 Invoke-bound 事件调用返回来通知控制线程，多个线程也有可能通过排队同时到达主线程，这会造成同样的线程阻塞问题。

我们已经看到：使用方法调用和事件都不是好的解决方法，因此不应局限于 COM 的调用能力来考虑这一问题。读者想到要使用 API 了吗？

13.4.3 直接跨线程存储器访问

Windows 不允许用户读写跨进程存储单元：有 GPF、AV、IPF 或者 TLA du jour。无论读者调用哪个，都会获知进程已经结束。如果两个线程运行在同一个进程中，它们能够合法的访问另一个的线程分配的地址空间。由于 VB 语言内含的同步使这种任意的跨线程存储器访问不可能发生，VB 没有提供访问线程之外的地址空间的内置机制。

让我们暂时忽略同步的要求，看一下如何获取和修改另一个线程的存储单元。为了修改存储单元，需要有一个指向该存储单元的指针。在前面已经使用过 VarPtr、CopyMemory、ZeroMemory 和指针描述符来访问存储单元。还有一组 API 调用能对 32 位的存储单元做一些有限制的修改，例如 InterLockedIncrement、InterLockedDecrement 和 InterLockedExchage。通过使用普通的 COM 调用来跨线程传递由 VarPtr 产生的指针，读者可以访问并修改另一个线程里的存储单元。但是应记住一点：不要在别的线程中调用 STA COM 对象里的方法，传递 ObjPtr 是一个很拙劣的主意。

传递和修改存储单元都是很容易办到的，但是做这些必须遵循一些简单规则。规则一是必须保证所访问的存储单元没有被释放。这通过让 VarPtr 指向类实例中的一个成员变量就可实现。若修改存储单元的线程拥有一个分配存储单元的类的引用，就可确保存储单元的有效性。规则二是不要让两个线程同时修改存储单元。当一个线程修改存储单元时，另一个线程可以访问这一单元，但是让两个线程同时修改存储单元会带来无法预料的结果。

规则三有一些微妙：当在拥有指针指向的地址的线程里访问一个变量时，不要相信 VB 会给出它的当前值。在 C++中，可以标志一个变量为易变的，这会使编译器直接访问它的存储单元以取得该变量的当前值。如果一个变量未标识为易变的，可以通过一个访问函数或者从变量的 VarPtr 中直接拷贝数据。其中，通过一个访问函数来访问是最容易的方法。

下面看一下这些代码。控制线程必须存储两个值：对工作线程的对象的引用和指向信号变量的指针。为避免在控制线程中为每一个工作线程创建一个帮助类，可将对象/指针对移入一个流通类型的变量中，以便于将这一对数据存储在 ObjPtr-keyed 集合中。

```
'Controlling thread
Private Type WorkerData
    NotifyPointer As Long
    WorkerObject As Worker
```

```
End Type
Private m_Workers As Collection
Private Sub Class_Initialize ()
    Set m_Workers = New Collection
End Sub
Friend Sub StartNewWorker (Data As Variant)
Dim WD As WorkerData
Dim CacheWD As Currency
Dim strKey As String

    'Creat the worker.
    Set WD.WorkerObject = CreateObject ("MyApp.Worker")

    'Lauch the task and retrieve its notify pointer.
    WD.NotifyPointer = WD.WorkerObject.DoStuff (Me, Data)

    'Transfer ownership of WD to the Currency
    'and place it in the collection
    strKey = CStr (ObjPtr (WD.WorkerObject))
    CopyMemory CacheWD, WD, LenB (CacheWD)
    ZeroMemory WD, LenB (CacheWD)
    m_Workers.Add CacheWD, strKey
End Sub
Public Sub WorkerDone ( _
    ByVal Worker As Worker, Data As Variant)
Dim WD As WorkerData
Dim CacheWD As Currency
Dim strKey As String
    'Do something with Data
    strKey = CStr (ObjPtr (Worker))

    'If worker has been signalled, then
    'it is already out of the collection,
    'so we have to error trap this.
    On Error Resume Next
    CacheWD = m_Workers (strKey)
```

```
    On Error GoTo 0
    If CacheWD Then
        CopyMemory WD, CacheWD, LenB (CacheWD)
        CacheWD.Remove strKey
    End If
End Sub
Friend Sub SignalWorkers ()
Dim WD As WorkerData
Dim CacheWD As Currency
Dim Iter As Variant
    For Each Iter In m_Workers
        CacheWD = Iter
        CopyMemory WD, CacheWD, LenB (CacheWD)
        InterlockedIncrement WD.NotifyPointer
        Set WD.WorkerObject = Nothing
    Next
    Set m_Workers = New Collection
End Sub
```

```
'Worker class
Implements FireTimer
Private m_Notify As Long
Private m_TimerID As Long
Private m_Data As Variant
Private m_Controller As Controller
Private Function HaveBeenNotified () As Boolean
    HaveBeenNotified = m_Notify
End Function
Public Function DoStuff ( _
    ByVal Controller As Controller, _
    Data As Variant) As Long
    m_Data = Data
    Set m_Controller = Controler
    m_TimerID = AddTimer (Me)
    DoStuff = VarPtr (m_Notify)
End Sub
```

```
Private Sub FireTimer_Go()
Dim Data As Variant
    m_TimerID = 0
    'Process Data
    'Do
    If HaveBeenNotified Then
        'Just quit, or call WorkerDone.
    End If
    'Loop
    'WorkerDone required for teardown.
    m_Controller.WorkerDone Me, Data
End Sub
Private Sub Class_Terminate ()
    If m_TimerID Then
        ClearTimer m_TimerID
    End If
End Sub
```

应该注意，本例对拆除操作顺序的要求是相当松的。如果调用 SignalWorkers 并且控制线程释放控制对象，工作对象会释放最后一个对控制对象的引用。在控制对象释放之后，工作线程上可能还有代码在运行。但是，如果在调用 SignalWorkers 之前所有的工作线程调用 WorkerDone，这说明最后运行的代码是在控制线程中，而不是在工作线程中。在 ActiveX EXE 中，由于各个单元都可独立运行，以任意方式拆除线程不会带来危险。VB 会自动协调好拆除工作。但在以后将会看到当使用在 DLL 中创建的线程时，任意的拆除线程会带来致命的错误。

13.4.4 直接内存访问

遵照上面介绍的规则一，为了让控制线程通知一个工作线程，控制线程必须含有工作对象的引用，该工作对象拥有信号到达的目的地址空间。如果要工作线程修改控制线程拥有的地址空间，也要注意类似的限制条件。让我们设计一个工作线程，它能不断的向控制线程发进度报告并可由控制线程取消或终结。

在这个例子中，与前面主要的差异在于工作线程含有一个指向控制线程拥有的一块存储单元的指针。为了提供这片地址空间，控制线程应为每一个工作线程创建一个私有对象（我们称之为线程数据），工作线程含有一个指向 ThreadData 类中地址部分的指针。

```
'ThreadData class
Private m_NotifyProgress As Long
Private m_CancelPtr As Long
Private m_Worker As Worker
Friend Sub CreateWorker (
    Parent As Controller, _
    Data As Variant, _
    Workers As Collection)
    Set m_Worker = CreateObject ("MyApp.Worker")
    m_CancelPtr = m_Worker.DoStuff ( _
        Parent, Data,VarPtr(m_NotifyProgress))
    Workers. Add Me, CStr (ObjPtr (m_Worker))
End Sub
Friend Sub CancelWorker()
     'Signal if we haven't signalled before.
     If m_CancelPtr Then
         InterlockedIncrement m_CancelPtr
         m_CancelPtr = 0
    End If
End Sub
Friend Property Get Progress () As Single
    'Return a percentage.
    Progress = m_NotifyProgress / 100
End Function
```

```
'Controlling class.
Private m_Workers As Collection
Friend Sub StartNewWorker (Data As Variant)
Dim ThreadData As New ThreadData
    ThreadData.CreateWorker (Me, Data, m_Workers)
End Sub
Public Sub WorkerDone ( _
    ByVal Worker As Worker, Data As Variant)
    'Do something with Data.

    'Remove the ThreadData object form the collection
```

```
    'Unlike the first example, we know it is
    'there because signaling does not remove the
    'item from the collection.
    m_Workers.Remove CStr (ObjPtr (Worker))
End Sub
Friend Sub SignalWorkers ()
Dim TD As ThreadData
    For Each TD In m_Workers
        TD.CancelWorer
    Next
End Sub
Private Sub Class_Initialize ()
    Set m_Workers = New Collection
End Sub
```

```
'Worker class
'Same as before, except that DoStuff takes a ProgressPtr
'parameter.Worker should increment this value 100 times
'with InterlockedIncrement while performing its
'processing.
```

完成整个设计后，读者会得到一个非常有效的线程结构，在这种结构中，可以在任意时刻取消工作线程，而且工作线程会向控制线程发送进度通知。可以在控制线程的用户界面中使用一个计数器来显示进度数据。只有在工作线程创建和终结时，跨线程 COM 调用才会发生，因此控制线程和它的工作线程可以独立的运行，不会相互阻塞。

13.4.5 消除 proxy/stub 的时间开销

即使将 COM 调用的数目降到最低限度，对程序做进一步的优化也是可能的。可以通过调整 COM proxy/Stub 对的数目来减少开销，proxy/stub 对是由排队建立，用来保存一些引用，这些引用可使信号指针指向有效的地址空间。很明显，由 CreateObject 返回的已排队的对象在启动工作线程并且调用该工作线程时是需要的，但在保持新对象运行时不再需要。读者也能消除控制线程创建 proxy/sub 然后将其传递给每一个工作线程的花费。

为了保持读者的通知指针有效，必须确保存在一个拥有该指针的对象的引用。本例和前一个例子之间的惟一区别在于在本例中创建引用的线程拥有引用。为了保持控制线程活动，每一个 ThreadData 都拥有一个公有控制线程的引用。工作线程只是通过拥有一个对自

身的引用来使自己继续活动。

```
'ThreadData class
Private m_NotifyProgress As Long
Private m_CancelPtr As Long
Private m_ Worker As Worker
Private m_ ReturnData As Variant
Private m_Parent As Controller
Friend Sub CreateWorker ( _
    Parent As Controller, Data As Variant, _
    Workers As Collection)
    Set m_Worker = CreateObject ("MyApp.Worker")
    m_CancelPtr = m_Woker.DoStuff ( _
    Data, VarPtr (m_NotifyProgress), VarPtr (m_ReturnData))
    Workers.Add Me, CStr (ObjPtr (m_Worker))
    Set m_Parent = Parent
End Sub
Friend Sub CancelWorker ()
    'Signal if we haven't signaled before
    If m_CancelPtr Then
        InterlockedIncrement m_CancelPtr
        m_CancelPtr = 0
    End If
End Sub
Friend Property Get Progress () As Single
    'Return a percentage.
    Progress = m_NotifyProgress / 100
End Function
Friend Function Finished (ReturnData As Variant) As Boolean
    Finished = Not IsEmpty (m_ReturnData)
    If Finished Then
        ReturnData = m_ReturnData
    End If
End If
```

```
'Worker class
```

```
Implements FireTimer
Private m_Notify As Long
Private m_TimerID As Long
Private m_ReturnDataPtr As Long
Private m_ProgressPtr As Long
Private m_Me As Worker
Private Function HaveBeenNotified () As Boolean
    HeaveBeenNotified = m_Notify
End Function
Public Function DoStuff (
    Data As Variant, _
    ByVal ProgressPtr As Long, _
    ByVal ReturnDataPtr As Long)
    m_ReturnDataPtr = ReturnDataPtr
    m_ProgressPtr = ProgressPtr
    m_TimerID = AddTimer (Me)
    DoStuff = VarPtr (m_Notify)
    Set m_Me = Me
End Function
Private Sub FirTimer_Go ()
Dim Data As Variant
Dim fDoIncrement As Boolean
    m_TimerID = 0
    'Process Data
    'Do
    If HaveBeenNotified Then
        Data = "Not Completed"
        GoTo Finish
    End If
    'fDoIncrement = True
    If fDoIncrement
        InterlockedIncrement m_ProgressPtr
    End If
    'Loop
Finish:
    'Transfer the memory into the DataPtr.
```

```
    CopyMemory ByVal m_ReturnDataPtr,ByVal VarPtr (Data), 16
    CopyMemory ByVal VarPtr (Data), 0, 2
    'We're done.Release ourselves and exit.
    Set m_Me = Nothing
End Sub
Private Sub Class_Terminate ()
    If m_TimerID Then
        ClearTimer m_TimerID
    End If
End Sub
```

读者在本例中有没有发现可使程序崩溃的错误？这段代码可能会崩溃，问题出在它可能会访问已经释放了的地址空间。在 Finished 返回 True 之前，如果不释放 ThreadData 类，m_ReturnDataPtr 的访问是安全的。由于 ThreadData 拥有一个对控制对象的引用，它会继续存在，这使主线程同样继续存在。另一个直接地址空间访问调用涉及到 m_CancelPtr，它在工作线程的生存期内保持有效。问题出在工作线程可在任意时间完成和终结，因此地址空间可能会是无效的。我们对 CancelWorker 的代码可以做出一些修改，让它首先检查一下数据是否已经返回，如已经返回，表明现在 m_CancelPtr 已经无效。

```
Friend Sub CancelWorker ()
    'Signal if we haven't signaled before.
    IF m_CancelPtr Then
        If Not IsEmpty (m_ReturnData) Then
            InterlockedIncrement m_CancelPtr
        End If
        m_CancelPtr = 0
    End If
End Sub
```

当调用 CancelWorker 而 Worker 类已经被释放时，这时会发生时序上的问题，这是跨线程同步错误的一个很典型的例子。因为代码是以任意的顺序运行的，这些错误很难可靠的再现。读者可以通过易出错误且于分费力的方法来手工检查代码，由此找出错误。在调试器内一步一步的执行程序，然后分析每个线程里的各个模块，看一看是否存在潜在的错误。一个代码块可以是一个过程，也可以是一个语句的一部分。下面举一个例子，让我们看一下 Worker 的 FireTimer_Go 中结尾部分的代码和 ThreadData 的 CancelWorker 之间的交互作用。来自 Worker 的行标为 W;来自 ThreadData 的行标为 T。

```
T:If m_CancelPtr Then
T:If Not IsEmpty (m_ReturnData) Then
W:CopyMemory ByVal m_ReturnDataPtr,ByVal VarPtr(Data),16
W:CopyMemory ByVal VarPtr(Data),0,2
T:InterlockedIncrement m_CancelPtr
T:m_CancelPtr = 0
W:Set m_Me = Nothing
W:End Sub
```

代码以上面的顺序运行会很正常，因为拥有地址空间的 Worker 类释放之前对 m_CancelPtr 作出修改。但是，代码如果以下面的顺序运行时，就存在着明显的问题。

```
T:If m_CancelPtr Then
T:If Not  IsEmpty(m_ReturnData)Then
W:CopyMemory ByVal m_ReturnDataPtr,ByVal VarPtr(Data),16
W:CopyMemory ByVal VarPtr(Data),0,2
W:Set m_Me = Nothing
W:End Sub
T:InterlockedIncrement m_CancelPtr
T:m_CancelPtr = 0
```

当以这种顺序运行时，运行到 InterlockedIncrement 行代码就会崩溃，因为 Worker 实例拥有 m_CancelPtr，而 Worker 的地址空间已经被释放掉了。为了处理这类情况，读者应显式的同步各个代码块，以消除两个线程中的运行代码以一种易崩溃的方式执行的可能性。

可以使用 Win32 提供的关键段（critical section）对象来确保 m_CancelPtr 在释放之后不会被访问，这样就会解决同步的问题。关键段对象变量较之互斥体(mutexes)和事件对象开销要少，因为关键段只在一个进程中运行。可以使用 EnterCriticalSection 来进入一个初始化了的关键段对象。第一个 EnterCriticalSection 对象确立对同步对象的所有权。像所有的阻塞机制一样，读者应尽量减少关键段同步的使用，因为阻塞进程会影响整个程序的性能。这是问题的一个方面，但是，在有些情况下，某些同步还是很有必要的。关键对象由控制对象所拥有，可被 Worker 和 ThreadData 对象所使用。这里用到的 API 都已在 Thread.Olb 中作出了声明（VBoost:用于线程的 API 声明）。

```
'Controller
Private m_CritSect As CRITICAL_SECTION
```

```
Private Sub Class_Initialize ()
    InitializeCriticalSection VarPtr (m_CritSect)
End Sub
Private Sub Class_Terminate ()
    DeleteCriticalSection VarPtr (m_CritSect)
End Sub
Friend Function CritSect () As Long
    CritSect = VarPtr (m_CritSect)
End Function
```

```
'Worker
Private m_pCritSect As Long
Public Function DoStuff ( _
    Data As Variant, ByVal ProgressPtr As Long, _
    ByVal ReturnDataPtr As Long, ByVal CritSectPtr As Long)
    'Same as before
    m_pCritScet = CritSectPtr
End Function
Private Sub FireTimer_Go ()
    'Same as before
Finish:
    'Transfer the memory to the DataPtr.
    EnterCriticalSection m__pCritSect
    CopyMemory ByVal m_ReturnDataPtr, ByVal VarPtr (Data), 16
    CopyMemory ByVal VarPtr (Data), 0, 2
    'We're done.Release ourselves and exit.
    Set m_Me = Nothing
    LeaveCriticalSection m_pCritSect
End Sub

'ThreadData class.
Friend Fub CancelWorker ()
    'Singnal if we haven't signalled before
    If m_CancelPtr Then
        EnterCriticalSection m_Parent.CritSect
        If Not IsEmpty (m_ReturnData) Then
```

```
            InterlockedIncrement m_CancelPtr
        End If
        m_CancelPtr = 0
        LeaveCriticalSection m_Parent.CriSect
    End If
End Sub
```

命令相关代码运行在如下两个命令之一中。

```
'Order 1
T:If Not IsEmpty(m_ReturnData) Then
T:InterlockedIncrement m_CancelPtr
T:End If
T:m_CancelPtr = 0
W:CopyMemory ByVal m_ReturnDataPtr,ByVal VarPtr(Data),16
W:CopyMemory ByVal VarPtr(Data),0,2
W:Set m_Me = Nothing

'Order 2
W:CopyMemory ByVal m_ReturnDataPtr,ByVal VarPtr (Data),16
W:CopyMemory ByVal Varptr(Data),0,2
W:Set m_Me = Nothing
T:If Not IsEmpty (m_ReturnData) Then
T:InterlockedIncrement m_CancelPtr
T:End If
T:m_CancelPtr = 0
```

所有这些命令都是安全的。

13.5 STA 单元中 Coordiate Gate 的崩溃

在 STA 线程模型中，COM 排队（marshaler）通过允许在一个时刻只能处理一个方法调用来提供同步。如果要在现在的方法调用返回之前运行别的代码，必须显式的迫使 COM 让别的调用进入进程中去。这种阻塞机制的一个例外发生在调用外部线程的一个方法时，读者会在调用返回之前收到来自于外部线程的方法调用。如果 COM 不允许这一调用通过，

系统会发生死锁，每个线程都在等待对方。

COM 排队通过在每线程上使用隐藏窗口显示出它的特殊功效。当读者在另外的线程中调用一个方法时，COM 会将调用信息放于被调用线程的同步窗口中等待消息队列进行处理。（这样说有些过于简化，但是具体细节与要讨论的内容无关。）没被处理的 COM 调用的消息在窗口中等待当前消息完成处理。为了在当前消息运行时能处理等待消息，读者必须处理来自隐藏窗口的信息。

Visual Basic 提供了 DoEvents 函数以在当前线程的窗口中任意的清除等待消息。由于等待方法“假装”为一个消息，调用 DoEvents 可使其通过。问题在于 DoEvents 也会让别的消息通过，这样读者得到的不是“涓涓细流”，而是“滔滔洪水”。如果读者只想得到“涓涓细流”，不应使用 DoEvents，而应采取别的方法。

DoEvents 首先清除 SendKeys 函数传递而来的击键消息，然后进入一个循环，该循环使用 PeekMessage API 清除所有的消息。最后，DoEvents 调用 Sleep API 来释放当前线程上余下的时间块。在 DoEvents 中惟一对读者有用的部分是 PeekMessage 循环，并且读者只想对 COM 同步窗口而不是所有的窗口使用运行 PeekMessage。通过限制消息清除的范围，可以消除 DoEvents 中通常出现的重入(reentrancy)副效应。

在处理等待的消息时有两个任务是必需的。首先，读者应找到同步窗口。其次，读者应使用 PeekMessage 和它的支持 API 函数来清除消息循环。如果读者不愿费力找到 OLE 窗口，可将要处理的信息的范围限制在 WM_USER 之内；这一范围是用来处理与方法调用相对应的消息的范围。

这一处理过程的不便之处在于定位窗口要依赖于操作系统。当前绝大多数操作系统(Windows95、OSR2、Windows95、Windows98、WindowsNT4 和 Windows 2000)都有一个窗口类 OleMainThreadWndClass 和一个名字 MainThreadWndName。但是，最早的 Windows95 使用一个以 WIN95 RPC Wmsg 开头的窗口类名。除了类名不同的困难之外，Windows 2000 还增加了一个单一消息窗口（message-only window）的概念。这一新的分类使 OLE 窗口更难在 Windows 2000 上找到。可以使用 FindWindow 或 FindWindowEx API 调用找到要找的窗口，不要使用 EnumThreadWindows 或 EnumChildWindow。

尽管 OLE 窗口在几个操作系统版本中保持相对稳定，但无法确保以后的软件版本能保持现在的定位机制。这样，读者在程序中存在着找不到窗口的可能性。如果确实找不到窗口，应使用第二种技巧：将所有的窗口限制在 WM_USER 范围之内。尽管理论上存在将来的版本的操作系统上运行的窗口定位程序会搜索到错误的窗口的可能性，更有可能会发生的情况是：现在提供的信息不足以在将来正确的定位窗口，它不得不将窗口限制在一个有限的范围内。

下面的例程 SpinOlehWnd 是本书 OlePump.Bas 文件中的一部分。SpinOleWnd 试图通过调用 FindOLEhWnd 来寻找正确的窗口。调用 SpinOlehWnd 允许在当前方法返回之前处理任何等待（Pending）消息。

```
Private m_ fInit As Boolean
Private m_OLEhWnd As Long

Public Sub SpinOlehWnd (ByVal fYield As Boolean)
Dim PMFlags As PMOptions
Dim wMsgMin As Long
Dim wMsgMax As Long
Dim MSG As MSG
    If Not m_fInit Then FindOlehWnd
    IF fYield Then
        PMFlags = PM_REMOVE
    Else
        PMFlags = PM_REMOVE Or PM_NOYIELD
    End If
    If m_OLEhWnd = 0 Then
    'Not sure which window to spin (this is very unlikely).
    'A peekMessage loop on all windows can still beat DoEvents
    'and reduce side effects just by looking at WM_USER
    'messages and higher.Note that the current implementation
    'uses only the single WM_USER message, but this code
    'never runs in the current implementation, so I'm playing
    'it safe.
    WMsgMin = &H400 'WM_USER
    WMsgMax = &H7FFF
    End If
    Do While PeekMessage ( _
        MSG,m_OlEhWnd,wMsgMin,wMsgMax, PMFlages)
        'Probably does nothing, but technically correct
        TranslateMessage MSG
        'Process the message
        DispatchMessage MSG
    Loop
End Sub
```

13.6 在 DLL 中创建工作线程

当读者编写一个 EXE 文件时，可以对线程和进程的生命期进行完全控制。而在编写 DLL 或 OCX 时，线程及其生命期在装载该库文件的 EXE 文件的完全控制之下。在 ActiveX EXE 中，编程人员可以指定 VB 必须为每一个外部创建的对象创建一个新线程。由于在 DLL 中 VB 并不会自动创建一个新线程，所以编程人员必须自己手工地创建工作线程。

读者创建的任何线程必须对进程的生命期反应良好，因为 DLL 无法控制线程的终结。EXE 通常会认为它创建的线程都在它的进程内运行。当最后一个线程结束时，进程也会终止。EXE 对读者在 DLL 中创建的工作线程一无所知，因此当进程被 EXE 终止时，工作线程有可能仍在运行，那时它会陷于孤立乃至崩溃状态。

如果要在 DLL 中创建工作线程，需要克服以下几个问题。其中一些问题与 EXE 文件中线程间交互时出现的问题相同，但生命期和线程初始化的问题是 DLL 中的工作线程所独有的。读者创建的每一个工作线程必须满足以下几个条件（除最后一个条件不是必需的条件以外）。

- VB 不会自动创建一个新线程，所以必须用 CreateThread 手工的创建一个线程。
- 只能使用类型库中定义的 API 函数来初始化已创建的线程。
- 当线程完成初始化，可以支持对象创建时，应在这一线程上的当前 DLL 中创建 VB 对象。
- 应该在 EXE 结束 DLL 中的一个线程之前，首先结束由此线程创建的工作线程。
- 应使主线程比较容易能通知工作线程，这样控制线程在拆分时不会被阻塞。
- 如果想要在工作线程中调用主线程里对象上的方法，就必须将调用对象排队进入工作线程。

下面将介绍在 DLL 中创建工作线程的结构的三种不同的形式。第一种结构可在主线程和工作线程之间传递对象，并在线程创建时能立即返回错误信息。结构二不支持跨线程的对象之间的通信，不提供及时的信息反馈。（在本书的光盘中包含了一个介于结构一和结构二之间的结构，它不支持对象，但支持及时的信息反馈。）当结构的特征减少了，它的开销也会随之减少。最后一种结构是第二种结构的一种变形，它能使读者再用工作线程，而不是要不断的创建和销毁它们。

DLL 中的工作线程在客户应用程序中是一个很强大的工具。但是，当要在服务器中使用它时，读者应特别慎重。服务器结构（例如 COM+和 IIS）会很小心地平衡在它们在进程空间中创建的线程的数量。如果读者创建了别的线程，就会打破服务器的线程之间的平衡。在一个线程终结时阻塞另一个线程也是一种不合适的做法。

认为单元线程在 IIS 空间内特别没有效率是一种常见的错误概念。实际上，IIS 线程本身就是单元线程，所以它们可以很好的运行单元线程化的 DLL。VB5 在对象创建和拆分时存在一些不必要的同步，带来了一些性能上的问题，但是 VB6 改正了这些错误。VB6 在创

建时只等待三个关键段（critical sections），当线程要创建时，这三个关键段都已经在堆栈中了。在一次单处理器计算机的测试中，VB6 能在一秒内处理 350 个 ASP 请求，这与自由线程化的 ATL 对象在性能上没有明显的性能差异。读者还可以使用 ActiveX DLL 的驻留内存（Retain In Memory）特征来保证一旦 ASP 线程完成初始化，在它终结之前会

一直保持。单元线程化一个对象对性能没有负面的影响，除非读者将它存储在会话范围（session scope）内。(IIS 不允许在应用程序范围（application scope）内使用单元线程化对象。）如果将一个单元线程化对象存储于会话范围之内，就会将会话锁定为只能使用一个线程。如果两个会话同时锁定于一个线程，它们会等待对方，即使现在另外还有 99 个不活动的线程。

13.6.1 基本结构

基本的线程化结构包括三个核心组件和两个用户定制组件。下面将对每一部分的作用进行描述，然后介绍三个不同的实现。对象之间的相互作用见图 13.2 所示.

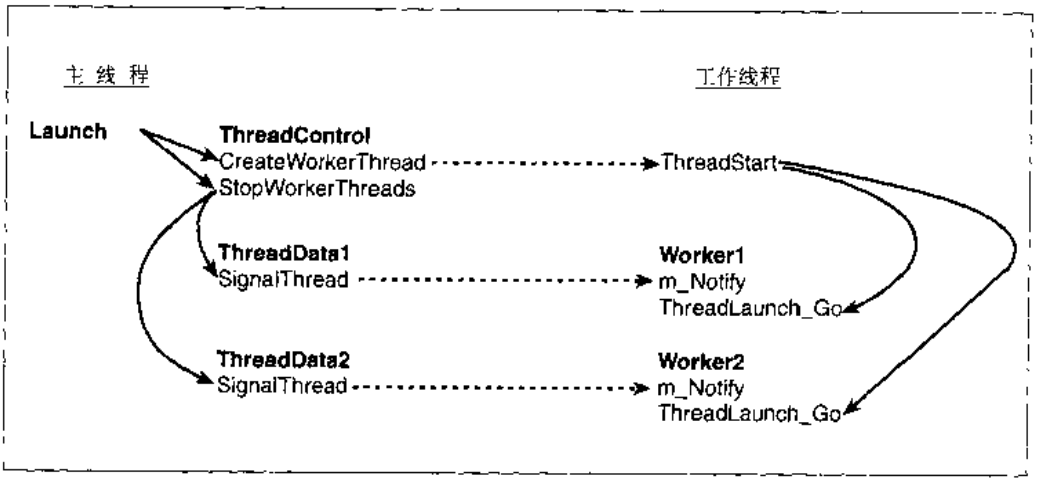

图 13.2 在线程边界的各个方面有良好定义的职能的对象，它能使工作线程的创建变得更加安全和有效率

1．ThreadControl 对象

ThreadControl 是一个负责创建工作线程并在工作线程终结时阻塞主线程的对象。ThreadControl 有两个主要的函数：CreateWorkerThread 和 StopWorkerThreads。使用 CreateWorkerThread 可以得到工作线程创建的对象的 CLSID，这些对象完成实际的工作。它还可以得到传递给新对象的 InputData 变体。每一种结构都为 CreateWorkerThread 指定一些可选参数。第二个方法 StopWorkerThreads 不带任何参数，它在所有的工作线程结束后才返回。

2. ThreadData 对象

ThreadData 和 ThreadControl 对象存在于同一个线程之中。每一个工作线程都创建一个 ThreadData 对象。ThreadData 有三种主要的职能。首先，它拥有一个对创建它的 ThreadContol 实例的引用，这保证了 ThreadContol 比它创建的所有工作线程生存的都要久。其次，它拥有调用线程的句柄。这一句柄允许 ThreadData 确定工作线程是否正在运行并在工作线程结束时获取它的结束代码。再次，ThreadControl 提供了一块可由工作线程在线程结束时修改的存储单元。这提供了一个用来通知线程已经结束的有效机制。ThreadData 需要用一种很麻烦的方式才能清除掉：由计时器或别的机制触发的 Launch 对象必须调用 CleanCompleted Threads 来显式的释放 ThreadData 对象并关闭它的线程句柄。CreateWorkerThread 和 StopWorkerThread 内含的调用 CleanCompletedThreads。

3. 工作（Worker）类

工作类是一个实现 ThreadLaunch 接口的 MultiUse 类。ThreadLaunch 只有一个函数：Go，它是一个新线程的函数入口点。在 ThreadLaunch_Go 完成之后，线程才会终结。工作线程必须提供一个指向 ThreadControl 对象的信号指针，并且它要不断的检查信号来确定是否应该结束线程。

4. ThreadStart 过程

ThreadStart 过程是一个新线程的起始点。ThreadStart 借助于 CreateThread API 来接收数据，其中 CreateThread API 可由 CreateWorkerThread 来调用。ThreadStart 使用 API 调用来初始化线程并且调用 CoCreateInstance API 来创建对象。然后，ThreadStart 查询已创建对象的 ThreadLaunch 接口并调用 Go 来完成剩余的工作。

5. Launch 对象

Launch 是拥有 ThreadControl 对象引用的任一对象。Launch 必须是 PublicNotCreatable 或 MultiUse 对象，这样外部对 Launch 对象的引用可以使服务器保持活动。在 Launch 释放 ThreadControl 对象之前，它必须调用 ThreadControl.StopWorkerThread。StopWorkerThreads 通常是在 Launch 对象的 Terminate 事件中来调用的。

13.6.2 有跨线程对象支持的工作对象

如果读者要创建一个使用标准的 COM 方法来调用主线程的工作线程，可以使用自己熟悉的编程模型。使用 COM 方法来回传递数据是易于编码实现的，但它带来了在方法调用时阻塞主线程和工作线程的时间开销。如果两个或更多的工作线程等待主线程，它们也会彼此等待。这样会带来同步瓶颈问题。

在第一个线程模型中，我曾经向工作线程传递了一个 ThreadControl 对象，这样做是为了在 ThreadLaunch_Go 函数的开始，每一个工作线程都能调用 RegisterNewThread 方法。ThreadControl 的实例属性设定为 PublicNotCreatable，而 RegisterNewThread 声明为一个公有函数。ThreadControl 上的其他可使用的方法为友元方法而非公有方法，这就迫使要对其做调用必须在 ThreadControl 的线程中进行（除注册调用返回之外）。

RegisterNewThread 带有四个参数，它们对工作线程的正常运行和拆分都是很关键的。跨线程的交互可见图 13.3。

1．ThreadDataCookie

ThreadDataCookie 是 ThreadData 对象的 ObjPtr。ThreadData 本身是一个私有对象，不能在线程间进行传递，但是它的指针可以在线程间传递以在 RegisterNewThread 中惟一标识工作线程。

2．ThreadSignalPointer

ThreadSignalPointer 是工作线程类中的变量的地址，它可以被增加以向线程传递信息。

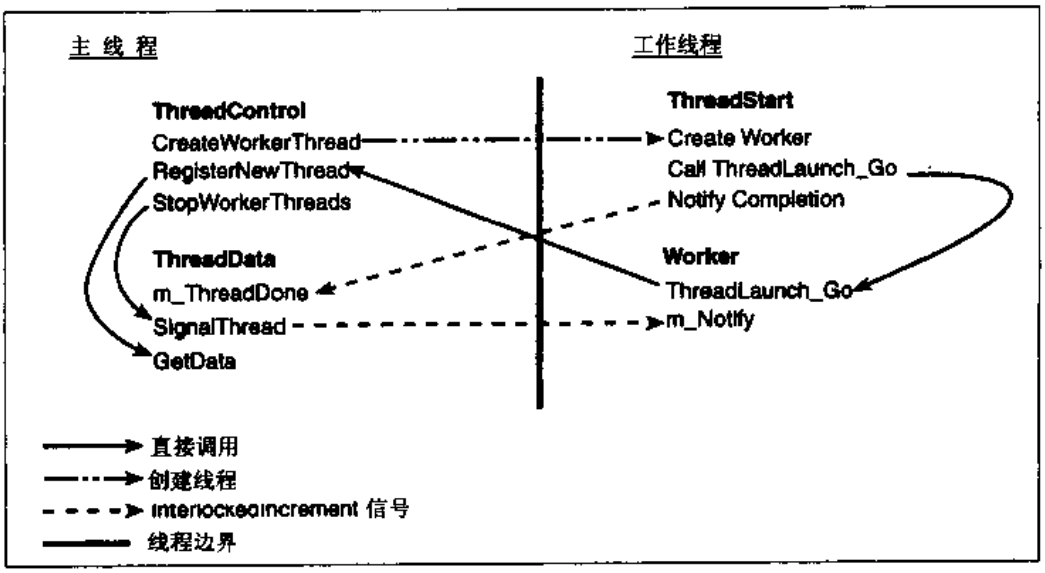

图 13.3 支持通过调用 COM 方法来与控制线程通信的工作线程所需的跨线程交互

3．ThreadControl

ThreadControl 是一个 ByRef 参数，它可以确保工作线程在 RegisterNewThread 调用之后不再拥有对 ThreadControl 的引用。如果工作线程拥有对 ThreadControl 的引用，有可能 ThreadControl 对象最后是在工作线程而不是在 Launch 类中释放的。

4. InputData

InputData 是传递给 CreateWorkerThread 的一个变体。InputData 可以包含任意类型的数据，包括对象类型的数据。为了对从控制线程传送给工作线程的 InputData 中的对象类型进行排队，Register 必须是公有类型。由于在工作线程中可以在任意时刻调用方法，并且在所有的工作线程结束之前，StopWorkerThreads 不会阻塞，所以 StopWorkerThreads 必须调用 SpinOlehWnd 来使消息通过。旋转（spin）消息能使等待的 RegisterNewThread 消息通过。但是 StopWorkerThreads 能确保消息只传送给控制线程。这对 InputData 对象施加了一个基本的限制：这些 InputData 对象必须是调用线程的 STA 的固有对象，从而保证调用线程能正确的结束。

读者必须通过 COM 排队来调用另外一个线程内的 STA 对象上的方法。任何使用 CoCreateInstance 在另一个线程或进程内创建的对象都已经自动的排队，就如同传递给跨线程对象的方法或属性的任意对象一样。除了在 DLL 中显式的创建线程时，由于对象的固有线程和调用线程都已经充分的 COM 初始化了，不需要排队外，其他的情况下都需要。线程创建是一个例外，因为当读者试图传递对象给它时，调用线程还没有被创建，更没有进行 COM 初始化。

COM 提供了两个 API 函数来在线程之间正确的给一个对象排队。读者可以在固有线程内调用第一个 API 函数 CoMarshalInterThreadInterfaceInStream 来为一个对象创建排队数据。这些数据存储在多线程流对象内，因此过程内的任意线程不需要排队就可以调用它们。ThreadAPI.olb 中的 API 声明将返回的 IStream 对象定义为一个长整型变量，这使它在 VB 中更易于使用。流指针传递给 ThreadStrart 过程，这一过程调用 CoGetInterfaceAndReleaseStream 来获取排队好的对象。读者应经常对一个对象的控制线程进行排队，然后使用它来获取其他已经正确的排队的对象。

CreateThread 函数是用来创建线程的关键 API 函数。CreateThread 有几个参数，其中有两个参数对这个结构是十分重要的。IpStartAddress 参数接收 ThreadStart 函数的地址，IpParameter 参数接收传递给 ThreadStart 的数据的地址。传递给 IpParameter 的数据是一个结构的 VarPtr，这个结构包含在新线程上创建对象以及与控制线程相协调所需的所有信息。其他的参数分别是工作类的 CLSID、指向未排队的数据流的指针和一个用来协调拆分的关键段指针。ThreadStart 接收到的数据实际上是一个 CreateWorkerThread 中的局部结构变量的引用。在工作对象从局部变量中读出所有的数据之前，ThreadControl 使用一个 Win32 事件对象来阻塞 CreateWorkerThread。

程序清单 13.1　用以支持跨线程对象的 ThreadLaunch、ThreadControl、ThreadStart 和 Thread Data

```
'ThreadLaunch interface definition,ThreadLaunch.cls
'Instancing = publicNotCreatable
Public Function Go (Controller As TrheadControl, _
```

```
    ByVal ThreadDataCookie As Long ) As Long
End Function
```

```
'ThreadControl class, ThreadControl.cls
'Instancing = publicNotCreatable

'Collection that holds ThreadData objects for each thread
Private m_RunningThreads As Collection
'Currently tearing down, so don't start anything new
Private m_fStoppingWorkers As Boolean
'Synchronization handle
Private m_EventHandle As Long
'Critical section to avoid conflicts when signalling threads
Private m_CS As CRITICAL_SECTION
'Pointer to m_CS structure
Private m _pCS As Long

'Called to create a new worker thread.
'CLSID can be obtained from aProgID via CLSIDFromProgID.
InputData contains the data for the new thread
'fStealInputData should be True if the data is large.If
' this is set InputData is Empty on return.If
' InputData contains an object reference, the object
' must have been created on this thread.
'fReturnThreadHandle must explicitly be set to True to
' return the created thread handle.This handle can be
' used for calls like SetThreadPriority and must be
' closed with CloseHandle.
Friend Function CreatWorkerThread ( _
    CLSID As CLSID, InputData As Variant,_
    Optional ByVal fStealInputData As Boolean = False,_
    Opional ByVal fReturnThreadHandle As Boolean = False) _
    As Long
Dim TPD As ThreadProcData
Dim IID_IUnknown As VBGUID
Dim ThreadID As Long
```

```
Dim ThreadHandle As Long
Dim pStream As IUnknown
Dim ThreadData As ThreadData
Dim fCleanUpOnFailure As Boolean
Dim hProcess As Long
    If m_fStoppingWorkers Then Err.Raise 5, , _
        "Can't creat new worker while shutting down"
    'We need to clean up sometime, this is as good a time
    'as any.
    CleanCompletedThreads
    With TPD
        Set ThreadData = New ThreadData
        .CLSID = CLSID
        EventHandle = m_EventHandle
        With IID_IUnknown
            .Data4 (0) = &HC0
            .Data4 (7) = &H46
        End With
        .pMarshalStream = _
            CoMarshalInterThreadInterfaceInStream ( _
            IID_IUnknown,Me)
        .ThreadDonePointer = ThreadData.ThreadDonePointer
        .ThreadDataCookie = Objptr (ThreadData)
        .pCritSect = m_pCS
        ThreadData.SetData InputData, fStealInputData
        Set ThreadData.Controller = Me
        m_RunningThreads.Add ThreadData, _
            CStr (.ThreadDataCookie)
    End With
    ThreadHandle = CreatThread (0, 0, _
        AddressOf ThreadProc.ThreadStart,_
        VarPtr (TPD), 0, ThreadID)
    If ThreadHandle = 0 Then
        fCleanUpOnFailure = True
    Else
        'Turn ownership of the thread handle over to
```

```
        'the ThreadData object.
        ThreadData.ThreadHandle = ThreadHandle
        'Make sure we've been notified by ThreadProc before
        'continuing to guarantee that the new thread has
        'gotten the data out of the ThreadProcData structure.
        WaitForSingleObject m_EventHandle,INFINITE
        If TPD.hr Then
            fCleanUpOnFailure = True
        ElseIf fReturnThreadHandle Then
            hProcess = GetCurrentProcess
            DuplicateHandle hProcess,ThreadHandle, _
            hProcess,CreateWorkerThread
        End If
    End If
    If fCleanUpOnFailure Then
        'Failure: clean up ctream by making a reference
        'and releasing it.
        CopyMemory pStream,TPD.pMarshalStream, 4
        Set pStream = Nothing
        'Tell the thread it's done using the normal mechanism.
        InterlockedIncrement TPD.ThreadDonePointer
        'There's no reason to keep the new thread data
        CleanCompletedThreads
    End If
    If TPD.hr Then Err.Raise TPD.hr
End Function

'Called after a thread is created to provide a mechanism
'for stopping execution and retrieving initial data for
'running the thread. Should be called in ThreadLaunch_Go
'with: Controller.RegisterNewThread _
' ThreadDataCookie. VarPtr (m_Notify), Controller, InputData
Public Sub RegisterNewThread ( _
    ByVal ThreadDataCookie As Long, _
    ByVal ThreadSignalPointer As Long, _
    ByRef ThreadControl As ThreadControl, _
```

```
    Optional InputData As Variant)
Dim ThreadData As ThreadData
Dim fInCiticalSection As Boolean
    Set ThreadData = m_RunningThreads ( _
        CStr (ThreadDataCookie))
    ThreadData.ThreadSignalPointer = ThreadSignalPointer
    ThreadData.GetData InputData

    'The new thread should not own the controlling thread
    'because the controlling thread has to teardown after
    'all the worker threads are done running code. This
    'can't happen if we happen to release the last reference
    'to ThreadControl in a worker thread. ThreadData
    'already holds an extra reference on this object, so
    'it is guaranteed to remain alive until ThreadData is
    'signaled.
    Set ThreadControl = Nothing
    If m_fStoppingWorkers then
        'This happens only when StopWorkerThreads is called
        'almost immediately after CreatWorkerThread. We could
        'just let this signal happen in the stopWorkerThreads
        'loop, but this allows a worker thread to be  signalled
        'immediately See note in SignalThread about
        'CriticalSection usage
        ThreadData.SignalThread m_pCS, fInCriticalSection
        If fInCriticalSection Then LeaveCriticalSection m_pCS
    End If
End Sub

'Call stopWorkerThreads to signal all worker threads
'and spin until they terminate.Any calls to an object
'passed via the Data parameter in CreateWorkerThread
'will succeed.
Friend Sub StopWorkerThreads ()
Dim ThreadData As ThreadData
Dim fInCriticalSection As Boolean
```

```
Dim fSignal As Boolean
    If m_fStoppingWorkers Then Exit Sub
    m_fStoppingWrkers = True
    fSignal = True
    Do
        For Each ThreadData In m_RunningThreads
            If ThreadData.ThreadCompleted Then
                m_RunningThreads.Remove _
                    CStr (ObjPtr (ThreadData))
            ElseIf fSignal Then
                'See SignalThread about CriticalSection usage.
                ThreadData.SignalThread _
                    m_pCS,fInCriticalSection
            End If
        Next
        If fInCriticalSection Then
            LeaveCriticalSection m_pCS
            fInCriticalSection = False
        Else
            'We can turn this off indefinitely because
            'new threads that arrive at RegisterNewThread
            'while stopping workers are singalled
            'immediately.
            fSignal = False
        End If
        If m_RunningThreads.Count = 0 Then Exit Do
        'We need to clear the message queue here in order
        'to allow any pending RegisterNewThread messages to
        'come through
        SpinOlehWnd false
        Sleep 0
    Loop
    m_fStoppingWorders = False
End Sub

'Releases ThreadData objects for all threads
```

```
'that are completed.Cleaning happens automatically
'when you call SignalWorkerThreads, StopWorkerThreads,
'and RegisterNewThread.
Friend Sub CleanCompletedThreads ()
Dim ThreadData As ThreadData
    For Each ThreadData In m_RunningThreads
        If ThreadData.ThreadCompleted Then
            m_RunningThreads.Remove CStr (ObjPtr (ThreadData))
        End If
    Next
End Sub

'Call to tell all running worker threads to terminate.If the
'thread has not yet called RegisterNewThread, it will
'not be signaled.Unlike StopWorkerThreads, this does not
'block while the workers actually terminate.StopWorker
'Threads must be called by the owner of this class before the
'ThreadControl instance is released.
Friend Sub SignalWorkerThreads ()
Dim ThreadData As ThreadDada
Dim fInCriticalSection As Boolean
    For Each ThreadData In m_RunningThreads
        If ThreadData.ThreadCompleted Then
            m_RunningThereds.Remove CStr (ObjPtr (ThreadData))
        Else
            'See SignalThread about CriticalSection usage.
            ThreadData.SignalThread m_pCS, fInCriticalSection
        End If
    Next
    If fInCriticalSection Then LeaveCriticalSection m_pCS
End Sub

Private Sub Class_Initialize ()
    Set m_RunningThreads = New Collection
    m_EvenHandle = CreatEvent (0, 0, 0, vbNullString)
    If m_EventHandle = 0 Then
```

```
        Err.Raise &H80070000 + Err.LastDLLError
    End If
    m_pCS = VarPtr (m_CS)
    InitializeCriticalSection m_pCS
End Sub
Private Sub Class_Terminate ()
    'Just in case: This generally does nothing.
    CleanCompletedThreads
    If m_EventHandle Then CloseHandle m_EventHandle
    If m_pCS Then DeleteCritialSection mpCS
End Sub
```

```
'ThreadStart procedure, ThreadProc.Bas
Public Type ThreadProcData
    pMarshalStream As Long
    EventHandle As Long
    CLSID As CLSID
    hr As Long
    ThreadDataCookie As Long
    ThreadDonePointer As Long
    pCritSect As Long
End Type
Private Const FailBit As Long = &H80000000
Public Function ThreadStart ( _
    ThreadProcData As ThreadProcData) As Long
Dim hr As Long
Dim pUnk As IUnknown
Dim TL As ThreadLaunch
Dim TC as ThreadControl
Dim ThreadDataCookie As Long
Dim IID_IUnknown As VBGUID
Dim pMarshalStream As Long
Dim ThreadDonePointer As Long
Dim pCritSect As Long
    'Extreme care must be taken in this function to
    'not execute normal VB code until an object has been
```

```
'created on this thread by VB.
hr = CoInitialize (0)
With ThreadProcData
    ThreadDonePointer =.ThreadDonePointer
    If hr And FailBit Then
        .hr = hr
        PulseEvent .EventHandle
        Exit Function
    End If
    With IID_IUnknown
        .Data4 (0) = &HC0
        .Data4 (7) = &H46
    End With
    hr = CoCreateInstance (.CLSID, Nothing, _
        CLSCTX_INPROC_SERVER, IID_IUnknown, pUnk)
    If hr and FailBit Then
        .hr = hr
        pulseEnvent .EventHandle
        CoUninitialize
        Exit Function
    End If
    'If we made it this far, we can start using
    'normal VB calls because we have an initialized
    'object on this thread.
    On Error Resume Next
    Set TL = pUnk
    Set pUnk = Nothing
    If Err Then
        .hr = Err
        PulseEvent .EventHandle
        CoUninitialize
        Exit Function
    End If
    ThreadDataCookie = .ThreadDataCookie
    pMarshalStream = .pMarshalStream
    pCritSect = .pCritSect
```

```
    'The cotrolling thread can continue at this point.
    'The event must be pulsed here because
    'CoGetInterfaceAndReleaseStream blocks if
    'WaitForSingleObject is still blocking the
    'controlling thread.
    PulseEnt .EventHandle
    Set TC = CoGetInterfaceAndReleaseStream ( _
       pMarshalSream, IID_IUnknown)

    'An error is not expected here. If it happens,
    'we have no way of passing it out because the
    'structure may already be popped from the stack,
    'meaning that we can't use ThreadProcData.hr.
    If Err Then
       'Note: Incrementing the ThreadDonePointer call
       'needs to be protected by a critical section once
       'the ThreadSignalPointer has been passed to
       'ThreadControl. Before that time, there is no
       'conflict.
       InterlockedIncrement ThreadDonePointer
       Set TL = Nothing
       CoUninitialize
       Exit Function
End If

'Launch background processing and wait for it to
'finish. Note: TC is released by ThreadControl.
'RegisterNewThread
ThreadStart = TL.Go (TC, ThreadDataCookie)
'Tell the controlling thread that this thread is done.
EnterCriticalSection pCritSect
InterlockedIncremet ThreadDonePointer
LeaveCriticalSection pCritSect
'Release TL after the critical section.This
'Prevents ThreadData.SignalThread from
```

```
    'Signalling a pointer to released memory.
        Set TL = Nothing
    End With
    CoUninitialize
End Function
```

```
'ThreadData class. ThreadData.Cls
Private m_ThreadDone As Long
Private m_ThreadSignal As Long
Private m_ThreadHandle As Long
Private m_Data As Variant
Private m_Controller As ThreadControl
Friend Function ThreadCompleted () As Boolean
Dim ExitCode As Long
    ThreadCompleted = m_ThreadDone
    If ThreadCompleted Then
        'Since code runs on the worker thread after the
        'ThreadDone pointer is incremented, there is a chance
        'that we are signaled but the thread hasn't yet
        'terminated. In this case, just claim we aren't done
        'yet to make sure that code on all worker threads is
        'actually completed before ThreadControl terminates,
        If m_ThreadHandle Then
            If GetExitCodeThread (m_ThreadHandle, ExitCode)
                Then
                    If ExitCode = STILL_ACTIVE Then
                        ThreadCompleted = False
                        Exit Function
                    End If
            End If
            CloseHandle m_ThreadHandle
            m_ThreadHandle = 0
        End If
    End If
End Function
```

```
Friend Property Get ThreadDonePointer () As Long
    ThreadDonePointer = VarPtr (m_ThreadDone)
End Property
Friend Property Let ThreadSignalPointer (ByVal RHS As Long)
    m_ThreadSignal = RHS
End Property
Friend Property Let ThreadHandle (ByVal RHS As Long)
    'This takes over ownership of the ThreadHandle.
    m_ThreadHandle = RHS
End Property
Friend Sub SignalThread ( _
    ByVal pCritSect As Long,ByRef fInCriticalSection As
    Boolean)
    'm_ThreadDone and m_ThreadSignal must be checked/modified
    'inside a critical - section because m_ThreadDone could
    'change on some threads while we are signalling.This
    'causes m_ThreadSignal to point to invalid memory, as
    'well as other problems.The parameters to this function
    'are provided to ensure that the critical section is
    'entered only when necessary. If fInCriticalSection is
    'set, the caller must call LeaveCriticalSection on
    'pCritSect.This is left up to the caller because
    'this function is designed to be called on multiple
    'instances in a tight loop. There is no point in
    'repeatedly entering/leaving the critical section.
   If m_ThreadSignal Then
      If Not fInCriticalSection Then
         EnterCriticalSection pCritSect
         fInCriticalSection = True
      End If
      If m_ThreadDone = 0 Then
         InterlockedIncrement m_ThreadSignal
      End If
      'No point in signalling twice.
      m_ThreadSignal = 0
   End If
```

```
End Sub
Friend Property Set Controller (ByVal RHS As ThreadControl)
    Set m_Controller = RHS
End Property
Friend Sub SetData ( _
    Data As Variant, ByVal fStealData As Boolean)
    If IsEmpty (Data) Or IsMissing (Data) Then Exit Sub
    If fStealData Then
        CopyMemory ByVal VarPtr (m_Data), ByVal _
            VarPtr (Data), 16
        CopyMemory ByVal VarPtr (Data), 0, 2
    ElseIf IsObject (Data) Then
        Set m_Data = Data
    Else
        m_Data = Data
    End If
End Sub
Friend Sub GetData (Data As Variant)
    'This is called only once.Always steal.
    'Before stealing make sure there's
    'nothing lurking in Data.
    Data = Empty
    CopyMemory ByVal VarPtr (Data), ByVal VarPtr ( m_Data_, 16
    Copymemory ByVal VarPtr (m_Data), 0, 2
End Sub

Private Sub Class_Terminate ()
    'This shouldn't happen, but just in case.
    If m_ThreadHandle Then CloseHandle m_ThreadHandle
End Sub
```

13.6.3 缺少跨线程对象支持的工作线程

无论读者是否从工作线程中调用已排队对象，第一种结构都运行良好。但是，如果没有传递一个对象，这种结构的许多部分都会出现问题。第二种结构使用共用内存单元来完成线

程间的通信并在线程间传递输入和输出数据。以这种方式共用内存听起来有些冒险，但是如果你不在同一个时刻从两个不同的线程内向同一块数据区写入数据，并且在已知一个线程已经写入数据时才在另一个线程内读取数据，就能保证操作的安全性。在同一进程内不同的线程共用数据没有 STA 的限制，从非固有线程内调用 STA 对象上的方法时也没有这一限制。

当从一系列要求中去除对象支持这一项时，也就没有必要对从主对象传送给工作对象的 InputData 进行排队。使用 InputData 参数来排队输入数据是 RegisterNewThread 为一个公有方法的惟一原因，这又使 ThreadControl 成为一个 PublicNotCreatable 类。由于缺少了 RegisterThread，没有必要显式的排队和去排队（unmarshal）ThreadControl 对象，因此可以从这一缺少对象支持的结构中删除 CoVeryLongAPIName 函数。在 RegisterNewThread 或别的用户方法拆分时调用主线程没有危险，因此没有必要在 StopWorkerThreads 中旋转（spin）这个 OLE 窗口。

除了去除所有的对象支持调用之外，第二种结构也从控制线程中去掉了事件同步对象。事件对象在线程创建时会立即返回一个错误信息，但它也有阻塞控制线程的副效应。缺少了事件对象，如果读者传递了一个无效的 CLSID 或出现其他一些问题时，就不会得到即时的反馈信息。但是，第二种结构中 ThreadControl 对象上添加了 GetWorkerOutput 函数，通过这一函数可以获得错误信息（仅仅为一个数值）。

事件对象可以确保 CreateWorkerThread 函数的栈上的 ThreadProcData 结构在工作线程获取所有必要的信息之前会一直停留在栈中。如果在 CreateWorkerThread 中不使用事件来锁定控制线程，读者可以将数据分配给一个比堆栈的生存期要长的存储单元。为了使用堆栈来简化问题，控制线程和工作线程所需的所有数据都会移入到 ThreadData 中，其中 ThreadData 现在是一个结构而不是一个类。这个新系统的跨线程交互如图 13.4 所示。每一个 ThreadData 结构的存储空间都是使用 CoTaskMemAlloc 来直接分配的。下面的代码片段表明了 ThreadData 结构中每一个字段的作用。

```
'Key for ThreadData comments
'WT:R = Read by worker thread
'CT:W = Written by controlling thread
'CT:RW = Written and read by controlling thread
'WT:RW = Written and read by controlling thread

Public Type ThreadData
    CLSID As CLSID          'CLSID to create (CT:W,WT:R)
    hr As Long             'An error code (WT:W,CT:R)
    pCritSect As Long      'Critical-section pointer
                           '(CT:W,WT:R)
    ThreadDone As Long     'Increment on completion
```

```
                                    '(WT:W,CT:R)
    InputData As Variant 'Input for the worker (CT:W,WT:R)
```

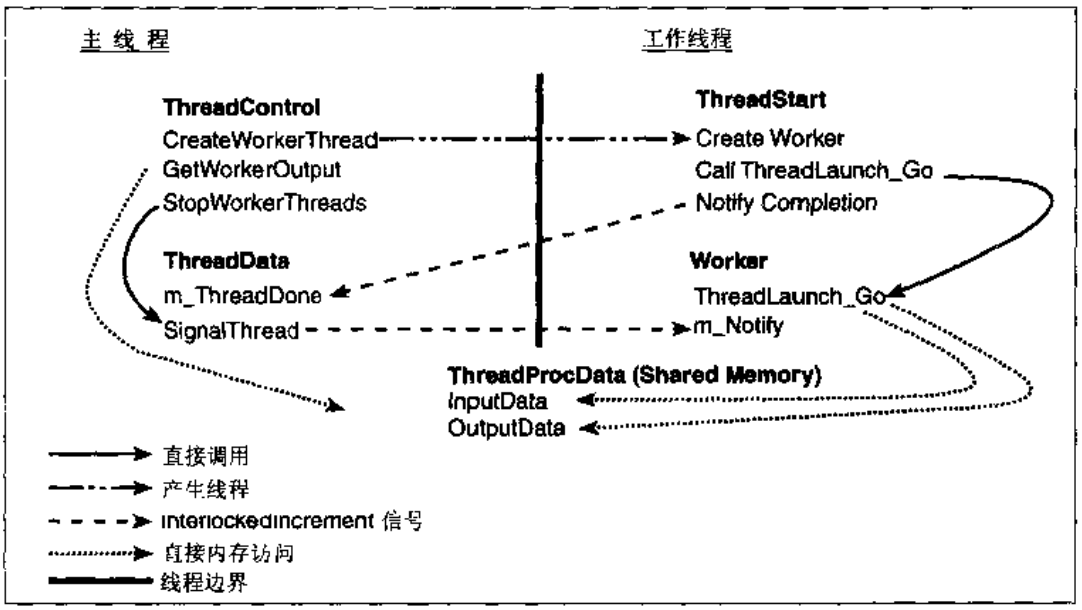

图 13.4 消除对跨线程对象的支持以及对象创建事件，从而从整个系统中移除了所有的跨线程 COM 调用。余下的跨线程交互通过直接内存访问产生

```
    OutputData As Variant   'Output from the worker
                            '(WT:W,CT:R)
    ThreadSignalPtr As Long 'Memory in worker (WT:W,CT:R)
    fSignaled As Boolean    '*TB.ThreadSignalPtr changed (CT:RW)
    fKeepData As Boolean    'Cache output after completion
                            '(CT:RW)
    ExitCode As Long        'The threads exit code (CT:RW)
    ThreadHandle As Long    'Handle to the current thread (CT:RW)
    Controller As ThreadControl  'Reference to controller
                                 '(CT:RW)
End Type
```

在 ThreadData 结构的每一字段上都严格的遵循读/写规则，依此来保证数据的完整性。工作线程能见到的字段是 InputData、OutputData 和 ThreadSignalPtr。ThreadLaunch 接口已经作了修改，下面的程序会将会用到它。对工作线程所作的惟一要求是工作线程应该尽早的设

置 ThreadSignalPtr，并且它不应将对象数据放入 OutputData 变体中。

```
Implements ThreadLaunch
Private m_Notify As Long

Public Function Go(InputData As Variant, _
    OutputData As Variant,ThreadSignalPtr As Long)As Long
    ThreadSignaLPtr = VarPtr(m_Notify)
    'TODO: Process InputData while
    'regularly calling HaveBeenNotified to
    'see if the thread should terminate.
    If HaveBeenNotified Then
        'Clean up and return.
    End If
    'Fill in OutputData.
End Function
Private Function HaveBeenNotified() As Boolean
    HaveBeenNotified = m_Notify
End function
```

读者可以用一个特殊值来调用 GetWorkerOutput，以此在线程结束之后获取线程的 OutputData。给出的这一数值实际上是分配给与工作线程相关的 ThreadData 结构的地址，它可以由 CreateWorkerThread 函数上的 OutputDataCookie 参数来返回。读者应该设置 fKeepOutputData 参数为 True，这样可以缓存输出值。如果决定保存输出，ThreadData 结构就应只是在 m_RunningThreads 和 m_FinnishedThreads 之间传送，而不是释放掉。在调用 GetWorkerOutput 时，这块内存被清空。

GetWorkerOutput 得到这一给定值并返回线程的 ExitCode、启动工作线程或运行 ThreadLaunch_GO 失败后的 HRESULT 以及由工作线程返回的 OutputData 变体。使用内存共用技术，可以通过将一个指向数据的指针值写入用户指定的存储单元来返回数据。GetWorkerOutput 机制是获取数据的一个便利的方法，但不是惟一的方法。

程序清单 13.2 用于 lean-and-mean(非对象)结构的 ThreadLaunch、ThreadStart 和 Thread Data

```
/ThreadLaunch interface definition, ThreadLaunch,cls
/Instancing=PublicNotCreatable
Public Function Go(InputData As Variant,_
```

```
  OutputData As Variant,ThreadSignalPtr As Long) As Long
End Function
```

```
'ThreadControl class, ThreadControl,cls
'Instancing = Private

'Collection to hold ThreadData objects
'for each running thread
Private m_runningThreads As Collection
'Collection to hold ThreadData objects
'for each finished thread
Private m_FinishedThreads As Collection
'Critical section to avoid conflicts
'when signaling threads
Private m_CS As CRITICAL_SECTION
'Pointer to  m_CS structure
Private m_pCS As Long

'Called to create a new  worker thread.
'CLSID can be obtained from a ProgID via CLSIDFromProgID
'InputData contains the data  for the  new thread. This
'   should  never be an object reference.
'fKeepOutData should be True if you want to  retrieve
'   output data  with  GetWorkerOutput. This must be set for
'   a valid  cookie to be  returned  in OutputDataCookie.
'OutputDateCookie retrieves a cookie that can be used
'   later to retrieve the exit code and output variant
'   from a completed worker thread.
'fStealInputData should be True if the data is large. If
'   this is set, Data will be Empty on return.
'fReturnThreadHandle must be explicitly set to True to
'   Return the created thread handle. This handle can be
'   Used for calls like SetThreadPriority and must be
'   Closed with CloseHandle.
Friend Function CreateWorkerThread( _
   CLSID As CLSID, InputData As Variant, _
```

```
    Optional ByVal fKeepOutputData As Boolean = False, _
    Optional OutputDataCookie As Long , _
    Optional ByVal fStealInputData As Boolean = False , _
    Optional ByVal fRetunThreadHandle As Boolean = False)_
        As long
Dim InitThreadData As ThreadData
Dim ThreadID As Long
Dim ThreadHandle As Long
Dim hProcess As Long
Dim pThreadData As Long_
    'We need to clean up sometime,this is as good a time as any.
    'CleanCompletedThreads
    With InitThreadData
        .CLSID = CLSID
        .pCritSect = m_pCS
        Set .Controller = Me
        If fStealInputData Then
          VBoost.MoveVariant .InputData, InputData
        'ElseIf IsObject(Data) Then
          'Dont't support this case: No objects allowed in data.
        Else
          .InputData = InputData
        End If
          .fKeepData = fKeepOutputData
    End With
    pThreadData = modThreadData.NewThreadData(InitThreadData)
    m_RunningThreads .Add pThreadData, CStr(pThreadData)
    If fKeepOutputData Then
        OutputDataCookie = pThreadData
    End If
    ThreadHandle = CreateThread(0, 0, _
        AddressOf ThreadProc.ThreadStart, _
        pThreadData, 0, ThreadID)
    If ThreadHandle Then
        'Turn ownership of the thread handle over to the ThreadData
        'object.
```

```
        modThreadData.ThreadHandle(pThreadData) = ThreadHandle
        If fReturnThreadHandle Then
           hProcess = GetCurrentProcess
           DuplicateHandle hProcess, ThreadHandle, hProcess, _
              CreateWorkerThread
        End If
     End If
End Function

'Call StopWorkerThreads to signal all worker threads and spin until
'they terminate.
Friend Sub StopWorkerThreads()
    modThreadData.StopThreads _
      m_RunningThreads, m_FinishedThreads, m_pCS
End Sub

'Releases ThreadData objects for all threads
'That are completed. Cleaning happens automatically when you call
'SigalWorkerThreads, StopWorkerThreads, and GetWorkerOutput.
Friend Sub CleanCompletedThreads( _
  Optional ByVal fTossCompletedData As Boolean = False)
Dim Iter As Variant
   modThreadData.CleanThreads _
      m_RunningThreads, m_FinishedThreads
   If fTossCompletedData Then
      With m_FinishedThreads
           Do While ,Count
              modThreadData.DestroyThreadData .Item(1)
              .Remove 1
           Loop
      End With
   End If
End Sub

'Call to tell all running worker threads to
'terminate. If the thread hasn't set its
```

```
'ThreadSignalPtr yet, it can't be signaled.
'Unlike StopWorkerThreads, this does not block
'While the workers actually terminate.
'StopWorkerThreads must be called by the owner
'Of this class before the ThreadControl instance
'is released.
Friend Sub SignalWorkerThreads()
   modThreadData.SignalThreads m_RunningThreads, _
      m_FinishedThreads, m_pCS
End Sub
'Call to retrieve the data and exit code from
'a worker thread launched with CreateWorkerThread.
'This returns False if the thread has not
'yet completed. You get one call to GetWorkerOutput
'for each cookie.
Friend Function GetWorkerOutput( _
  ByVal OutputDataCookie As Long, hr As Long, _
  ExitCode As Long, OutputData As Variant) As Boolean
Dim DataKey As String
    CleanCompletedThreads
    DataKey = CStr(outputDataCookie)
    On Error Resume Next
    m_FinishedThreads.Item DataKey
    If Err Then
       On Error GoTo 0
       Exit Function
    End If
    On Error GoTo 0
    modThreadData.GetOutputData _
       OutputDataCookie, hr, ExitCode, OutputData
    modThreadData.DestroyThreadData OutputDataCookie
    m_FinishedThreads.Remove DataKey
    GetWorkerOutput = True
End Function

Private Sub Class_Initialize()
```

```
    Set m_RunningThreads = New Collection
    Set m_FinishedThreads = New Collection
    m_pCS = VarPtr(m_CS)
    InitializeCriticalSection m_pCs
End Sub

Private Sub Class_Terminate()
    'Just in case:This generally only cleans completed data.
    CleanCompletedThreads True
    If m_pCS Then DeleteCriticalSection m_pCs
End Sub

'ThreadStart procedure, ThreadProc.bas
'ThreadData structure defined earlier
Public Function ThreadStart(ThreadData As ThreadData) As Long
Dim pUnk As IUnknown
Dim TL As IUreadLaunch
Dim IID_TUnknown As VBGUID
    'Extreme care must be taken in this function to
    'run normal VB code until an object has been
    'created on this thread by VB.
    With ThreadData
        .hr=CoInitialize(0)
        If .hr And FailBit Then
            .ThreadDone=1
            Exit Function
        End If
        With IID_IUnknown
            .Data4(0)=&HC0
            .Data4(7)=&H46
        End With
        .hr=CoCreateInstance (.CLSTD,Nothing,_
        CLSCTX_INPROC_SERVER, IID_IUnknown,pUnk)
        If .hr And FailBit Then
                .ThreadDone=1
                CoUninitialize
```

```
            Exit Function
        End If

        'If we've made it this far, we can start using
        'normal VB calls because we have an initialized
        'object on this thread.
        On Error Resume Next
        Set TL=pUnk
        Set pUnk=Nothing
        If Err Then
            .hr=Err
            .ThreadDone=1
            CoUninitialize
            Exit Function
    End If

    'Launch the background thread and wait for it to
    'finish.
    ThreadStart=TL.Go(_
    .InputData, .OutputData,. ThreadSignalPtr)
    .hr=Err

    'Tell the controlling thread that this thread is done.
    'Note that the critical section coordinates between
    'ThreadSignalPtr and ThreadDone. ThreadSignalPtr isn't
    'set before TL.Go, so we don't need a critical
    'section to increment ThreadDone until now.
    EnterCriticalSection. pCritSect
    ThreadDone = 1
    Leave Critica Section.pCritsect
    'Release TL after the critical section. This
    'prevents ThreadData. SignalThread from
    'signalling a pointer to released memory.
    Set TL=Nothing
End With
CoUninitizlize
```

```
End Function
```

```
'modThreadData module, in ThreadData.bas
Prvate Type OwnedThreadData
    Owner As ArrayOwner
    pSA() As ThreadData
End TYPE
Private m_Data As OwnedThreadData

'Allocate a ThreadData object on the heap and transfer the
'bits from the incoming structure.
Public Function New ThreadData (InitData As ThreadData) As Long
    With m_Data
      If .Owner.SA.cDims=0 Then
        InitArrayOwner. Owner, LenB(.pSA(0)),_
          FADF_AUTO or FADF_FIXEDSIZE, False
      End If
      NewThreadData = CoTaskMemAlloc (LenB(.pSA(0)))
      If NewThreadData = 0 Then Err.Raise 7 'Out of memory
      CopyMemory ByVal NewThreadData,_
        InitData. CLSID, LenB(.pSA(0))
      ZeroMemory InitData.CLSID,LenB(.pSA(0))
   End with
End Function
Public Sub DestroyThreadData (ByVal pThreadData As Long)
   With m_Data
       .Owner.SA.pvData=pThreadData
      With .pSA(0)
          'This shouldn't happen, but a safety valve
          'is good.
        If .ThreadHandle Then CloseHandle .ThreadHandle
     End With
    'Frees any leftover Variant information and
    'the controller.
  Erase .pSA
End With
```

```
    CoTaskMemFree pThreadData
End Sub

Public Property Let ThreadHandle (_
    ByVal pThreadData As Long, ByVal RHS As Long)
    'This takes over ownership of the ThreadHandle.
    With m_Data
        .Owner.SA.pvData=pThreadData
        .pSA(0).ThreadHandle=RHS
    End With
End Property

Public Sub GetOutputData (ByVal pThreadData As Long,_
    hr As Long, ExitCode As Long, OutputData As Variant)
     With m_Data
          .Owner.SA.pvData=pThreadData
          With .pSA(0)
              VBoost.MoveVariant OutputData, .OutputData
              ExitCode=.ExitCode
              hr=.hr
          End With
       End With
   End Sub

   Private Function ThreadCompleted(_
     ThreadData As ThreadData) As Boolean
   Dim ExitCode As Long
      With ThreadData
          'See comments in first listing.
             ThreadCompleted = .ThreadDone
             If ThreadCompleted Then
                If .ThreadHandle Then
                   If GetExitCodeThread( _
                       .ThreadHandle, ExitCode) Then
                       If ExitCode =    STILL_ACTIVE Then
                          ThreadCompleted = False
```

```
                        Exit Function
                    End If
                End If
                CloseHandle .ThreadHandle
                .ThreadHandle .=0
                .InputData = Empty
                .ExitCode = Exitcode
            End If
        End If
    End With
End Function

Private Sub SignalThread(ThreadData As ThreadData, _
    ByRef fUnregistered As Boolean, _
    ByVal pCritsect As Long, _
    ByRef fInCriticalSection As Boolean)
    'See comments in first listing
    With ThreadData
        If Not .fsignaled Then
            If .ThreadSignalPtr Then
                If Not fInCriticalSection pCritSect
                    EnterCriticalSection pCritSect
                    fInCriticalSection = True
                End If
                If  .ThreadDone = 0 Then
                    InterlockedIncrement .ThreadSignalPtr
                End If
                'No point in signalling twice.
                .fSignaled = True
            Else
                'The worker hasn't set ThreadSignalPtr
                fUnregistered = True
            End If
        End If
    End With
End Sub
```

```
Public Sub StopThreads(RunningThreads As Collection, _
   FinishedThreads As Collection, ByVal pCritSect As Long)
Dim fInCriticalSection As Boolean
Dim fSignal As Boolean
Dim fUnregistered As Boolean
Dim Iter As Variant
Dim pThreadData As Long
Dim DataKey As String
    fSignal = True
    With m_Data
        Do
            fUnregistered = False
            For Each Iter in RunningThreads
                pThreadData = Iter
                . Owner .SA.pvData = pThreadData
                If ThreadCompleted(.pSA(0)) Then
                    DataKey = CStr (pThreadData)
                    RunningThreads. Remove DataKey
                    If .pSA(0).fKeepData Then
                        Set .pSA(0) .Controller = Nothing
                        FinishedThreads .Add pThreadData,
                          DataKey
                    Else
                         'Don't call DestroyThreadData while
                         '.pSA(0) is a current With context.
                        DestroyThreadData pThreadData
                    End If
                    DataKey = vbNullString
                ElseIf fSignal Then
                  SignalThread .pSA(0), fUnregistered, _
                    pCritSect, fInCriticalSection
                End If
            Next
            If fInCriticalSection Then
                LeaveCriticalSection pCritSect
```

```
            fInCriticalSection = False
        Else
            'We can turn this off indefinitely if
              'fUnregistered is False because all threads
              'will have been signaled at this point.
              fSignal = fUnregistered
          End if
          If RunningThreads.Count = 0 Then Exit Do
          'Give up the rest of our time slice.
          Sleep 0
      Loop
   End With
End Sub

Public Sub CleanThreads (RunningThreads As Collection, _
  FinishedThreads As Collection)
Dim pThreadData As Long
Dim Iter As Variant
Dim DataKey As String
   With m_Data
      For Each Iter In RunningThreads
         pThreadData = Iter
          . Owner .SA.pvData = pThreadData
         If ThreadCompleted(.pSA(0)) Then
            DataKey = CStr (pThreadData)
            RunningThreads. Remove DataKey
            If .pSA(0).fKeepData Then
               Set .pSA(0) .Controller = Nothing
               FinishedThreads .Add pThreadData, DataKey
            Else
               'Don't call DestroyThreadData while
               '.pSA(0) is a current With context.
               DestroyThreadData pThreadData
            End If
            DataKey = vbNullString
         End If
```

```
        Next
    End With
End Sub

Public Sub SignalThreads (RunningThreads As Collection, _
  FinishedThreads As Collection, ByVal pCritSect As Long)
Dim pThreadData As Long
Dim Iter As Variant
Dim fInCriticalSection As Boolean
Dim fUnregistered As Boolean 'Dummy
Dim DataKey As String
    With m_Data
        For Each Iter In RunningThreads
            pThreadData = Iter
            . Owner .SA.pvData = pThreadData
             If ThreadCompleted(.pSA(0)) Then
                 DataKey = CStr (pThreadData)
                 RunningThreads. Remove DataKey
                 If .pSA(0).fKeepData Then
                     Set .pSA(0) .Controller = Nothing
                     FinishedThreads.Add pThreadData, DataKey
                 Else
                     'Don't call DestroyThreadData while
                     '.pSA(0) is a current With context.
                     DestroyThreadData pThreadData
                 End if
                 DataKey = vbNullString
               Else
                 SignalThread .pSA(0), fUnregistered, _
                   pCritSect, fInCriticalSection
               End If
            Next
            If InCriticalSection Then _
              LeaveCriticalSection pCritSect
      End With
  End   Sub
```

13.6.4 工作线程的再利用

工作线程花费最多的地方在它们自己创建和销毁的时候。创建线程、初始化 COM、初始化 VB 线程和工程数据都不是免费的。如果创建运行在后台做长时间计算的工作线程，线程初始化的开销相对来说是微不足道的。但是，如果工作线程运行多个较短的操作，这种开销相对来说就会变得十分可观。

我们对线程结构所做的最后的改变是在使线程完成要做的工作之后进入休眠状态，而不是让其结束。当工作线程在休眠时，控制线程将所有的 ThreadData 成员返回到初始状态，它还提供新的输入数据并且在有新的请求时唤醒工作线程。虽然工作线程再用结构与前面的结构基本类似，但也作了一些基本的改变。

第一个改变是每一个 ThreadData 结构都添加了一个 Win32 事件对象。当工作线程处理完数据之后，它调用 ThreadData 事件对象上的 WaitForSingleObject 方法，这一事件对象是一个非通知同步对象。在控制线程调用 PulseEvent 来唤醒工作线程之前，工作线程一直处于被阻塞的状态。当工作线程被唤醒后，它会检查 ThreadData 里的 ThreadDone 字段是否被设置。如果 ThreadDone 字段为 True，ThreadStart 知道现在应该退出 ThreadStart 并且结束线程。

工作线程在离开时总是处于阻塞状态，因此没有必要使用前面结构中使用关键段来保证 ThreadSignalPtr 的有效性。工作线程也不清除工作类。读者应该清除会话之间的任何数据，例如 m_Notify 成员变量。

这种结构中的 ThreadControl 函数同第二种结构中的函数相同。这就带来了一个小的问题：对象返回的 OutputDataCookie 不能作为一个指向 ThreadData 数据的指针来使用，因为 ThreadData 也被工作线程所使用，当一段代码调用 GetWorkerOutput 时，工作线程可能正在运行别的调用。这一问题的一个解决方法是创建一个叫做 ThreadDataOutput 的新结构，它包含 ThreadData 中所包含的 hr、ExitCode 和 OutputData 字段。一个 ThreadData 对象在它的生命期内可以拥有多个 ThreadDataOutput 对象。由 CreateWorkerThread 返回的 OutputDataCookie 现在是一个指向 ThreadDataOutput 的指针，而不是一个指向整个 ThreadData 结构的指针。

程序清单 13.3 用于结构再利用的 ThreadControl、ThreadStart 和 ThreadData

```
'ThreadControl class, in ThreadControl. cls
Private m_RunningThreads As Collection
Private m_FinishedThreads As Collection
'Same comments as previous architecture
Friend Function CreateWorkerThread( _
  CLSID As CLSID, InputData As Variant,
  Optional ByVal fKeepOutputData As Boolean = False, _
```

```
        Optional OutputDataCookie As Long, _
        Optional ByVal fStealInputData As Boolean = False, _
        Optional ByVal fReturnThreadHandle As Boolean = False) _
         As Long
Dim InitThreadData As ThreadData
Dim ThreadID As Long
Dim ThreadHandle As Long
Dim hProcess As Long
Dim pThreadData As Long
Dim pThreadDataOutput As Long
Dim pIdleThreadData As Long
    pIdleThreadData = modThreadData.CleanThreads( _
      m_RunningThreads, m_FinishedThreads )
    With InitThreadData
        .CLSID = CLSID
       Set .Controller = Me
        If fStealInputData Then
           VBoost.MoveVariant .InputData, InputData
       Else
            . InputData = InputData
       End If
        . fKeepData = fKeepOutputData
    End With
    If pIdleThreadData Then
        ThreadHandle = modThreadData.WakeSleepingThread( _
          pIdleThreadData, InitThreadData, pThreadDataOutput)
    Else
        pThreadData = modThreadData.NewThreadData( _
          InitThreadData, pThreadDataOutput)
        m_RunningThreads.Add pThreadData, CStr(pThreadData)
        ThreadHandle = CreateThread(0, 0, _
          AddressOf ThreadProc.ThreadStart, _
          pThreadData, 0, ThreadID)
        If ThreadHandle Then
            'Turn ownership of the thread, handle over to
            'the ThreadData object.
```

```
            modThreadData .ThreadHandle (pThreadData) = _
              ThreadHandle
        End If
    End If
    If fKeepOutputData Then
        OutputDataCookie = pThreadDataOutput
    End If
    If ThreadHandle Then
        If fReturnThreadHandle Then
            hProcess = GetCurrentProcess
            DuplicateHandle hProcess, ThreadHandle, _
              hProcess, CreateWorkerThread
        End If
    End If
End Function
```

```
'Other functions in ThreadControl are the same except that
'there is no critical section, and the OutputData cookies
'are destroyed with modThreadData.DestroyThreadDataOutput
'instead of modThreadData.DestroyThreadData.
```

```
'ThreadStart procedure, ThreadProc.bas
Public Type ThreadDataOutput
    hr As Long              'An error code (WT:W, CT:R)
                            ' (keep first)
    ExitCode As Long        'The thread's exit code  (CT:RW!)
    OutputData As Variant 'Output from the worker (WT:h', CT:R)
End Type
Public Type ThreadData
    CLSTD As CLSID        'CLSID to create (CT:W, NT:R)
    ThreadDone As Long   'Increment on completion (WT:W, WT:R)
     InputData As Variant 'Input for the worker(CT:W, WT:R)
    ThreadSignalPtr As Long 'Memory in worker (WT:W, CT:R)
    fSignaled As Boolean '*TB. ThreadSignalPtr changed (CT:RW)
     fKeepData As Boolean 'Cache output after completetion (CT:RN)
    ThreadHandle As Long 'Handle to the current thread (CT:RW)
```

```
    Controller As ThreadControl 'Reference to controller
                                ' (CT:RW)
    pRecycleEvent As Long 'Synchronization handle (CT:RW)
    pOutput As Long      'ThreadDataOutput pointer
                        ' (CT:RW, WT:R)
End Type
Private Const FailBit As Long = &H80000000
Public Function ThreadStart (ThreadData As ThreadData) As Long
Dim pUnk As IUnknown
Dim TL As ThreadLaunch
Dim IID_IUnknown As VBGUID
Dim SA1D As SafeArrayId
Dim pOutputData ( ) As ThreadDataOutput
Dim hr As Long
    With ThreadData
        hr = CoInitialize(0)
        If hr And FailBit Then
            .ThreadDone = 1
            CopyMemory . pOutput, hr, 4
            Exit Function
        End If
        With IID_IUnknown
            .Data4(0) = &HCO
            .Data4(7) = &H46
    End With
    hr = CoCreateInstance(.CLSID, Nothing, _
        CLSCTX_INPROC_SERVER, IID_IUnknown, pUnk)
    If hr And FailBit Then
        .ThreadDone = 1
        CopyMemory .pOutput, hr, 4
        CoUninitialize
        Exit Function
    End If

    On Error Resume Next
    Set TL = pUnk
```

```
    Set pUnk = Nothing
        If Err Then
        hr = Err
        CopyMemory .pOutput, hr, 4
        .ThreadDone = 1
        CoUninitialize
        Exit Function
    End If
'Launch the background thread and wait for it to
'finish.
With SA1D
        . cDims = 1
        .cElements = 1
             .cbElements = LenB(pOutputData(0))
        End With
        CopyMemory ByVal VarPtrArray(pOutputData), _
          VarPtr (SA1D), 4
 Recycle:
        SA1D.pvData = .pOutput
        With pOutputData(0)
             .ExitCode = TL.Go(ThreadData.InputData, _
              .OutputData, ThreadData. ThreadSignalPtr)
              .hr = Err
        End With
         'Flag this pass as done.
        ThreadData .ThreadDone = 1
        'Wait until the event is pulsed to enable us to
        'recycle our data. If .ThreadDone is still set
        'after we clear this event, we should terminate.
        WaitForSingleObject .pRecycleEvent, INFINITE
        If .ThreadDone = 0 Then GOTO Recycle
        'Use VarPtrArray before releasing TL so that
        'the runtime is still with us.
        ZeroMemory ByVal VarPtrArray(pOutputData), 4
        Set TL = Nothing
    End With
```

```
    CoUninitialize
End Function
```

```
 'modThreadData module, in ThreadData. Bas
Private Type OwnedThreadData.
    Owner As ArrayOwner
    pSA() As ThreadData
End Type
Private Type OwnedThreadDataOutput
    Owner As ArrayOwner
    pSA() As ThreadDataOutput
End Type
Private m_Data As OwnedThreadData
Private m _DataOutput As OwnedThreadDataOutput

'Allocate a ThreadData object on the heap and transfer the
'bits from the incoming structure.
Public Function NewThreadData(InitData As ThreadData, _
    pThreadDataOutput As Long) As Long
Dim TDO As ThreadDataOutput 'Dummy for LenB.
    With m_Data
        If . Owner .SA.cDims = 0 Then
            'Establish flags so we can call Erase
            'to destroy data.
            InitArrayOwner.Owner, LenB ( .pSA (0)), _
                FADF_AUTO Or FADF_FIXEDSIZE, False
            With m_DataOutput
                'fFeatures of 0 is OK because we
                'never Erase ThreadDataOutput.
                InitArrayOwner _
                    .Owner, LenB (.pSA (0)), 0, False
            End With
        End if
    End With
With InitData
    .pRecycleEvent = CreateEvent(0, 0, 0, vbNullString)
```

```
    If .pRecycleEvent = 0 Then _
        Err.Raise &H80070000 + Err.LastDLLError

    On Error GOTO Error
    .pOutput = CoTaskMemAlloc(LenB(TDO))
    If .pOutput = 0 Then Err .Raise 7 'Out of memory
    pThreadDataOutput = .pOutput
    ZeroMemory ByVal .pOutput, LenB(TDO)

    NewThreadData = CoTaskMemAlloc(LenB(InitData)).
    If NewThreadData = 0 Then Err .Raise 7 'Out of memory
    CopyMemory ByVal NewThreadData, .CLSID, LenB(InitData)
    ZeroMemory .CLSID, LenB(InitData)
End With
Exit Function
Error:
    With InitData
        If .pRecycleEvent Then CloseHandle .pRecycleEvent
        If .pOutput Then CoTaskMemFree .pOutput
    End With
    With Err
        .Raise .Number
    End With
End Function

 'A new function to reinitialize and pulse a sleeping thread.
Public Function WakeSleepingThread( _
  ByVal pThreadData As Long, InitData As ThreadData, _
  pThreadDataOutput As Long) As Long
Dim TDO As ThreadDataOutput 'Dummy for LenB
    pThreadDataOutput = CoTaskMemAlloc (LenB (TDO))
    If pThreadDataOutput = 0 Then Err. Raise 7
    ZeroMemory ByVal pThreadDataOutput, LenB (TDO)
    With m_Data
        . Owner. SA. pvData = pThreadData
       With .pSA(0)
```

```
            .fSignaled = False
            . ThreadDone = 0
            .fKeepData = InitData.fKeepData
           VBoost.MoveVariant .InputData, InitData.InputData
            Set .Controller = InitData.Controller
            .pOutput = pThreadDataOutput
            PulseEvent .pRecycleEvent
        End With
    End With
End Function

Public Sub DestroyThreadData(Byval pThreadData As Long)
    With m_Data
        .Owner. SA. pvData = pThreadData
        With .pSA(0)
            'This shouldn't happen, but a safety valve is
            'good.
           If .ThreadHandle Then CloseHandle .ThreadHandle
            'Clean up the event handle: This is expected.
           If .pRecycleEvent Then CloseHandle .pRecycleEvent
            'Clean any remaining ThreadDataOutput.
           If .pOutput Then DestroyThreadDataOutput .pOutput
        End With
        'Frees any leftover Variant information and
        'the controller.
        Erase . pSA
    End With
    CoTaskMemFree pThreadData
End Sub

Public Sub DestroyThreadDataOutput ( _
  ByVal pThreadDataOutput As Long)
    With m_DataOutput
        . Owner .SA.pvData = pThreadDataOutput
        .pSA (0) .OutputData = Empty
    End With
```

```
    CoTaskMemFree pThreadDataOutput
End Sub

Public Property Let ThreadHandle( _
  ByVal pThreadData As Long, ByVal RHS As Long)
    With m_Data
        .Owner .SA.pvData = pThreadData
        .pSA (0) .ThreadHandle = RHS
    End With
End Property

Public Sub GetOutputData(ByVal pThreadDataOutput As Long, _
  hr As Long, ExitCode As Long, OutputData As Variant)
    With m_DataOutput
        .Owner. SA.pvData = pThreadDataOutput
        With .pSA (0)
            VBoost.MoveVariant OutputData, .OutputData
            ExitCode = .ExitCode
            hr = .hr
        End With
    End With
End Sub

Private Function ThreadCompleted( _
  ThreadData As ThreadData) As Boolean
 Dim ExitCode As Long
    With ThreadData
        ThreadCompleted = .ThreadDone
        If ThreadCompleted Then
            If .ThreadHandle Then
                If GetExitCodeThread( _
                   .ThreadHandle, ExitCode) Then
                    If ExitCode = STILL_ACTIVE Then
                        'Wake the thread without clearing
                        'ThreadDone, forcing it to terminate.
                       PulseEvent.pRecycleEvent
```

```
                    ThreadCompleted = False
                    Exit Function
                End If
            End If
            CloseHandle .ThreadHandle
            .ThreadHandle = 0
        End If
    End If
  End With
End Function

Private Sub SignalThread (ThreadData As ThreadData, _
  ByRef fUnregistered As Boolean, ByRef fSignaled As Boolean)
  With ThreadData
    If Not . fSignaled Then
      If .ThreadSignalPtr Then
        fSignaled = True
        If .ThreadDone = 0 Then
          lnterlockedincrement . ThreadSignalPtr
        End if
        'No point in signalling twice.
        .fSignaled = True
      Else
        'The worker hasn't set ThreadSignalPtr.
        fUnregistered = True
      End If
    End If
  End With
End Sub

Public Sub StopThreads(RunningThreads As Collection, _
  FinishedThreads As Collection)
Dim fSignal As Boolean
Dim fSignaled As Boolean
Dim fUnregistered As Boolean
Dim Iter As Variant
```

```
Dim pThreadData As Long
    fSignal = True
    With m_Data
        Do
            fUnregistered = False
            fSignaled = False
            For Each Iter In RunningThreads
            pThreadData = Iter
            . Owner . SA. pvData = pThreadData
            If ThreadCompleted(.pSA(0)) Then
                RunningThreads.Remove CStr (pThreadData)
                With .pSA(0)
                    If . fKeepData Then
                        FinishedThreads.Add .pOutput, _
                            CStr( .pOutput)
                        pOutput = 0
                    End If
                End With
                DestroyThreadData pThreadData
            Elself fSignal Then
                SignalThread .pSA(0), fUnregistered, _
                    fSignaled
            End If
        Next
        If Not fSignaled Then
            'We can turn this off indefinitely if
            'fUnregistered is False because all threads
            'will have been signaled at this point.
            fSignal = fUnregistered
        End If
        If RunningThreads. Count = 0 Then Exit Do
        'Give up the rest of our time slice.
        Sleep 0
        Loop
    End With
End Sub
```

```
Public Function CleanThreads(RunningThreads As Collection,_
  FinishedThreads As Collection) As Long
Dim pThreadData As Long
Dim Iter As Variant
Dim DataKey As String
    With m_Data
        For Each Iter In RunningThreads
            pThreadData = Iter
             . Owner .SA.pvData = pThreadData
            With .pSA(0)
                If .ThreadDone And CBool(.pOutput) Then
                    Set .Controller = nothing
                    If . fKeepData Then
                        FinishedThreads.Add .pOutput, _
                          CStr( .pOutput)
                    Else
                        DestroyThreadDataOutput .pOutput
                    End If
                    'Clear most everything, but leave
                    'ThreadDone so signalling the sleeping
                    'thread without providing new data
                    'will cause it to terminate instead of
                    ' loop.
                    .pOutput = 0
                    . fKeepData = False
                    . fSignaled = True
                    .ThreadSignalPtr = 0
                    . InputData = Empty
                    If CleanThreads = 0 Then
                        CleanThreads = pThreadData
                    End If
                ElseIf CleanThreads = 0 Then
                    If .pOutput = 0 Then
                        CleanThreads = pThreadData
                    End If
```

```
                End If
            End With
        Next
    End With
End Function

Public Sub SignalThreads(RunningThreads As Collection, _
     FinishedThreads As Collection)
    Dim pThreadData As Long
    Dim Iter As Variant
    Dim fUnregistered As Boolean 'Dummy
    Dim fSignaled As Boolean "Dummy
        With m_Data
            For Each Iter In RunningThreads
                pThreadData = Iter
                 . Owner .SA.pvData = pThreadData
                SignalThread .pSA(0), fUnregistered, fSignaled
            Next
        End With
    End Sub
```

13.6.5 无模式窗体和工作线程

VB5 在 ActiveX DLL 中增添了显示无模式窗口的能力。实际上，ActiveX DLL 的许多宿主（host）都需要添加对无模式窗体的支持。载入 ActiveX DLL 的可执行程序通过将它的消息循环交给窗体使用来与 DLL 相协调。这一协调过程并不是无关紧要的，但它没有被文档所记录（除了 VBA 宿主 SDK 中有所提及外）。

如果读者在 DLL 中创建了一个工作线程，因为在工作线程上没有协调的消息循环，所以不能显示一个无模式窗体。读者在 DLL 中只能显示有模式窗体。实际上的情况没有读者所想像那样坏，因为模式只是适用于现在的线程，而不是适用于整个应用程序。两个不同线程上的有模式窗体实际上彼此之间是无模式的。

但现在我要告诉大家一些坏消息。在 VB6 SP3 或更早的版本中，运行时间有一个恼人的错误：App.NonModalAllowed 标志后的数据被工作线程所破坏。这一标志应存储在 TLS 中，但是它实际上是存储为全局数据。这一标志假定最近创建的线程的模式已经设置。结果在创建工作线程之后，标准 EXE 或 ActiveX EXE 不能显示新的无模式窗体。这一错误与线

程如何创建无关。如果在 C++中创建一个新线程并且这个新线程使用 VB DLL 创建的对象时，也会遇到这种问题。

我知道这一问题的一个解决方法，但是它是对调用 EXE 做改变，而不是对 DLL 作出改变。这就要求读者应修补使用该 DLL 的每一个 EXE，这会使读者处于很不利的境地。为了激活一个无模式窗体，必须创建一个线程，这一线程允许无模式窗体能将全局设置返回到它应该在的地方。显式的创建线程需要一个 ActiveX EXE，因此这在标准 EXE 文件中无法做到。下面是这一方法的具体步骤：

（1）将标准 EXE 变为 ActiveX EXE;

（2）在 Project\Properties 中，设置 StartUp Object 为 Sub Main，线程模型为每个对象线程（Thread per Object）；

（3）使用前边介绍的 ThreadCheck 类来找出第一个线程；

（4）向 ActiveX EXE 中添加一个叫做 Dummy 的 MultiUse 类；

（5）添加下面的 ShowNonModal 过程到一个 BAS 文件中（相应的调整 ProgID）：

```
Public Sub ShowNonModal (ByVal Form As Form)
        If Not App.NonModalAllowed Then
           CreateObject "MyApp. Dummy"
        End If
        Form. Show
 End Sub
```

（6）现在调用 ShoWModal New Form1，不要直接使用 Show 方法。

当将一个标准 EXE 变为一个 Thread-per-Object ActiveX EXE 时，存在两个限制条件。首先，在工程内不应有 MDI 窗体。这一限制没有回旋的余地。其次，不能在工程内创建 UserControls。这个限制存在一种突破的方法：使用一个伴随 ActiveX OCX 工程。如果读者的工程内存在 CTL 文件并且需要转换 EXE 文件的类型，读者需要做一些工作。我建议首先创建一个伴随 OCX，然后用一个包含任一控件的窗体来创建一个 Dummy 工程。还应注意要设置 OCX 为二进制兼容。现在，手工编辑原始工程中的 FRM，变换控件的类型，这时 FRM 文件就相当于对控件名字的引用。这样解决看起来有些麻烦，但从长远来看，这要比删除控件然后重新设置每个控件要简单的多。现在有一个好消息：AppModalAllowed 的错误在 msvbvm60.Dll 的 SP4 版中已经得到了修正。

第十四章

VB 中的字符串

我们很难遇到不使用字符串的 VB 程序。毕竟，我们设计程序的目的在于与用户进行交互，或者是产生一些其他的输出，这两种情况都需要与文本打交道。由于越来越多的程序是为了在 Internet 中使用而编写的，字符串在我们所编写的程序中将扮演愈加重要的角色。HTML、XML 和其他的一些 Internet 格式，都包含大量的字符串；并且它们经常需要产生一些长字符串，这使得对字符串的操作的性能成为整个应用程序性能的关键所在。基于提高字符串的性能的目的，本章讲述了 VB 如何定义字符串以及如何与字符串进行交互。尽管我将介绍几种有用的帮助类，但是本章并非意在提供一个字符串操作函数的完整函数库。本章将给读者提供编写出自己的高度优化的字符串操作程序的工具。

在读者能有效地操作 VB 字符串之前，需要懂得 VB 字符串的结构。在 VB 中字符串相当于 C 或 C++中的 BSTR 类型。BSTR 通常被定义成指向无符号短整型数据的指针。实际上，BSTR 是一个带有长度前缀（length-prefix）和内置的 NULL 结束符的 UNICODE 字符串，它通常由 OLE Automation(OleAuto32.Dll)来分配和释放。下面我们对上面一句话的各部分分别进行解释。长度前缀是一块存储字符串的长度的内存单元。当前缀作为字符串指针返回之前，它是一个四字节的长整型数值。这个值等于字符串的字节数，其中不包括用做前缀的四个字节和用做 NULL 结束符的两个字节。长度前缀之后有一系列 UNICODE 字符，每一字符占两个字节。BSTR 结尾是一个 NULL UNICODE 字符（00h）。结尾的 NULL 字符使 BSTR 能用于任何需要以 NULL 结尾的字符串的地方。BSTR 的内存结构如图 14.1 所示。

长度前缀能使一个 BSTR 做一些普通的以 NULL 结尾的字符串不能做到的事情。首先，长度前缀允许不需要一些附加信息就可对字符串进行拷贝。当字符串在跨线程或跨进程的公有接口中使用时，这一点尤其重要。第二，长度前缀使 BSTR 包含内置的 NULL 字符。第三，BSTR 的长度很容易计算。用来返回字符串中字符的个数的 Len 函数执行需要占用大量时间。相比而言执行 LenB 花的时间要少些。（LenB 不需要将字节数除二来得到字符的个数。）这就是为什么将 Len 或 LenB 与 0 相比较是确定字符串是否含有数据的方法的原

因。与“”或 vbNullString 相比较会产生一个串比较调用，这要比简单的从内存中读取字符串的长度开销大得多。

```
Dim strTest As String
'Worst test
```

8	0	66	83	84	82	0

图 14.1 包含字 BSTR 的字符串的内存布局。字符串变量是一个指向 3rd2-bity 位置的指针，它包含了 B(66)

```
If strTest = "" Then Beep
'Not much better
If strTest = vbNullString Then Beep
'Very good
If Len(strTest)= 0 Then Beep
'Best
If LenB(strTest) = 0 Then Beep
```

因为“”占有六个字节的内存单元，所以与“”相比较比与 vbNullString 比较要糟糕；vbNullString 仅仅是一个 NULL 字符串指针。这两种类型在 VB 中所起的功能相同，但是因为 vbNullString 不需要任何存储单元，所以它要更好一些。应该总是使用 vbNullString 而不是“”去清空一个字符串变量。当传递一个真正的 NULL 而不是一个空的字符串时，可能会在使用 API 调用时遇上麻烦。如果必须使用一个需要一个字符串缓冲区而非 NULL 的 API 调用，应该使用 StrPtr 函数去检验传来字符串是否为零，如为 0 则用“”来代替 0。使用 StrPtr 是区分 vbNullString 和“”的惟一方法。

```
If StrPtr(strData) = 0 Then strData = ""
```

因为在 C 语言中“NULL”总是表示数值 0，所以术语“null”的各种转化形式可能会令人费解。VB 有五个内置的关键字和常量包含 null。首先，VB 的 Null 关键字表示一个带有无效数据的变量。这种变量的 VarType 是 vbNull，其值为 1。运行时间函数 IsNull 检查变量的 VarType。一个初次接触 VB 的程序员常犯的错误是用 vbNull 来代替 0、Nothing、vbNullString 或其他一些概念。其他的 null 的转化形式都是字符串。VbNullString 是一个值为 0 的字符串常量，而 vbNullChar 是一个包含 NULL 字符的长度为 1 的字符串常量。

在 VB 中没有完全大写的 NULL 形式。当读者看到 NULL 这种形式，就可以认为它是在 C 中一样的 0 值。一个 NULL 字符是一个数值值为 0 的字符，而以 NULL 结尾字符串是一个字符串，它的结尾字符为 NULL。

14.1 UNICODE 转换

UNICODE 字符串标准规定每个字符占用两个字节，而在 ANSI 标准中一个字符只占用一个字节。ANSI 和 UNICODE 的基本区别是在 ANSI 中解释 ANSI 字符串需要一个代码页，以及同一个字节在 Cyrillic 代码页和在西欧代码页中表示不同的字符。UNCODE 标准将代码页映射为单字符行；为了标识一个 UNCODE 字符，读者无须指定一个代码页。

ANSI 与 UNCODE 另一个不同之处在于 UNCODE 中所有的字符都是同长度的。一些 ANSI 代码页使用某字节作为首字节（lead bytes）。首字节和尾字节(trail bytes)一起决定实际字符。所以尽管 ANSI 字符可能由一个字节组成的，但它也可能是由多个字节组成的。ANSI 有时被称为“多字节字符集”，而称 UNCODE 为“宽字符”。读者在 WIN32 API 中可以看到这一点，在 WIN32 中有 MultiByteToWideChar 和 WideCharToMultiByte 函数。尽管 VB 以 UNCODE 形式来存储字符串，它通常也把 UNCODE 字符转换成 ANSI 和 vice-versa 字符。任何自动转换都是根据当前系统的代码页进行的。

编写真正国际化的应用程序（那就是，一个无论机器操作系统采用何种语言，都可以运行于多种语言之上的应用程序）并不是一个无关紧要的问题。这一问题的核心是，要做到国际化就要求读者对用来完成 ANSI/UNCODE 转换的代码页进行显式控制。通常读者在代码的语言元素中进行显式的代码转换，但是 VB 的窗体软件内在的使用 ANSI，它没有提供指定代码转换的代码页的机制。无法显式的控制代码页就意味着即使读者安装了一个语言兼容性字体，如果不使用一些烦人的循环语句，也不能将文本赋给 Label 或 TextBox 控件。参考 Michael Kaplan 的一篇著作可以找到使用 VB6 来创建国际化应用程序的更多信息。

隐式的字符串转换在 VB 中是很常见的，读者通常不知道已经发生了字符串转换。但是，字符串转换是一种费时的操作，读者应该尽量的避免使用它。读者可以采取几步简单的操作来减少字符串转化的次数。第一步是使用 Asc 和 Chr$函数的 W 版本。Acs 在返回之前首先将固有的 UNICODE 字符串转化为 ANSI 字符串；AscW 只是读取当前存储在字符串中数值。同样的，Chr$将由读者给的数值转化为 UNICODE 数值。结果 AscW 和 ChrW$要比 Asc 和 Chr$运行快得多。

读者应尽量的使用字符串函数的$版本。如果一个函数存在$版本，那它的非$版本返回一个变体而不是字符串。尽管 VB 编译器足够智能，能够在将变体函数的返回值赋给一个字符串变量时，避免复制这个字符串。但是在使用非$函数时，仍存在着大量的时间花费并且生成了大量的代码。如果读者认为$ 版本中一些附加的字符很麻烦，请看一下非$ 版本生成的代码，就会明白使用$版本的好处了。在对象浏览器内查找$就可以得到一组$ 函数的列表。

VB 做自动的 ANSI/UNICODE 转换的最常见的地方是在将一个字符串或固定长度的字符串变量传送给 Declare 函数的时候。为了在内部（内部通常是 UNICODE 字符串）无缝的

处理外部的字符串（外部通常是 ANSI 字符串），VB 将所有的 API 调用都看作是符合 ANSI 的。当读者要做 ANSI API 调用时，从编码的角度来讲就简单了，因为读者无需作任何工作就可以在 VB 中作 ANSI API 调用，但是从性能的角度来讲，在运行时就多了字符串转换的花费。如果编写一个完全在 Windows NT 或 Windows 2000 下运行的程序，由于这两种操作系统能够充分的支持 UNICODE API，在 API 调用时作字符串转换只是对时间的浪费。

隐式转换的一个不利之处是读者无法使一个 Declare 语句将所有的字符串都看作 UNICODE 并将它们置之不理。VB4 的文章都采用了一种特别麻烦、复杂的调用 UNICODE API 函数的方法。这种技术涉及在字符串和 Byte 数组之间传递数据，还包含将 Byte 数组的第一个元素 ByRef 传递给 API 调用。这种技术需要的多个字符串之间进行复制好像并不是太坏的事，但是许多人在这里出了问题，因为他们在将字符串复制给 Byte 数组后，没有在尾部添加 NULL 字符。尽管 VB 允许将一个字符串传递给一个 Byte 数组，但它并不复制结束符 NULL。许多人在做 UNICODE 调用时实际上传递的是没有 NULL 结束符的字符串。VB5 和 VB6 允许读者将 ByVal As String 类型转变为 ByVal As Long，并且它传递 StrPtr(String) 而不是对字符串作一个深度拷贝。

```
'Deprecated method using a Byte array.
Public Declare Function FindWindowW Lib "user32" ( _
    ByVal lpClassName As Long, lpWindowName As Byte) AS Long
Public Function LocateWindow(StrWindowName As String) As Long
Dim bTmpBuf( ) As Byte
    'The vbNullChar is required. This actually generates
    'two copies of the String; one to concatenate
    'the NULL character and one to copy into the byte array.
    bTmpBuf = strWindowName & vbNullChar
    LocateWindow = FindWindowW(0, bTmpBuf(0))
End Function

'The preferred mechanism with StrPtr.
Public Declare Function FindWindowW Lib "user32" ( _
    ByVal lpClassName As Long, _
    Byval lpWindowName AS Long) As Long
Public Function LocateWindow(strWindowName As String) As Long
    LocateWindow = FindWindowW(0, StrPtr(strWindowName))
End Function
```

很明显，使用执行起来特别快的 StrPtr 函数要比复制字符串两次好得多。但是，As String 参数只是问题的一半。读者可以直接将字符串传递给 Declare 函数，也可以将它作为结构中的

一个字段来传递。StrPtr 不适用于内置的字符串或固定长度的字符串。为了不作改变的传递一个结构，可以简单的将 ByRef As MyType 声明变为按值(ByVal) As Long 并放置一个 VarPtr 在这个结构上。参见第十五章中对使用带有用于进行 UNICODE API 调用的显式字符串类型（而不是 VarPtr）的类型库函数声明的讨论。

读者还可以在作 ANSI API 调用时使用 StrPtr 以防止 VB 重复的对多个调用使用同一个字符串转换。例如，如果调用 Declare 函数，这个函数在循环中不断的接收同样 ByVal ANSI String，这时使用 StrConv 转换字符串并声明参数为 ByVal As Long，然后使用 StrPtr 来做调用是一种更加有效的方法。

```
Dim strANSIName As String
Dim pANSIName As Long
    strANSIName = StrConv(strName,VbFromUnicode)
    pANSIName = StrPtr(strANSIName)
    'You can now pass pANSIName multiple times with a
    'single conversion
```

14.2 字符串的分配

字符串同对象和指针一样都是指针类型，因此字符串变量包含一个指向一块有 BSTR 特征的内存单元的指针。所有的 BSTR 都是由 OleAut32.Dll 分配（借助于 SysAllocString、SysAllocStringLen 或 SysAllocStringByteLen）并且使用 SysFreeString API 来释放的。在分配和释放字符串时，VB 运行时间（runtime）使用 API 调用来进行字符串管理，因此读者通常无需自己调用它们。但是如果传递给读者一个指向以 NULL 结尾的字符串的指针并且需要在 VB 中读取这个指针，读者可以使用这些 API 调用来分配自己的字符串。所有的 OleAut32 字符串 API 都在类型库 VBoostTypes 中进行了声明。

```
'Create a VB-readable String from a UNICODE pointer.
Dim strReadable As String
    StrReadable = SysAllocString(pUNICODEString)
```

要从一个 ANSI 指针中得到一个可读的字符串需要做一些工作，但是如果读者指定字符串的字节长度并让 VB 来转换字符串那就会变得相当容易。

```
'Create a VB-readable String from an ANSI String pointer.
Dim strReadable As String
```

```
strReadable = StrConv( _
   SysAllocStringByteLen(pANSIString, _
      lstrlen(pANSIString)), _
   VbUnicode)
```

VB 经常假定字符串变量的存储单元是可写的。这对我们来说既是好事又是坏事。说它是好事是因为这样就可以对字符串进行动态的修改，而无需分配新的存储单元（下一节中将讲到），但这也意味着 VB 编译器没有只读字符串这一概念。缺少只读字符串会带来两个直接的不良后果。首先，当读者将一个常量字符串传递给一个 ByRef 字符串参数的时候，VB 必须复制这个字符串。其次，VB 自动的复制通过 ByVal 字符串参数接收到的字符串，以确保 VB 已经将数据写入了这个参数。但是，如果复制一个常量字符串之前就对它作出修改，改变字符串中的数据会导致编译后的 EXE 文件中发生冲突。为了安全起见，VB 就将传递来的数据复制到一个局部变量中，读者可以对这一变量进行编辑而无需改变传递来的字符串。

使用 ByRef 字符串是避免对同一工程内的字符串作字符串拷贝的最简单的方法。但是，如果读者实现一个有可能接收长字符串作为它的参数的接口，而没有方法避免作字符串拷贝。因为读者不得不使用 ByVal String。为了处理这种情况，读者需要实现这个接口的 VB 化版本，这个版本接收 ByVal 字符串而不是 ByRef 字符串并且只是“借用”字符串指针。这可以使用一个轻量对象来完成，其中轻量对象在 VB 有机会调用 SysFreeString 之前使用结构终止代码来将字符串变量的值设为 NULL。

当数据没有跨线程或跨进程排队时，ByRef 传递一个字符串是最快的方法。但是 ByRef As String 翻译为[in,out]BSTR*，并且它告诉排队引擎（engine）在调用完成后将字符串复制到目的线程和调用线程。建议对已排队字符串使用 ByVal 传递以消除将数据复制回调用线程的开销。如果需要使用源于已排队、同线程的用户的公共对象，通过使用 PowerVB 后期绑定类型库修改符来将 ByRef 字符串上的排队标签变为[in]BSTR*，可以得到 ByRef 字符串的线程内好处而无需增加排队的负担。

```
'VTable Structure declarations omitted

Public Type StringRef
   pVTable As Long
   pthisObject As IUnknown
   Ref As String
End Type

Public Sub InitStringRef( _
```

```
    SR As StringRef, ByVal pString As Long)
        'Initialize the VTable (omitted)
    With SR
        .pVTable = m_pVTable
        VBoost.Assign .pThisObject, VarPtr(.pVTable)
        VBoost.Assign .Ref, pString
    End With
End Sub

'QueryInterface and AddRef omitted
Private Function Release(This As StringRef) As Long
    'Clear the String pinter before VB gets hold of it.
    VBoost.AssignZero This.Ref
End Function
```

```
'Calling code
Public Sub IFoo_DisplayString(ByVal pString As Long)
Dim SR As StirngRef
    InitStringRef SR, pString
    MsgBox SR.Ref
End Sub
```

对于所有字符串来说（在我的测试中，少于 30 个字符的字符串除外），传递 StrPtr 给 ByVal String 参数并且初始化 StringRef 结构要比传递一个字符串给 ByVal 字符串参数并且允许进行字符串拷贝快得多。即使对于很短的字符串，StringRef 也不会比 ByVal 传递慢百分之四十以上；但是，对于长的字符串，StringRef 要比 ByVal 传递快好几个数量级。字符串拷贝存在的问题在于拷贝的花费是字符串长度的函数，而 StringRef 的花费是固定的。在使用 StringRef 时应该十分小心。调用代码将指针传递给一个有效的 BSTR，无论如何也不能改变 StringRef.Ref 中的数据。

优化字符串性能的最好方法是首先尽量减少字符串的数量，然后将分配的字符串的大小降至最小。如果应用程序运行在一个多线程的服务器环境中，应当特别注意分配的字符串的数量。BSTR 是由 OLE Automation 使用 COM 分配器分配的，它随之从缺省的进程堆中得到分配给它内存空间。这个进程堆是一个公用资源，因此在每一时刻只有一个线程可以使用它。分配的字符串越多，线程在进行字符串（或其他）分配等待的可能性就越大。

我们做以下一些事情来尽可能的减少要分配的字符串的数量。第一件事情是要清楚拷贝来的字符串是否还会使用。如果不再使用这一字符串，可以简单的使用 VBoost.AssignSwap 函数得到这个字符串。AssignSwap 以两个四字节变量的形式来交换这个字符串，因此在两个

字符串变量之间做交换只是交换字符串的指针，而没有分配数据。例如，Property Let 函数右侧的参数是 ByVal 传送的，在传送时自动的给出了字符串的局部拷贝。实际上，即使声明它为 ByRef 传送，VB 也会传递字符串的一个拷贝，因此最好使用 ByVal 方式。这样可以使用字符串的一个拷贝而不会带来任何副作用。

```
Private m_strName As String
Public Property Let Name(ByVal RHS As Stirng)
    VBoost.AssignSwap m_strName, RHS
    'The previous m_strName value will be freed
    'when RHS goes out of scope.
End Property
```

按上面的方法使用字符串可以避免一些不必要的字符串拷贝，这就达到了减少分配的字符串的数量的目标。减少分配的字符串的第二种方法是使用 Join 函数，这是 VB6 中新引入的一个函数。Join 函数使用分配来将一个字符串数组和一个字符串合并起来。使用 Join 函数来合并一些字符串要比在一个循环中依次添加字符串要更有效率一些。在 VB 中写出坏的字符串合并函数是很容易的事。下面一个程序先从一个文件中读取一些行，然后使用标准的字符串合并函数来重新构建一个字符串。

```
Public Function ReadFile(strFileName As String) As String
Dim fNum As Integer
Dim strLine As String
    fNum = FreeFile
    Open strFileName For Input As #fNum
    'Note: ReadFile = Input(LOF(fNum),fNum)is sufficient,
    'here, but doesn't help with a concatenationHdemo.
    Do Until EOF(fNum)
        Line Input #fNum, strLine
        ReadFile = ReadFile & strLine & vbCrLf
    Loop
    Close #fNum
End Function
```

虽然读者已经知道这些代码性能不好，但是仍然可能对它的性能差的程度感到惊讶。对 VBoost.Bas（大小大约为 110k）运行这一个循环大约需要 21 秒。使用我下面将要讨论的另一种合并技术可以将时间减为十分之一秒，性能提高了足足 21000 倍!

这个程序存在的问题是当字符串变得更大时分配起来要更加费时。如要在一个长的字符串上添加小的字符时，实际上是在为每一个&操作符分配一个大的字符串。这个用来测试的程序（VBoost.Bas）近似有 2800 行，函数每行创建两个字符串，这样总计要分配 5600 次。如果首先合并较小的字符串然后将它们添加到总的字符串上，可以减少分配的数量并提高函数执行的速度。读者可以通过添加括号来确定首先合并哪一个字符串，以提高 VB 执行的性能。这一对合并所做的较小的改变可以将时间由 21 秒减为 8.5 秒。这种速度也是不可接受的，但它表明了分配区大小对性能的影响。添加括号并不改变分配的数量，但它对性能还是有较大的影响。

```
ReadFile = ReadFile & (strLine & vbCrLf)
```

OLE Automation 对小的字符串有字符串缓冲，因此许多小的字符串都是从先前分配的字符串的缓冲池中取出的，而不是从一个新的系统分配区中取出的。但是，字符串缓冲池的大小不能大于 64k。字符串缓冲池保证无论读者的代码的有多糟糕，在处理小的字符串时程序性能都不会太坏。但是，合并时的花费曲线在字符串比缓冲区大的这一点是不连续的。一些处理 30k 或甚至 60k 的字符串时还可以的代码的性能会在字符串长度超出缓冲器长度时突然变坏。在字符串变得更大时，完成字符串合并需要更系统的方法。

VB6 中新引入的 Join 函数是减少字符串合并的数量的一个很强大的工具。Join 函数使用一个调用来合并字符串数组，在数组的每一个元素中间有选择的插入一些限范围符。Join 首先越过前面的字符串来确定最后一个字符串的大小，然后在拷贝这一字符串数据之前做一次数据分配。这样就会大大的减少合并字符串所需的分配的数量，但是 Join 函数本身并不能完全解决合并问题。

Join 函数有两个缺点。首先，读者在开始的时候，并不知道分配的数目，这样就难以为数组选择一个合理的大小。其次，函数并不带有一个 Count 参数，这就要求读者 ReDim Perserve 这个数组到一个较小的或者指定 vbNullString 为 Delimiter 的参数，这样就不会有别的限范围符序列添加到得到字符串上。可以使用 VarPtrStringArray 来代替 ReDim Preserve 以获得对数组结构中的 cElment 字段的引用，然后临时的改变 Join 调用的生存期的值。这些可在下面的程序段中表现出来。

```
Dim Strings () As String
Dim pElemCount As Long
Dim cElems As Long
Dim strTotal As String
    with VBoost
        pElemCount = .UAdd ( _
          .Deref(VarPtrStringArray (Strings)), 16)
        .AssignSwap ByVal pElemCount,cElems
        strTotal = Join(Strings,vbCrLf)
```

```
        .AssignSwap ByVal pElemCount,cElems
    End With
```

本书提供了一个叫做 SmartContact 的帮助类来帮助读者优化合并的性能而无需直接处理 Join 函数。SmartContact 用两步来处理 Join 函数。但读者调用 AddString 或 AddStringSteal 方法时，SmartContact 将这个字符串放于 m_SmallStrings 数组中。当数组已满或者数组的总长度超过一定的范围时，就会使用 Join 函数来合并数组，得到的数组存于一个中间的数组 m_MediumStrings 内。在调用 GenerateCurrentString 方法之前，m_MediumStrings 根据需要（使用 ReDim Perserve）一次增加几项。这个方法对 m_MediumStrings 数组中的所有字符串最后执行一次 Join，结果是一个较少数量的小分配区、有限数量中等大小分配区和一个大分配区的合并方案。下面的代码中包含 ReadFile 函数的改进版本上的 SmartContact。这个版本在十分之一秒内载入测试文件；这与前面版本中的 21 秒和 8.5 秒相比要优越的多了。

```
Public Function ReadFile(strFileName As String) As String
Dim fNum As Integer
Dim strLine, As String
Dim Concat As SmartConcat
    FNum = FreeFile
    Open strFileName For Input As #fNum
    Set Concat = New SmartConcat
    Concat.Separator = vbCrLf
    Do Until EOF(fNum)
        Line Input #fNum, strLine
        Concat.AddStringSteal strLine
    Loop
    Close #fNum
    Concat.AddString vbNullString
    ReadFile = Concat.GenerateCurrentString
End Function
```

14.3 作为数值的字符串

字符串是一组字符，其中每一个字符都是数值。作为一个通用的编程原则，读者应该将字符串看作数值数组而不是自治单元，这样可以使字符串程序达到最优的性能。不幸的是，VB 没有提供将字符串看作整型数的数组的内置机制。这迫使 VB 编程人员使用字符串

分配来完成“可能为”数值型的操作。本节将把 VB 字符串看作整型数的数组，它允许读者应用对数值而不是对普通的字符串变量进行操作的字符串算法。我将主要依赖第二章中介绍的 SafeArray 技术和第八章介绍的 ArrayOwner 轻量技术。

Visual Basic 在正式的场合是使用 Asc 和 AscW 函数来提供字符串的数量。这些函数读出字符串中第一个字符的数值；Asc 返回 UNICODE 数值，Asc 返回 ANSI 数值。AscW 在一个优化的 Select Case 语句中用来确定字符串的第一个字符时是非常有用的。下面的两个程序段在函数功能上是相当的，但是完成这些功能的代码是完全不同的。

```
'Snippet1
Select Case Left$(TestString, 1)
   Case "@"
      'etc
   Case "!"
      'etc
   Case Else
      'etc
End Select

'Snippet2
Select Case AscW(TestString)
   Case 64 '@
      'etc
   Case 33 '!
      'etc
   Case Else
      'etc
End Select
```

在程序段 1 中，VB 首先分配一个新的字符串，然后遍历一系列 If/Else 语句，在每个 Case 语句中比较字符串。代码段 2 中没有字符串分配，并且 VB 使用整型常量值来产生一个切换表。这要比程序段 1 生成的 If 语句要更有效率。总之在各个方面程序段 1 都要稍逊一筹。注意到在设计模式中任何时候都可以在调试窗口内键入?AscW(“@”)来得到正确的字符串值。虽然作注释是可选择的，但是读者在下次阅读代码时会发现注释的用处。

AscW 能很好的处理字符串中的第一个字符，但是它不能处理后面的字符。使用后面的字符的数值的最简单的方法是在一个字符串拥有的内存内指向一个整型数组。具体可见图 14.2。下面的类 FastMid 能生成一个单字符数组，读者可以动态的更新一个字符串以保

持另一个字符串中字符值，这中间不需要重新分配。读出单个字符通常需要调用Mid$函数，这一函数为每一个字符分配一个新的字符串。但是 FastMid 允许一次一个字符的遍历字符串，不需要为每一个字符分配一个新的字符串。当使用长字符串时，这要节省许多时间。

```
'Requires: ArrayOwner.Bas or ArrayOwnerIgnoreOnly.Bas
Private Type OwnedInteger
    Owner As ArrayOwner
    pSA( ) As Integer
End Type
Private m_FullString As OwnedInteger
Private m_SingleChar As OwnedInteger

Public Function GetMidString(FullString As String) As String
    GetMidString = Sting$(1,0)
    m_SingleChar.Owner.SA.pvData = StrPtr(GetMidString)
    With m_FullString.Owner.SA
        .pvData = StrPtr(FullString)
        .cElements = Len(FullString)
    End With
End Function
Public Sub SetMidPosition(ByVal Index As Long)
    m_SingleChar.pSA(0) = m_FullString.pSA(Index)
End Sub
Private Sub Class_Initialize()
    With m_FullString
        InitArrayOwner .Owner, 2, 0
        'FastMid is 1-based to map directly to the Mid$
        'function.
        .Owner.SA.lLbound = 1
    End with
```

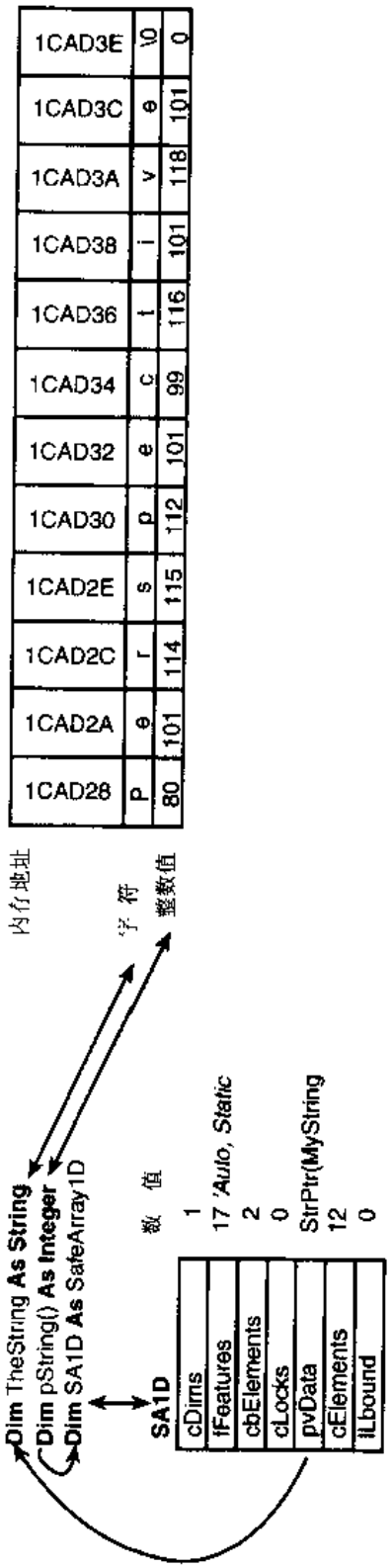

图 14.2　通过一个整型数组中的数值，使用 SafeArray 结构来对字符串中的字符进行操作。在这种情况下，StrPtr(TheString)=VarPtr(pString(0))

```
InitArrayOwner m_SingleChar.Owner, 2, 0
End Sub
```

```
'Use of FastMid, gives the same output as
'Debug.Print strWalkMe.
Dim strWalKMe AS String
Dim strMid As String
Dim i As Long
    With New FastMid
        strMid = .GetMidString(strWalkMe)
        For i=1 To Len(strWalkMe)
            .SetMidPosition i
            Debug.Print strMid;
        Next i
        Debug.print
    End With
```

很明显，如果在读者正在观察的字符串或 MidString 释放或重新分配之后调用 FastMid.SetPosition，程序就会立即崩溃。但是，只要不调用 SetMidPosition，FastMid 要比它指向的字符串存活的时间长一些。FastMid 通过将字符串上的一个字符看作整数数组的一个入口，就可以不使用 VB 的任何字符串函数来改变字符串。

第二个数值处理的例子是使用附加到字符串上的整型数组来创建一个简单的 Tokenizer 类。这个类带有一些单字符限幅符号，这个符号将字符串分为几个字段。Tokenizer 类支持三种类型的限幅符号：从后面的集合中去除的限幅符号、当作限幅符号来使用但仍保留在后面的字段集中的字符、表示处理应当结束的字符。Tokenizer 类可用来解析简单的等式。它对引用的字符串的识别功能是内置的。我首先介绍它的用法，然后给出这一个类的代码。

```
'A snippet to tokenize a line of Basic code and strip all
'spaces.
Dim strFields () As String
Dim iStoppedAt As Integer
Dim iFields As Integer
With New Tokenizer
    .SetDelimiters " ",  "^&:;*,/\-+=<>.()", "'"
    For iFields = 0 To _
        .TokenizeLines(strLine, iStoppedAt, strFields) - 1
```

```
        Debug.Print strFields(iFields)
    Next iFields
    If iStoppedAt Then _
        Debug.Print Mid$ (strLine, iStoppedAt)
End With
```

```
'Tokenizer Class from Tokenizer.cls
'Requires: ArrayOwner.Bas or ArrayOwnerIgnoreOnly.Bas
Private Type OwnedInteger
    Owner As ArrayOwner
    pSA() As Integer
End Type

'The complete set of tokens
Private m_Tokens As String
'An Owned integer array pointing to m_Tokens
Private m_RefTokens As OwnedInteger
'The Last end-of-line token: stop processing if tokens
'before this are hit.
Private m_LastEOLToken As Integer
'The last token that is a delimiter we should keep
'All tokens after this one are tossed out.
Private m_LastkeepToken As Integer.
'An owned integer array pointing to the line being processed.
Private m_RefLine As OwnedInteger
Private Const iDoubleQuote As Integer = 34

Public Sub SetDelimiters(TossDelimiters As String, _
    KeepDelimiters As String, StopAt  As String)
        m_LastEOLToken = Len(StopAt)
        m_LastKeepToken = m_LastEOLToken + Len(KeepDelimiters)
        m_Tokens = stopAt & KeepDelimiters & TossDelimiters
        With m_RefTokens.Owner.SA
            .pvData = StrPtr(m_Tokens)
            .cElements = Len(m_Tokens)
        End With
```

```
End Sub
Public Function TokenizeLine(strLine As String,_
   iStoppedAt As Integer, StrReturn() As String) As Integer
Const RedimIncrement = 20      'The grow size of the return array
Dim iRetArrSize As Integer     'The current size of the return
                               'array
Dim iPos As Integer          'The current position in strLine
Dim iMaxLen As Integer       'Cache for Len(strLine)
Dim iTestChar As Integer       'Character in the line being
                               'tested
Dim iStarPos As Integer        'Starting position of current
                               'field
Dim iFieldLength As Integer      'Length of current field
Dim iToken As Integer
Dim cToken As Integer
Dim fHaveNexToken As Boolean

   TokenizeLine = 0

   iMaxLen = Len(strLine) 'Cache the length
With m_RefLine
   With .Owner.SA
      'Plug the String Into our SafeArray structure
      .pvData = StrPtr(strLine)
      'Go to +1 so trailing NULL terminator is in array
      .cElements = iMaxLen +1
   End With
   iStoppedAt = 0
   cTokens = Len(m_Tokens)

   'The main processing loop
   Do While iPos < iMaxlen
      iStartPos = iPos 'Keep the current position
      iTestChar = .pSA(iPos)

      'See if we've hit any tokens
```

```
If fHaveNextToken Then
    fHaveNextToken = False
Else
    For iToken = 0 To cTokens - 1
        If m_RefTokens.pSA(iToken) = _
            iTestChar Then
            Exit For
        End If
    Next iToken
End If
If iToken = cTokens Then
    'Not a token character: check for double -
    'quote and next token
    If iTestChar = iDoubleQuote Then
        'Consider quoted Strings a single field.
        Do
            iPos = iPos + 1
            If iPos >=iMaxLen Then
                Erase strReturn
                Err.Raise 5
            Else
                iTestChar = .pSA(ipos)
                if iTestChar = iDoubleQuote Then
                    'Check for 2 double quotes.
                    'Note that this won't look
                    'beyond the end of the array
                     'because the array length for
                     'the line includes the
                     'trailing Nu11.
                     If .pSA (iPos + 1) = _
                       iDoubleQuote Then
                           iPos = iPos + 1
                     Else
                       Exit Do
                     End If
                End If
```

```
                End If
            Loop
            iPos = iPos +1
            iFieldLength = iPos - iStartPos
        Else
            'Walk until the next token is hit .
            Do
                iPos = iPos + 1
                IF iPos = iMaxLen Then Exit Do
                iTestChar = .pSA(iPos)
                For iToken = 0 To cTokens - 1
                If m_RefTokens.pSA(iToken) =_
                        iTestChar Then
                            fHaveNextToken = True
                            Exit Do
                    End If
                    Next iToken
            Loop
            iFieldLength = iPos - iStartPos
        End If
    ElseIf iToken <m_lastEOLToken Then
        iStoppedAt = iPos + 1
        Exit Do
    ElseIf iToken <m_LastKeepToken Then
        iPos = iPos + 1
        iFieldLength = 1
    Else 'Toss token
        iFieldLength = 0
        iPos = iPos + 1
    End If

    If iFieldLength Then
        'Return this field
        'Grow the array if necessary
        TokenizeLine = TokenizeLine + 1
        'Increment the size of the array in blocks to
```

```
        'avoid Redim Preserve every time
        If TokenizeLine > iRetArrSize Then
            iRetArrSize = iRetArrSize + RedimIncrement
            ReDim Preserve strReturn(iRetArrSize - 1 )
        End If
        'Save the field to return.
        strReturn (TokenizeLine -1) = _
            Mid$(str1ine, iStartPos +1,iFieldLength)
        End If
    Loop
End With
If TokenizeLine Then
    'Remove unused array elements.
    ReDim Preserve strReturn(TokenizeLine - 1)
Else
    Erase strReturn
End If
End Function

Private Sub Class_Initialize()
    InitArrayOwner m_RefTokens.Owner, 2 , 0
    InitArrayOwner m_RefLine.Owner, 2 , 0
End Sub
```

Tokenizer 类在 IDE 中运行良好，但是在一个复杂的可执行文件中运行之前，不会完全看到它性能上的好处。为了使程序达到最高的性能，一定要在编辑选项对话框的“高级”部分禁止整数上溢和数组边界检查。读者可以对这个程序作许多改进，这只是一个例子程序。但是，它也确实表明了将一个整数数组指向一个字符串缓冲区并编辑这个字符串缓冲区是很容易的事情。读者现在可以在需要时将标准的 VB 代码转换为数值处理的程序，这样就减少了分配的数量、字符串比较和其他费时的字符串操作执行的次数。这会大大提高读者处理字符串的能力。

第十五章

类型库和 VB

类型库构成了 VB 使用的二进制描述语言，这种语言可以用来使用对象并描述由类型库产生的对象。在对类型库所包含的对象进行编译之前，读者的工程应包含一个对类型库的引用，这个类型库定义了读者所要使用的类型。当创建一个 ActiveX 组件时，VB 也会产生一个类型库。简而言之，在 VB 中没有类型库，读者一行代码也运行不了，更不能创建用户定制的 ActiveX 组件。

VB 集成开发环境没有给出在底层察看类型库的方法，更不用说创建一个新的类型库或者编辑 VB 为读者的工程创建的新类型库了。在绝大多数情况下，这种只能在高层察看类型库的限制（使用对象浏览器）是有好处的。作为一种良好的高级语言，VB 应该尽可能的隐藏它实现的技术细节。但是，正如理解 COM 对象的二进制输出能大大提高 VB 的性能一样，对 VB 使用和创建的类型库加以控制可以提高读者使用 VB 来编程的能力，有助于读者写出更专业的程序。

我们对类型库的讨论包含四个方面。首先，我们将简略的看一下 VB 为读者产生的类型库。其次，看一下在工程内创建用户定制的类型库有何好处。在此，我们将看一下二进制兼容性文件，并且将讨论一下何时以及为何对其做出修改。最后，将介绍修改VB为ActiveX组件产生的类型库的原因。

本章的目的是讨论与 VB 直接有关的类型库。它不是类型库疑难解释工具的参考手册，就像 MkTypLib 或 MIDL 那样。这些都是很好的 MSDN 文档，中间有许多例程。为了补充随 Visual Studio 一同发行的类型库编译器，本书的光盘中包含了三个类型库处理插件，可以用它们来完成本书所讲述的所有操作。PowerVB 类型库编辑器 (EditTlb.Dll)允许在 VB 的集成开发环境中创建和编辑类型库。PowerVB 二进制兼容性编辑器使读者对二进制兼容性文件可以做有限的修改。PowerVB 后期构建类型库修改符(PostBuild.Dll)允许修改 VB 产生的类型库。本章将集中讨论使用这些工具的好处，而不是如何使用这些工具，读者可以在本书所附带的光盘中找到使用这些工具的文档。

对类型库进行编辑在所有的 Win32 开发平台上都是可行的。但是编辑二进制兼容文件和可执行文件需要完全支持资源替代 API 集，这只有在 NT 4.0 和 Windows 2000 操作系统上才能得到满足。

15.1 VB 产生的类型库

从字面意义上可以看出，类型库包含一些类型。但是在这些类型都完全定义之前，类型库并不表示任何意义。就像 VB 和其他的一些编程语言中那样，类型库是由一些原子类型组合起来定义函数、构成复杂的类型。这些函数可以以一种松散的方式组合起来（模块），也可以以一种严格的方式组合起来（接口）。这些定义类似于读者在普通的编程语言中所做的那样，不同的是类型库只是产生一个描述而不是实现。为了在类型库描述符和可执行文件所提供的实现之间提供联系，类型库应该包含一个叫做 coclass（Component Object Class，组件对象类）的辅助组合，这是一些接口的组合。无论 COM 对象是由 VB 创建的还是由其他的语言创建的，它都定义为 coclass 类的实现。为了描述一幅图像，类型库定义了一种枚举类型（与 VB 中的同名类型相对应）和别名(alias)类型，alias 类型在 VB 中没有对应的类型。

表 15.1 类型库的布局。每一列中的元素都由其左边列中的项来进行定义。加*则表示该项可被加入至首列中的定制类型中，从而产生一个螺旋类型系统

Step 1	Step 2	Step 3	Step 4
原有 Types	Functions	Interfaces*	CoClasses*
定制 Types	Functions	Modules	
	Constants		
	Records*		
	Enums*		
	Aliases*		

VB 语言的元素映射到相应的类型库元素。VB 类型是类型库类型的一个真子集，因此 VB 语言的所有描述元素在（与实现元素相对）类型库中有一个相应的入口。其他的一些类型库元素可以被 VB 所使用，尽管在 VB 中这些元素没有进行定义。

下面给出一个例子。下面一小段 VB 程序是 MultiUse 类的一部分，这就意味着不仅这个类而且 Public Enum 和 Public Type 入口都要写入到生成的类型库中。对象定义语言（ODL）也可表示相应的结构。在 VB 和 ODL 例程中，可以清楚的看到从原子类型到用户定制类型到函数的发展。从函数到接口再到类的发展只有在 ODL 中才可见，因为 VB 自动的为类产

生了一个缺省接口和一个事件接口。

程序清单 15.1 一个 VB 类模块及等价的 ODL

```
'VB code in a MultiUse class module. Several miscellaneous
'types are shown for demonstration purposes.
Public Enum Style
    All = 0
    Style1 = 1
    Style2 = 2
End Enum
Public Type Entry
    Name As String
    Data(0 To 2) As String
End Type
Public Type Entries
    Style As Style
    Entries () As Entry
End Type

'Event declarations
Event EntriesChanged()

'Methods and properties
Public Function GetEntries( _
    ByVal Style As Style, Entries () As Entries) As Boolean

End Function
```

```
//The equivalent ODL

[
    uuid(0983F45D-C5F6-11D3-BC23-D41203C10000),
    version(1,0)
]
library EntriesProject
{
    importlib( "stdole2.tlb");
```

```
        typedef [uuid(0983F460-C5F6-11D3-BC23-D41203C10000)]
        enum EntryStyle {
            All = 0,
            Style1 =1,
            Style2 = 2
    } EntryStyle;

    typedef  [uuid(0983F461-C5F6-11D3-BC23-D41203C10000)]
Struct Entry {
    BSTR Name;
    BSTR Data[3];
} Entry;

typedef [uuid(0983F462-C5F6-11D3-BC23-D41203C10000)]
struct Entries {
    EntryStyle Style;
    SAFEARRAY(Entry) Entries;
} Entries;

[
    uuid(09832F45E-C5F6-11D3-BC23-D41203C10000),
    hidden, dual, nonextensible
]
interface _EntriesClass : IDispatch {
    HRESULT GetEntries(
                    [in] EntryStyle Style,
                    [in,out] SAFEARRAY(Entries) * Entries
                    [out, retval] boolean * retVal);
};

[
    uuid(0983F463-C5F6-11D3-BC23-D41203C10000),
    hidden,nonextensible
]
dispinterface _EntriesClass{
    properties:
    methods:
            [id(1)]
```

```
        void EntriesChanged();
};
[
    uuid(0983F45F-C5F6-11D3-BC23-D41203C10000)
]
coclass EntriesClass {
    [default] interface _EntriesClass;
    [default,source] dispinterface _EntriesClass;
    }
}
```

表 15.2 VB 类型和元素与对应的类型库元素之间的映射

VB 类型	类型库等价类型
Integer	short
Long	long
Single	float
Double	double
Currency	currency
Date	DATE
String	BSTR
Object	IDispatch*
Boolean	boolean
Variant	VARIANT
Byte	unsigned char
Array of type x as a Parameter	SAFEARRAY(X)*
Type/End Type	record
Variable-size Array in a UDT	SAFEARRAY(x)
Fixed-Size Array in a UDT	x FieldName[2]
Declare and Const in a standard module	module
Class Module	coclass
Enum/End Enum	enum

类型系统的 ODL 版本显示了一些 VB 的行为，这些行为在 VB 产生的类型库内是常见的。

- 一个类有三个类型库入口。EntriesClass coclass 是类自身，它包含 the_EntriesClass 接口和 the_EntriesClass 源接口。如果类有别的实现语句，读者也会在看到每一个实现语句的事件接口之前看到别的接口。
- VB 要求接收事件的对象提供一个 IDispatch 绑定的对象来接收事件。在 coclass 接口的列表内，事件接口标为源(sourse)属性，这表明接口是由特定类使用的，而不是提供的。Dispinterface 规定 VTable 是通过 IDispatch::Invoke 方法支持的，因此调用对象必须通过 Idispatch 才能进行。
- VB 产生有双重接口的对象。这个双重接口是一个“假扮”为 DispInterface 的 VTable 绑定的接口。不支持 VTable 绑定的工具将该接口看作 Dispinterface 并据此进行处理，而更新的工具（VB5 或以后的版本）可以和 VTable 相互作用。VB4、Office 97 发行之前的 VBA 版本和所有的脚本语言（VBScript、JavaScript 等等）都只是使用 IDispatch 绑定。双重属性也暗含着 OleAutomation 属性，这意味着所有的排队可以使用类型库中的类型描述来实现。产生的所有公共类都是双重接口，因此它们都支持 IDispatch，并且所有的公共类都百分之百的依赖于正确注册的类型库来跨线程、进程或机器对象进行排队。
- 所有的 VB 对象都是不可扩展的，这就意味着如果 VB 编译器在类型库中找不到一个方法或属性，就会在编译时产生错误信息。由于 IDispatch::GetIDsOfNames 和 IDispatch::Invoke 在理论上可以添加类型库中不存在的属性和方法，所以如果 VB 不能 VTable 绑定一个特定的调用，就必须将 VTable 绑定变为后期绑定。在实际中倒不常这样做，它也不被 VB 对象所支持（除非读者可以熟练的使用 VBoost 来提供一个用户定制的 IDispatch 实现）。这种不可扩展属性常常加到对象上，以迫使在根据类型信息无法辨识属性、方法或参数时，出现一个编译时错误。
- 非事件接口上的所有 VB 的方法和属性返回 HRESULT，这是用来返回错误代码的 COM 机制。OleAutomation 实际上允许返回 void 或 HRESULT，但是 VB 不允许读者对不返回错误信息的非事件对象进行编码。在第八章的“To Err or Not to Err”一节中，对不返回 HRESULT 的方法做了广泛的讨论。
- VB 按值参数翻译成一个简单的类型，而不是一个指针类型（long 而非 long*）。它经常使用[in]参数属性。[in]参数告诉排队引擎只应向接收方法调用的 Stun 对象复制数据，而不应向调用者复制数据。ByRef 参数经常翻译为带有一个指针类型的[in,out]参数，因此在 ODL 中，ByRef long 变为[in,out] long*。注意到对象、数组、字符串都内在的为指针类型，因此 ByVal MyObject 为[in] MyObject*，ByRef MyObject 为[in,out] MyObject**。VB 传递对象类型时只是传递它的指针，它实际上从不传递数据。尽管 OleAutomation 支持按值(ByVal)传送数组或结构，但 VB 不能传递这两种类型。

- 除了对函数的返回值（它实际上是一个参数，因为真值为 HRESULT）之外，VB 无法指定一个[out] only 参数。这很不幸：数据在不需要排队时还可能会被排队。在本章后面，读者将学到更多的关于[out]参数的知识。
- VB 向它所定义的所有元素分配 GUID(由 ODL 中的 uuid 属性来规定)。类型库编译器需要所有接口、Dispinterface 和 coclasses 的 GUID，而排队需要所有结构(structure)的 GUID。但是，无论是编译器还是排队都需要枚举类型的 GUID。毕竟，枚举类型在编译和排队时只是被看作一个长整形(long)数据。在对二进制兼容性的讨论中，读者将会看到枚举版本问题造成的负面影响。

读者会看到元素和属性非常类似于上面讲的在所有类型库中 VB 创建 ActiveX 组件时产生的东西。这种标准化方法带来的好处是在创建公共类时不必担心类型库产生或者 ODL 定义。不好的一面是读者对 VB 所产生的东西不能加以控制。例如，没有方法直接控制参数上的排队标志、结构的组合程度（packing level）、类型库的版本或接口和类的 GUID。

读者需要采取一种“三叉”的方法以从底层控制工程中使用和产生的类型库。如果要定义 VB 不能描述的类型，就需要创建自己的类型库。如果有显式的要求（例如 VB 需要知道现存的 coclass 或接口标示符以匹配二进制实现和类型库描述），需要在二进制兼容文件中修改类型库和别的资源，这样 VB 在重编译工程时能再现你的设置。最后，如果需要改变只是对象的使用者才关心的设置（例如参数排队信息），应在 VB 完成写入后修改可执行文件中包含的类型库。

15.2 VB 友好的用户定制类型库

尽管 VB 能产生一些高度标准化的类型库，但它能使用更多函数和接口。一个常见的错误认识是 VB 仅能使用双重接口。这句话很简单，但并非事实。VB 能调用几乎所有的接口，通过重定义（或称 VB 化（VB-izing））VB 使用的接口可以将它不能调用的接口数减为零。注意到调用一个接口应归入使用类中；使用实现来实现一个接口表明 VB 既能使用一个接口又能为它产生代码，这意味着实现的接口的要求要比在接口上调用成员函数严格得多。

VB 只是不能处理一些有限的类库元素。如果读者愿意写一些代码或者定义一个 VB 化的代码，就能突破这些限制。

15.2.1 VB 不能使用的类型库元素

1. 参数列表中的按值(ByVal)传送数组

VB 只能处理 SAFEARRAY(elementtype)*数组（对应着 C++中的 SAFEARRAY**）。为

了调用这些类型，应将其重新定义为 ByVal Long 并结合 VBoost.Deref 使用 VarPtrArray 来传递这种类型。使用实现，可以通过将参数改变为 ByVal Long，然后使用第二章介绍的 SafeArray 去引用（dereference）技术，就可以突破这种限制。

2. 参数列表中的 ByVal 结构

参数列表中的 ByVal 结构是很难处理的，但也很少遇到它。如果要实现一个接收 ByVal 参数的结构，应提供一系列长整型参数来正确的填充堆栈。读者可以将所有的参数都设为 ByVal long，并且以每次四个字节的形式重组对象，或者可以将对应着结构中的第一个元素的参数设为 ByRef Long，并且将第一个参数的 VarPtr 引用为 SAFEARRAY 来读取这一结构。读者可以使用类似的做法来调用一个接收 ByVal 结构的函数。

3. 结构中的指针字段

VB 不能处理含有指针类型字段（long*，long**，IDispatch**，MyStruct*等等）的结构。为了调用这些类型，应该使用一个长整形字段并将其使用 VBoost.Deref 或者 SafeArray 技术来引用。在结构中，含有固有指针类型的字段（对象、字符串、可变长度的数组）也是指针类型。

4. 无符号类型

VB 不能使用含有无符号长整形和无符号短整形的结构。VB 惟一使用的无符号类型是 Byte，它对应着一个无符号字符。在这种情况下，在 VB 中有符号、单字节类型（char）都不能访问。幸运的是，解决这一问题是很容易的，但它通常使用 VB 化的类型库。VB 可以使用无符号类型的别名（alias），而不是类型自身。ODL 中的一行代码“typedef [public] unsigned long ULONG;”定义了一个叫做 ULONG 的别名，读者可以在自己的类型库中使用这个别名，这仅仅需要用别名替换无符号长整型或无符号短整型就可以了。在所有的数学运算中，VB 将无符号长整型看作长整型，因此在需要无符号运算时，应使用 VBoost 的无符号算术函数。读者可以忽略别名定义，仅仅使用长整型来替换整型。但是，当要调用函数时，别名提供了有用的提示功能。VB 不能调用字符型或无符号整型的别名。

5. 结构中固定长度字符串

见下面的“字符串类型”一节

6. 联合类型（Union）

联合类型是 VB 不能使用的结构中的一个元素。在所发布的代码中，我从没看到这个类型，所以我不认为这是个多大的损失。

读者可能已经发现了负面的东西很少，但好的东西还在后面。我将集中讨论一些非常有用的类型库类型和属性，VB 使用这些类型但不定义它们。

7. [out]参数

Visual Basic 将 ByVal MyType 映射为[in]MyType，将 ByRef MyType 映射为[in,out]MyType*。不加上[in]，VB 无法产生一个[out]参数。这些标志主要是用来排队接口的；任何 ByRef 类型都是由调用者的进程复制到被调用者的进程，即使这一参数只是用来获取数据的。获得[out]参数的最容易的方式是使用函数的返回值，但这只适用于获取单个数值的情况。

即使没有排队层，[out]参数的一个副效应也是很有用的。[out]规定表示了没有传来新的数据，并且被调用者不应该读取或释放参数中的数据。考虑到[out]参数经常是 ByRef 传送的事实，所要求的新来数据的缺乏意味着在调用函数之前，VB 必须释放传递给[out]参数的变量中的任意数据。VB 使用一个带有[out,retval]参数的临时变量，因此它在清除分配给它的变量之前可以证实函数已经执行成功，但它不使用一个带有[out]参数的临时变量。在函数调用之前，这一变量就被清空。这种用法的一个例子是 VBoost.Move 变体，它将[out]放于 varDst 参数之上，以迫使 VB 在 VBoost 看到新来的参数之前清空它。

如果所有的调用者传递空变量到函数的 out-only 参数，就不会从 auto-free 行为中获取多少东西：排队一个空数组、字符串、变体或对象的费用是微乎其微的。但是，如果调用者传递一个已经包含数据的数组，排队层会将所有的数据复制到单元中，因此可以完全地忽略它。除此之外，如果调用者传递了一个固定长度的数组，并且读者试图 ReDim 这一数组，就会发生一个恼人的错误。就像 VB 不允许将固定长度的数组赋给函数的返回值一样，它不允许读者传递固定长度的数组给[out]参数。被调用者仅仅需要设定一个[out]标志，就可以保证新传来的数据是空的，这样排队层就不会复制不必要的数据浪费资源。

既然已经看到了[out]参数的一些好处，读者可能希望在自己的工程内使用它。VB 不允许读者直接地或在 Implement 接口中使用[out]参数，但是有好几种方法可以欺骗一下 VB。使用 Implement 接口，读者可以生成一个没有[out]参数的接口的 VB 化版本（见“实现接口的 VB 化”）。对于读者的主接口上的函数，需要做后期构建的修改以改变参数的属性（见本章的“后期构建类型库的修改”）。在每一种情况下，读者都需要使自己代码准备好接收[out]参数而不是[in,out]参数。

第一个问题出现是因为可以定义[out]参数来接收零指针值，而[in,out]参数则不能。零指针是很有用的，因为它告诉被调用者对应于参数的数据没有产生和返回。但是，如果读者仅仅通过将[out]参数变为[in,out]来 VB 化一个接口，零指针值会使 VB 崩溃，因为 VB 假设新来的[in,out]参数包含一个有效的、非零的指针。为了使用零指针来接收[out],需要将参数重新定义为[in] long。这允许读者能检验新传来的指针值。读者还可以定义希望得到的类型的局部变量，然后在函数返回之前将其复制到输出参数。可以不使用 long，而是使用一个别名来表示 Intellisense Quick Tip 窗口内参数的真实类型。

```
'The pData parameter is an [out] SAFEARRAY(long) *.
Private Function IVBizedInterface_UseOutParam(_
    ByVal pData As LongArrayPtr) As Long
Dim Data() As Long
    If pData Then
'See next paragraph.
        VBoost.AssignZero ByVal pData
        'Fill Data
        VBoost.AssignSwap ByVal pData, _
            ByVal VarPtrArray(Data)
    End If
End Function
```

尽管下一个问题在理论上是一个问题，但我在实际中还从来没有碰到过。与[in,out]参数不同，[out]参数不保证它接收到一个正确初始化的值。对于[in,out]，指针和它包含的数据都是有效的。[out]参数的指针都保证是有效的，但是数据没有定义。如果没有定义一个数据，使用 VB 将其赋给一个参数，就可能会释放掉没有定义的数据并使整个程序陷于崩溃。例如，如果新传来的字符串参数中包含垃圾数据，将新值赋给那个参数会释放掉原来的数据。考虑到这种情况，我经常在函数的头部显式的将任何[out]参数置零。VB 也经常在调用之前对这些值置零，类型库驱动的排队代码也会做同样的事情。调用代码做这些是出于习惯，它不能保证能防范可能出现的错误，

15.2.2 API 调用的 HRESULT 返回

当读者使用 VB 的 Declare 语句时，不能控制 VB 为函数调用产生的错误处理的类型。在使用 Declare 函数时，VB 会调用 GetLastError API，并且在调用 Declare 函数之后将值存储在 Err.LastDllError 中。这种方法有几个不利之处。首先，存在额外调用的开销，它大约相当于调用一个已声明的函数的百分之十五到二十。其次，许多函数返回 HRESULT 值来直接指定错误条件，而不是使用 SetLastError 机制来返回错误信息。再次，返回 HRESULT 的函数应该被声明为有一个长整型的返回值，因此在函数调用之后应该显式的对其进行检查。

对于 Declare 语句，不幸的是 HRESULT 是 VB 内置的错误处理机制的基础。可以在类型库中使用一个模块单元来代替声明语句，通过这种方法来使用这种错误处理机制。类型库可以使读者声明一个 HRESULT 返回类型并且显式的确定在函数调用之后 VB 是否调用 GetLastError。让我们将 CoCreateGuid 看作使用 HRESULT 的一个例子。VB 不能使用系统

定义的 GUID 类型，因为它使用了无符号类型，在这里可以使用 VBoostTypes 类型库定义的与 VB 兼容的 VBGUID 类型。

```
'Code for calling VB-declared CoCreatGuid.
Public Declare Function CoCreateGuid _
   Lib "ole32.dll" (NewGuid As VBGUID) As Long

'Calling code
Public Function NewGuid() As VBGUID
Dim hr As Long
   hr = CoCreateGuid(NewGuid)
   If hr Then Err.Raise hr
End Function

'Equivalent code using a Module section in a typelib.
[Dllname("ole32.Dll")]
module Ole32
{
   [entry("CoCreateGuid")]
   HRESULT CoCreateGuid([out,retval] VBGUID* retVal);
}

'Calling code using typelib definition.
Public Function NewGuid() As VBGUID
   NewGuid = CocreateGuid
End Function
```

这段调用代码表明当使用类型库声明时，CoCreateGuid 是比较容易调用的。如果 API 调用返回错误，VB 的错误处理机制就自动地捕获这一错误。返回的 HRESULT 也能使读者将这个参数指定为一个 retval，因此读者将看到一个返回 VBGUID 的函数，而不是一个带有一个返回长整型的参数的函数。为了自动地填充 Err.LastDllError 字段，要在定义类型库函数时，将 *usesgetlasterror* 属性添加到入口属性上去。

Declare 语句和类型库模块入口之间的另一个差别是当可执行文件载入时对所有的类型库函数进行解析，而 Declare 函数在第一次被调用时才进行绑定。在程序开始时迫使程序绑定大量的函数有两个不利之处：这要增加程序起始的时间，并且如果没有找到 DLL 或入口点的话，甚至不能启动整个程序。在类型库中声明 API 函数时，最好咨询一下 MSDN 以确

定这些函数存在于将来程序运行的目标操作平台上。

当使用类型库声明时，调用的 DLL 在程序的整个生命期内都阻塞在内存之中。在使用 Declare 语句时，第一次成功的调用 DLL 就将其置于阻塞状态。如果因为内存的原因或者动态升级的原因需要显式的卸载掉 DLL，就需要执行读者自己的 LoadLibrary 和 FreeLibrary 的调用。使用第十一章中介绍的 FunctionDelegator 对象就可以完成这些。由于函数代理使用类型库定义的函数，读者会得到如同普通函数调用时那样的类型和 HRESULT 的好处。不幸的是，在 FuctionDelegator 完成它的调用之后，没有方法迫使 VB 作一个显式的 GetLastError 调用。

15.2.3 字符串类型

Visual Basic 的字符串类型一般是 BSTR，但是在类型库内有三种字符串类型。BSTR 就表示 BSTR，LSTR 表示以 NULL 结尾的 ANSI 字符串，而 LPWSTR 表示以 NULL 结尾的 UNICODE 字符串。通过指定字符串的类型，就可以使字符串的类型的定义为显式定义，这样就避免了使用带有 ByVal Long 的 StrPtr 来作一个 UNICODE API 调用。

为了指定函数需要一个以 NULL 结尾的 ANSI 字符串，要对 ByVal String 使用 [in]LPSTR，并且对 ByRef String 使用[in,out]LPSTR*。VB 负责所有剩下的 ANSI/UNICODE 转换。读者还可以使用 LPSWTR 来表示以 NULL 结尾的 UNICODE 字符串，这个字符串不一定是长度预先确定的 BSTR。在对象浏览器和提示窗口中，这三种字符串类型都显示为 As String。对[out] LP[W] STR*的处理与对 [out]BSTR*的处理稍微有所不同：可以将传递给这些参数的 BSTR 看作一个必要的输出缓冲器，它在函数调用之前不会被释放掉。

如果在类型库中声明的结构需要一个字符串类型，读者还要做一些别的工作。BSTR 是 VB 在记录（Record）中惟一能处理的字符串类型。LPWSTR 和 LPSTR 不编译，没有办法在类型库中定义一个固定长度的字符串。读者可以给出这三种类型的等价类型，但不能无缝的集成这三种类型。让我们首先从 LPWSTR 和 LPSTR 开始。

LPWSTR 和 LPSTR 都是指向以 NULL 结尾的字符串的指针。BSTR 也是指向以 NULL 结尾的字符串的指针，因此可以用 BSTR 来代替 LPWSTR 或 LPSTR，并将这个字段当作一个普通的字符串（LPWSTR）来处理或者赋值 StrConv(,vbFromUnicode)给 LPSTR。这种修改对于作为输入参数的结构工作良好，但是如果被调用的函数负责修改输入结构或者负责在开始时填充这个结构，就会带来灾难性的后果。在这种情况下，只需将类型变为长整型而不是 LPSWTR 或者 LPSTR 并且使用 SysAllocString*函数将字符串指针变为 VB 可读的 BSTR 即可。

当使用固定长度的字符串字段时，对于 UNICODE 字符串要使用短整型数组（在 VB 内为 Integer），而对于 ANSI 字符串要使用无符号字符（在 VB 内是 Byte）。然后需要使用 CopyMemory 将数据复制进结构或从结构中复制出来。这种技术并不太好，它是对内存的错误使用。但是它是可行的。

```
'VB Type
Public Type FLStringField
   TheString As String * 38
End Type

'Typelib equivalent, UNICODE
record TLibFLStringField
{
   short[38] TheString;
}TLibFLStringField;

'Calling code, UNICODE
Dim FLSF As TLibFLStringField
Dim strData As String
Dim cBytes As Long
   'Write to the field.
   CBytes = LenB(strData)
   If cBytes > 38 Then cBytes = 38
   CopyMemory FLSF.TheString(0),ByVal strData, cBytes
   'Read from the field.
   StrData = SysAllocString(VarPtr(FLSF.TheString(0)))
```

```
'Typelib equivalent, ANSI
record TLibFLStringField
{
   unsigned char[38] TheString;
}TLibFLStringField;

'Calling code, ANSI
Dim FLSF As TLibFLStringField
Dim strData As String
Dim strDataA As String
Dim pStringA As Long
Dim cBytes As Long
   'Write to the field.
```

```
    StrDataA = StrConv(strData, vbFromUnicode)
    cBytes = LenB(strData)
    If cBytes>38 Then cBytes = 38
    CopyMemory FLSF.TheString(0), ByVal strDataA, cBytes
    'Read from the field.
    pStringA = VarPtr(FLSF.TheString(0))
          strData = StrConvC_
       SysAllocStringByteLen(pStringA, lstrlen(pStringA), _
          vbUnicode)
```

15.2.4 别名的定义

Visual Basic 不能定义别名类型，但是只要这一别名不是枚举类型的别名，VB 就能很好地使用这一别名。之所以使用别名有几种原因。别名提供了使用类型的规则。例如，尽管变体 ArrayPtr 只是一个长整型，但它能告诉调用者传递变体数组的第一个元素的 VarPtr。除此之外，类型定义(typedef)允许 VB 声明一个未引用的类型库中的类型。在第四章“VTable 绑定用户定制控件接口”一节中讨论的 OCXDirect 插件使用别名，这样读者不需任何对 OCX 自身的工程引用，就可使用任意数量的 OCX 中的类型。

```
//ODL syntax, some aliases in a typelib.
[version(1.0),
   uuid(FEB67C60-CDC2-11d3-BC2E-D41203C10000)]
library AliasLib
{
   importlib("stdole2.Olb");
   importlib(mscomctl.Ocx");

   //Provide a more- instructive type than long.
   //The [public] attribute is required to actually
   //get the type in the typelib.
   //The uuid attribute is not required for an alias.
   typedef [public] long VARIANTArrayPtr;
   //Enable use of Node type without referencing
   //mscomctl.Ocx.
   typedef [public] MSComtlLib.Node Node;
}
```

VB编译器通过将别名充分解析成它们所代表的类型来处理它们，因此它将变体ArrayPtr看作 VB 的长整型，将节点类型（Node）看作在 mscomctrl.Ocx 中声明的节点类型。但是，在原子类型的别名和用户自定义类型的别名之间还存在一些差别。首先，在对象浏览器中，原子类型的别名不是一个类型。其次，如果在公有的方法和属性中使用别名，原子类型的别名会写入到 VB 产生的类型库中，而在将所有其他类型的别名写入类型库之前，会首先对它们进行解析。

当在重新发布类型库时，这种行为上的差异是很重要的。如果读者希望自己的类型库适用于所有的机器，必须要重新发布含有原子类型的类型库。也可以对类型库进行后期绑定，以使别名类型成为可执行的类型库（将在本章后面进行介绍）的一部分。类型库的使用者会在对象浏览器和智能提示窗口中看到一些描述性的类型。用户定制的类型不能写入到公共类型库中去，这一事实意味着可以在工程中使用这些别名而无需将实现的细节广播到所有区域。当对用户定制控件的固有 VTable 接口（见第四章）进行编程时这显得尤其重要。

在 OLE Automation 和 VB 对枚举类型的处理中存在着一个错误。因为这一错误能破坏后向兼容性，所以很难对它进行修改。在类型的名字空间内（在 VB 内位于 As 之后），枚举的别名可以正确的绑定，但在普通的名字空间内则不能。如果 MyEnumAlias 是 MyEnum 的别名，MyEnum.FirstValue 可以正确的编译，但 MyEnumAlias.FirestValue 则不能。读者可以使用别名的值，但不能使用别名的名字对它们进行限定。在使用 MIDL 来构建类型库时，要特别注意这种限制：MIDL 喜欢产生枚举的别名。在本章后面将告诉读者如何克服这种缺省的行为。

15.2.5 常量的定义

在类型库中对枚举的定义十分类似于在 VB 中对枚举结构的定义。但是，枚举不是读者能放置于类型库中的惟一常量类型。实际上，可以定义字符串常量、日期常量和其他的数值类型常量。MyTypLib 不允许读者定义 LPWSTR 或变体类型的常量。MIDL 允许定义这些常量，但是 VB 拒绝使用 LPWSTR，当它访问一个变体常量时会陷于崩溃。在模块中所有的类型库常量都要进行声明，它也适用于 API 模式的函数声明。

```
//mktyplib requires Dllname to compile a module.
[DLLname( "bogus")]
module MyConstants
{
    const double PI = 3.14159265358979;
    const DATE NothingHappened = "2000-01-01";
    const LPSTR Description = "About this library";
}
```

在字符串定义中，MkType 和 MIDL 都能辨认出 escape 序列。如果读者精通于 C 或 C++ 编程的话，可能对这些比较熟悉。但是如果只使用过 VB 字符串，读者就会对这一切感到陌生。在字符串定义中，escape 序列是一个内嵌的字符串序列，它表示一个特定的字符。ESCAPE 序列通常以反斜杠“\”开头，后面是其他的数据。例如，“\t”插入一个制表符，“\r”插入一个回车符，“\n”表示另起一行。除了这些标准的 escape 名字以外，“\”也指定一个字符数。这些数以十六进制（\x##）或八进制（\0###）的格式键入。（注意 mktyplib 仅支持八进制 escape 序列。）

```
//Three representations of carriage return/line feed.
Const LPSTR CRLF = "\r\n";
Const LPSTR CRLFOctal = "\015\012";
Const LPSTR CRLFHex = "\xd\xa"; //MIDL only

//Other escape sequences.
Const LPSTR EmbeddedBackslash = "DLLs\\MyDll.Dll";
Const LPSTR DoubleQuote = "\"";
Const LPSTR TwoTabs = "\t\t";
```

15.2.6 排列（aligned）的结构

所有 VB 定义的结构都是以四个字节为单位排列；这些结构中经常包含没有使用的空间，这些空间常称为 padding，这使结构元素有一个相当的大小。例如，包含整型和长整型的结构有八个字节长，而不是六个字节长，因为在整型数据后面添了两个零，这使长整型数据的边界在第四个字节上。这种排列策略提供了十分有效的代码：CPU 处理没有排列的数据要比处理排列好的数据多做许多工作。但是如果使用有不同排列方式的外部数据，可能会遇到麻烦。例如，通讯端口得来的数据通常是以字节形式排列的，这使以四字节形式读取数据变得很困难。

VB 编译器可以使用任意排列方式的结构，即使它不能定义这些结构。类型库也允许指定结构的排列，因此避开字节排列的问题不谈是很困难的。只要能保证重新发布定义的类型库并且在应用程序中注册类型库（见本章的“注册的效应和负效应”），就能在代码中、公有函数中或 Implememts 中使用任意在类型库中定义的结构。如果这些类型只用于编译，不应指定结构上的 uuid 属性。

使用 line/align# 开关命令，可以指定类型库的缺省结构排列方式。例如，下面的命令使用 MkTypLib 来产生以字节形式排列的结构。

```
mktyplib /align 1 StructLib.Odl
```

MIDL 除了支持缺省的排列以外，还支持 pragma pack，以这种方式读者可以以一种非缺省的方式排列一种结构。

```
#pragma pack(2)
typedef struct PackedStruct
{
    short Field1;
    long Field2;
};
#pragma pack()
```

15.2.7 将 Implements 接口 VB 化

在调用接口上的方法和实际提供接口的一种实现之间存在着很大的差异。VB 能调用几乎任何东西，但是它在实现时的功能就不如在解释时的功能那样强大。如果将本书前面讨论的轻量对象和聚合技术结合起来，有可能构建一个任意接口的实现并将它聚合进任意类的控制 IUnknown 中。但是，读者修改接口的方法也更加简单，VB 也能直接对这些方法进行处理。这就叫做对接口进行 VB 化处理，它需要一个专门为 VB 编译器设计的类型库。

有三个问题是靠将接口 VB 化所不能解决的，它们都涉及 HRESULT 返回值。读者不可能让 VB 实现一个带有不返回 HRESULT 的函数的接口，读者也不能实现一个返回成功的 HRESULT 的接口。成功的 HRESULT 返回代码并不意味着失败，但它提供了关于成功的进行函数调用的一些信息。例如，如果 IoleInPlaceActiveObject 接口的 TranslateAcclerator 函数确实对信息作了一些处理，它就返回 0（S_OK），反之，它返回 1（S_FALSE）。在 VB 中无法返回这个值。最后，读者也无法返回被 VB 映射为标准 VB 错误号的任意 HRESULT 值。例如，读者无法返回 E_NOINTERFACE(&H800A000D)，因为它映射为 VB 的类型不匹配错误信息（&H800A000D）。读者需要使用 VBoost 或者其他的 VTable 构造或修改机制来处理这些情况。

通过将读者自己实现的接口 VB 化可以解决一些不兼容问题。最常见的问题出现于使用 VB 没有定义的类型的时候，例如使用无符号类型和 VT_UNKNOWN 类型的时候。不兼容性问题来源于不直接继承自 IUnknown 或 IDispatch 接口的[out]参数和接口。按下面几步读者可以将 stubborn 接口进行 VB 化。

（1）将接口（包括 IID）复制进一个新的 ODL。从 OleView.Exe 工具或 MSDN 中，可以获得对接口的正式描述。

（2）确定双重和 oleautomation 属性没有设置。同时要增加 ODL 属性。

（3）通过将基类的方法（而不是 IUnknown 和 IDispatch 的方法）加到函数列表的顶部

来 flatten 接口。

（4）检查无符号类型，增加类型定义或者将其用相近的有符号类型来代替。

（5）检查[out]参数。如果仅有一个[out]参数并且这是最后一个参数，只要将其变为[out,retval]即可。如果不能转变，应将其转变为[in,out] type*p1 或者[in] long p1。参考前面对[out]参数的讨论来确定如何在实现中处理这些参数。

（6）将 IUnknown 变为 stdole.IUnkKnown。

（7）如果不能正常工作，就一直对接口进行操作直到 VB 同意编译 Implememts 语句和它的所有方法。如果不知道问题出在何处，注释掉 ODL 中的所有方法并将其依次加回到 ODL 中。这样做可能稍微有些麻烦（需要关掉引用的工程来重新构建类型库），但是花上一段时间，读者就可以修正错误。

一个简单的 IPersistHistroy 接口实现展示了对一个对接口进行 VB 化所需的过程。Internet Exploer(版本 4.0 或更高)使用 IPersistHistory 来缓存用户定制控件的信息，因此当用户使用前进（Forward）或后退（Backward）按钮时，控件可返回到它的当前状态。IPersistHistory 通过对 IE 提供的 IStream 的实现进行读写来维持它的状态。尽管可以使用 VB 化的 lStream 接口来调用流，但没有必要这样做，因为 VB 已经支持 IPersistStreamInit 接口。应该将任何流操作交给接口的 VB 实现。实际上，我将使用两个接口来传递[in] long 参数而不是[in] lstream*参数。这就允许通过 IPersistHistory 实现来传递流指针到 VB 的 IPersistStreamInit，而无需考虑它的真实类型。

IPersistHistory 的 MSDN 描述如下：

```
[
    uuid(91A565C1-E38F-11d0-94BF-00A0C9055CBF),
    object, pointer_default(unique)
]
interface IPersistHistory :IPersist
{
    typedef [ unique ] IPersistHistory *LPPERSISTHISTORY;

    HRESULT LoadHistory(
            [in] IStream *pStream,
            [in] IBindCtx *pbc);
    HRESULT SaveHistory(
            [in] IStream *pStream);
    HERSULT Set PositionCookie(
            [in] DWORD dwPositioncookie);
    HERSULT GetPositionCookie(
            [out]DWORD *pdwPositioncookie);
}
```

这里我要做三个修改。首先，通过包括进 GetClassID 函数，可以将 IPersist 接口 flatten 为 VB 化的接口。GetClassID 带有一个类型为 CLSID 的[out]参数：它可变为[in] long 参数，所以可将其提交给一个简化的 IPersistStreamInit 定义。IStream*和 IBindCtx*参数变为[in] long*，DWORD 变为长整型并且[out] DWORD*变为[out,retval] long*。MIDL 中特有的属性（object、pointer_default）和类型定义都不再需要。这样得到的一个类型库也包括 IPersistStreamInit 接口的一个简化的重新定义，类型库如下所示：

```
[
    uuid(6FBB99C0-D342-11d3-BC39-D41203C10000),
    helpString ("IPersistHistory: Simplified Types" ),
    lcid(0x0),
    version(1.0)
]

library PersistHistory
{
    importlib("stdole2.tlb");
    [
        uuid(7FD52380-4E07-101B-AE2D-08002B2EC713)'
        odl
    ]
    interface IPersistStreamInit :IUnknown
    {
        HRESULT GetClassID([in] Long pClassID);
        //IsDirty returns S_OK or S_FALSE. VB Can't
        //distinguish between these values ,so we
        //change the return type to long.
        Long IsDirty();
        HRESULT Load ([in] long pStream);
        HRESULT Save ([in] long pStream),
                      [in] long fClearDirty);
        HRESULT GetSizeMax([in] long pcbSize);
    }
    [
        uuid(91A565C1-E38F-11d0-94BF-00A0C9055CBF)
        odl
    ]
```

```
    interface IPersistHistory : IUnknown
    {
        HRESULT GetClassID([in] long pClassID);
        HRESULT LoadHistory([in] long pStream,
                            [in] long pbc);
        HRESULT SaveHistory ([in] long pStream);
        HRESULT SetPositioncookie([in] long
                                  dwPositioncookie);
        HRESULT GetPositioncookie ([out, retval] long*
                                   pdwPositionCookie);
    }
}
```

定义完成之后，控件的实际实现就会变得惊人的简单。读者除了能帮助编译器之外，还可以通过干预将接口 VB 化的过程来简化实现。下面的控件例子包括一个 TextBox（叫做 txtData），它在浏览器的整个生命期内始终保持。（这一工程包含在本书所附带的光盘的 Samples\PersistHistory 目录下。）

```
Implements IPersistHistory

'The three functions we care about simply defer to the
'IPersistStreamInit implementation
Private Sub IPersistHistory_GetClassID( _
    ByVal pClassID As Long)
Dim pPSI As IPersistStreamInit
    Set pPSI = Me
    pPSI.GetClassID pClassID
End Sub
Private Sub IPersistHistory_LoadHistory(_
    ByVal pStream As Long, ByVal pbc As Long)
Dim pPSI As IPersistStreamInit
    Set pPSI =Me
    pPSI.Load pStream
End Sub
Private Sub IPersistHistory_SaveHistory( _
    ByVal pStream As Long)
Dim pPSI As IPersistStreamInit
```

```
    Set pPST =Me
    pPSI.Save pStream, 0
End Sub

'These both return E_NOTIMPL, as indicated in MSDN.
Private Sub IPersistHistory_SetPositionCookie (_
    ByVal dwPositionCookie As Long)
        Err.Raise E_NOTIMPL
End Sub
Private Function IPersistHistory_GetPositionCookie() As Long
    Err.Raise E_NOTIMPL
End Function

'Property bag procedures are called by the control's native
'IPersistStreamInit implementation.
Private Sub UserControl_InitProperties()
    txtData.Text = "<No Data>"
End Sub
Private Sub UserControl_ReadProperties
    (PropBag As PropertyBag)
        txtData.Text = PropBag.ReadProperty("Data")
End Sub
Private Sub userControl_WriteProperties( _
    PropBag As PropertyBag)
        PropBag.WriteProperty "Data", txtData.Text
End Sub

Private Sub UserControl_Resize()
    txtData.Move 0, 0, ScaleWidth, ScaleHeight
End Sub
```

15.2.8 注册的效应和副效应

类型库和注册紧密相关。当注册类型库时，读者添加了一个键到注册表的HKEY_CLASSES_ROOT\TypeLib单元。但是这个键不是在注册时惟一增加的东西：类型库

在注册时也会将键加到 HKCR\Interface 单元，这一单元在跨线层、进程和机器边界来调用 COM 方法时起着重要的作用。当注册一个 VB 创建的 ActiveX 服务器时，VB 自动的注册其中包含的类型库资源（包括对接口单元进行的改变）。

```
HKEY_CLASSES_ROOT
    Interface
        {39719D96-9A7A-11D3-BBEC-D41203C10000}
            (Default)= IMyInterface
            ProxyStubClsid
                (Default)= {00020424-0000-0000-C000-
                                000000000046}
            ProxyStubCIsid32
                (Default)= {00020424-0000-0000-C000-
                                000000000046}
        TypeLib
                (Default) ={39719D93-9A7A-11D3-BBEC-
                          D41203C10000}
                Version = 1.0
```

如果跟踪 CLSID 单元中 ProxyStubClsid32 键，会发现下面的代码：

```
HKEY_CLASSES_ROOT
    CLSID
        {00020424-0000-0000-C000-  000000000046}
            (Default) = PSOAInterface
        lnProcServer
            (Default) = ole2disp.Dll
        InProcServer32
            (Default) = oleaut32.Dll
            ThreadingModel = Both
```

同样的，这些键表明 IMyInterface 接口中的方法和属性都使用 OleAut32.Dll 来排队，OleAu32.Dll 在指定的类型库内找寻以确定如何完成排队。如果 COM 找不到 Interface 或 TypeLib 键，读者就不能排队这个接口。这一键是增加到所有标有 oleautomation 或 dual 属性的接口上的，它告诉 OLE Automation 基于类型库来排队。无论类型库是在何时注册的，这些键都写入到注册表中，即使它们已经存在了。这样做通常会得到期望的结果，但是它

要求读者要小心对待 VB 化的接口，以避免覆盖了系统设置。例如，IObjectSafety 的设置如下所示：

```
HKEY_CLASSES_ROOT
    Interface
        {CB5BDC81-93C1-11CF-8F20-00805F2CD064}
            (Default) = IObjectSafety
            NumMethods
                (Default)=5
            ProxyStubClsid32
            (Default) = _
                {B8DA6310-E19B-11D0-933C-00A0C90DCAA9}

HKEY_CLASSES_ROOT
    CLSID
        {B8DA6310-E19B-11D0-933C-00A0C90DCAA9}
            (Default) = PSFactoryBuffer
            InProcServer32
                (Defalt) = "c:\WINNT\System32\ACTXPRXY.Dll"
                ThreadingModel = Both
```

如果创建 IObjectSafety 的 VB 化版本并覆盖注册表的设置，读者机器上的 IObjectSafety 接口就不再正常工作。同时，关于 Interface 键的以前的内容的信息也丢失了，因此重新设置读者的系统不是无关紧要的小事。Interface 键被破坏是一个很难发现的错误：它一直潜伏着，直到几周或几星期后错误才爆发。当在 VB 化接口时，要注意不要使用 oleautomation 或 dual 属性。注意到有一个 proxy 标志可以使对接口键作的修改无效，无论其他标志是如何设置的。但是绝大多数的 MkTypLib 和 MIDL 都不支持这一属性。

读者可能会认为对接口进行 VB 化就会带来排队上的困难。但是，实际上没有出现这一问题。用户定制的接口仅仅影响读者的工程内的代码。一旦读者离开工程的边界，就将控制权交给系统来完成排队工作。系统知道的关于接口的所有信息是它的 IID，接口的 IID 在系统接口的固有和 VB 化版本内都是相同的。系统使用已注册的 ProxyStubClsid32 入口所指定的对象来完成排队，这就意味着一旦到了排队层，VB 化的接口就不再起任何作用。

VB6 中公共结构的引入给 VB 与正确注册的类型库关联性（dependency）带来了问题。公有结构允许在公共类模块中声明一个公共类型并正确的排队这一类型。只有在排队接口时，才需要担心注册表的设置问题，但是读者也要在读者的局部工程中处理正确注册的记录类型。结构的类型库驱动的排队需要类型库，这与排队接口也需要类型库一样，但是恢

复机制并不需要知道接口的设置。对结构进行排队是通过记录类型的类型库描述对接口进行排队的一部分。但是，排队一个简单的类型只是其中的一种考虑。

除了简单的传递一个的记录类型之外，读者也可以在变体或数组中传递一个记录。这都需要加上对记录的描述，这种描述叫做 IRecordInfo。VB 使用 GetRecordInfoFormGuids API 可获取一个 IRecordInfo 加到变体(rariant)或 SafeArray 上。GetRecordInfoFromGuids 使用 LIBID 来表示类型库，使用记录的 GUID 来标识类型库中的类型。尽管 IRecordInfo 只是用来进行排队，但是当创建数组或变体时，无法知道是否需要进行排队。当读者使用一个带有公共结构的数组或变体时，VB 总是设法定位类型库或记录，即使读者没有对它进行排队。

考虑到所有这些因素，如果读者要排队一个记录或者在变体或数组中使用记录，必须正确注册包含记录的类型库。如果需要一个数组，有一个简单的解决方法：在定义这个类型时忽略 uuid 属性。没有了 uuid，VB 编译器就不再试图定位 IRecordInfo，这时就不需要一个注册了的类型库。但是，不能将这一结构放于一个变体中，并且当没有指定 IID 时，读者不能排队一个结构数组。

当读者在定义公共接口时，也应注意类型库的关联物，因为一个给定接口的所有关联物都要进行正确的注册以便进行排队。例如，VBoostTypes 类型库定义了一个不带有 uuid 参数的 VBGUID 记录类型。如果要使用 VBGUID 并正确的排队接口，必须在所有使用公有接口的机器上注册 VBoostType6.Olb。但是，如果要创建 VBGUIDs 数组，就不必注册 VboostTypes，因为这时没有指定 uuid 属性。同样，读者也不能在自己发行的公共接口中使用原子别名类型，因为这会增加类型库相关性。这会给 VBoostTypes6.Olb 和其他的一些仅仅用于开发的类型库带来一些问题，因此要使用以后要介绍的后期构建类型库修改插件来将外部的类型库关联物移植到发行的产品中去。

表 15.3 使用公有记录的规则

所希望的特征	要求
Marshal simple type	Registered typelib，no uuid required
Marshal array or variant	Registered typelib，uuid repuired
Assign to variant	Registered typelib，uuid repuired
Assign to array	Registered typelib，only required if uuid specified
Pass simple type in same thread	Registered typelib，no uuid repuired

15.2.9 MkTypLib、MIDL 与直接 API 调用

由于 Microsoft 倾向于使用类型库编译器，它正式的使用 MIDI.EXE 来代替 MkTypLib.EXE。但是，根据以前的经验，我发现 MkTypLib 在产生严格为 VB 设计的类型

库方面要工作的更好。MIDL 是比 MkTypLib 功能更强大的多用途工具。MIDL 的主要任务是产生用户排队的 DLL，而 MyTypLib 的首要任务是产生类型库。MIDL 使用起来比较复杂，读者不必使用它来为 VB 的编译器创建类型库。在 MIDL 中有许多原因使它难于用来产生类型库。

- MIDL 经常将 stdole.IUnknown 解析为 VT_UNKNOWN 类型而不是带有 IID_IUnknown 的 GUID 的 VT_USERDEFINED 类型。使用 MIDL 无法完成通常的在 VB 化的实现中使用 MIDL 的要求。
- MIDL 不允许读者重新定义一个已经在移入的库中使用的 IID。读者无法创建在 impotlib 中定义的接口的 VB 化版本，这就迫使读者不得不重新定义（不是取别名）接口需要的其他类型。
- MIDL 需要一些额外的代码以防止它为枚举和记录类型产生别名。在 MIDL 中，下面的第一个 ODL 产生两种类型：一个名字与_MIDL_MIDL_itf_filename_0000_0001 类似的枚举类型和这个枚举类型一个叫做 MyEnum 的别名。读者必须做一些变化以使 MIDL 产生一个单一类型。前面已经提及过枚举的别名对编译器是一个坏消息，所以要求使用 MIDL 的第二种语法。

```
//Generates an enum and an alias.
typedef [public] enum
{
   Value 1
}MyEnum;
//Just generates an enum.
typedef [public] enum MyEnum
{
   Value1
}MyEnum;
```

- MIDL 在正确编译时要求额外的路径信息。特别是，它要求 Include 路径包含 Visual Studio 安装的 VC98\Include 子目录。
- MkTypLib 与独立的 VB 产品一起发行，而 MIDL 则不能。
- 对布尔类型（Boolean），MIDL 和 MkTypLib 使用不同的语法。MkTypLib 使用 boolean，但在 MIDL 中，布尔类型映射为单字节类型而不是 OLE Automation 标准中的双字节类型。为了使 ODL 与两种编译器都能正常工作，应该在类库体内使用 Variant_BOOL 而不是 boolean，并且在 ODL 文件的头部包含下面的几行。

```
#ifdef _MKTYPLIB_
```

```
#ifdef VARIANT_BOOL
#undef VARIANT_BOOL
#endif
#define VARIANT_BOOL boolean
#endif //_MKTYPLIB_
```

MkTyLib 和 MIDL 并不是创建类型库惟一的两种选择。实际上，这些编译器只是对调用 ICreateTypeLib2 和 ICreateTypeInfo2 接口的封装。在 OleAuto32.Dll 中的 CreateTypeLib2 API 上可以找到 ICreateTypeLib2 接口，在任意的 ITypeLib 引用上调用 QUeryInterface 也可以完成这一工作。类似的，也可以从任意的 ItypeInfoy 的引用上得到 ICreateTypeInfo2 接口。所有的类型库接口和函数都是在 TLBType.olb 中定义的（VBoost:类型库类型和接口）。除此之外，还有一个叫做 PowerVB 类型库编辑器的类型库编辑器插件，它能使读者在 VB 的 IDE 中定义和修改所引用的类型库。（本书中所带的光盘里已经含有了这个编辑器和其他工具的源代码）

使用 API 来直接使用类型库有几个原因。首先，API 能启动一个对用户友好的图形用户界面，这在 ODL 中是无法做到的。其次，API 能够复制或编辑现存的类型库。再次，使用 API 可以完成几种使用编辑器无法完成的工作。例如，读者可以为一个类型库指定多个帮助文件、使用以非零元素结尾的固定长度的数组、显示的表示 VTable 顺序。API 也支持使用未在类型库中定义的类型（前向引用）。VB 使用一种特殊的顺序产生类型库。例如，枚举和记录类型要在函数使用完后，才进行定义。如果读者使用 ODL 来编辑由 VB 产生的类型库并对它进行编译，结果常会出错。可靠的编辑兼容性文件和其他由 VB 产生的类型库的惟一方法是直接使用 ICreateType*接口。

读者可能会认为类型顺序并不重要。但是如果使用 VBA 中的库并改变类型顺序，读者必须至少要增加库的较小版本号。如果不那样做的话，VBA 工程会在试图对新库进行重新编译时陷于崩溃。我不知道其的关于库中类型顺序的工具，但确实存在一些别的工具。

15.3 二进制兼容性

对于大多系统来说，对 COM 组件进行升级是一个常见的要求。毕竟，使用组件的一个主要原因是它具有不用升级整个系统就可以升级的能力。读者可能会想到要升级 DLL 或 OCX 来修补错误或增添一些新的特征。在每一种情况下，读者都要保证在旧的版本上可以运行的代码可以在新的版本上运行。这就是 VB 的工程兼容性设置要达到的目标。

VB 为兼容性提供了三个不同的选项。第一个选项是无兼容性，它要求 VB 在每次重新编译时重新生成所有可执行文件的外部可见元素。类型库的 LIBID 改变了，CLSID、IID、VTable 顺序、DISPID 和其他任意读者能想象得到的 ID 都会发生改变。第二个兼容层次是

工程兼容，它锁定 LIBID 和 CLSID 的值，但其他的一些都是易变的。这就允许在开发周期内进行持久跨过程引用。当设置无兼容性时，读者必须经常的为所使用的工程编辑工程引用。对于早期开发来说，工程兼容性是一个方便的设置，但不能提供足够的对升级组件的版本的控制。在组件离开你的机器之前，通常应该选择最大程度的兼容性选项，也就是二进制兼容性选项。

读者通常要给可执行文件的编译后版本设置兼容性文件。尽管 VB 允许对目标可执行文件设置兼容性文件，但基于以下两种原因，应该尽量避免这样做。首先，如果在类型之间存在引用的话（例如返回类型为 ClassY 的对象的方法），MAKE EXE 步会出现错误，因为 VB 正在使用这些文件。其次，如果将兼容性文件指向要经常进行升级的目标代码，兼容性文件中会包含一些现在不再需要的构建信息。一个典型的开发周期内包含许多中间的构建，但很少有在开发和质量保证机构之外的机器上做中间构建的情况。读者只是关心现在的版本与已经发行到用户手中的老版本之间的兼容性，而不是与用户没有见到过的版本的兼容性。但 VB 不区分中间版本和发行版本，它只是很高兴地看到程序不仅与正式发行的版本相兼容，而且与一些中间的垃圾程序也保持兼容。

15.3.1 兼容性的好处

读者已经看到 VB 几乎能产生所有的类型库。但是，兼容性又给自动生成的类型库带来了新的麻烦，因为类型库不仅要与旧版本相兼容，而且要与新版本相适应。类型库的输出既基于兼容性文件，又基于当前的工程。对于每一个接口，VB 能保证兼容性文件中定义的接口的所有方法和属性在新工程内也可以找得到。如果接口是二进制兼容的，但 VB 找到了新的方法和属性，它要做五件事情，每做一件事情时，VB 都不通知读者作出的任何改变。

- VB 给出接口的新 IID。
- VB 增加一个别名类型到类型库中。将原先接口的 IID 赋给这一别名，别名重新指向当前的接口。这一别名是一个隐藏类型，它的名字类似于 Class1_v0，它同先前接口的版本有相同的版本号。
- VB 增加接口上的较小版本号。注意到这个版本号并不用于实际上进行排队或编译，它只是一种记录机制。
- VB 增加类型库的较小版本号。这个版本号特别是被 VBA 所使用。如果一个被引用的类型库的较小版本号改变，VBA 自动重新编译引用这一类型库的代码。注意在库中类型的顺序改变时，增加版本号是很关键的。如果读者改变类型号而没有改变版本号，VBA 会在重新编译时崩溃。
- VB 修改 the_IID_CLASS1 资源。VB 为工程中定义的每一个接口来构建资源。IID 资源的二进制格式是很简单的：头四个字节表示列表中的 IID 的个数，后面跟着的是一系列 IID。

如果将这个新文件存储为兼容性文件，那要在重新编译前增加一个或者多个方法时，就要重复整个过程。如果不升级兼容性文件，VB 就将它用作构建新的类型库的基础。实际上，这意味着读者的接口的实际 IID 不断改变，对于使用它的用户来说变成了一个易变的值。例如，考虑一下下面的步骤。

（1）有一个 ActiveX DLL 和一个二进制兼容文件，读者需要在自己的类上增加一个新方法。

（2）添加方法并编译 DLL；

（3）关闭 DLL 工程，并打开另一个调用 DLL 的工程。读者可能需要也可能不需要这个新方法。编译这个工程。

（4）下一步，修改新方法中的错误并重新汇编 DLL；

（5）当读者的已编译 EXE 文件试图获取接口时，会因出现类型不匹配的错误而导致调用失败。

在这里读者丢失的一步是如果兼容性文件找不到新的方法，每次读者重新构建可执行文件时，VB 会产生一个新的 IID。注意到根据读者在编译使所作的编辑的数量和类型，如果读者在同一个 IDE 中编译两次的话，IID 不会改变。必须将包含新方法的 DLL 拷贝为读者的兼容性文件以锁定 IID。这里的困难之处是每次增加一个方法，兼容性文件中垃圾的数量增加，读者的发行的产品中垃圾的数量也会相应的增加。

Visual Basic 需要知道关于旧接口 IID 的所有信息，以使 DLL 成为先前版本的类型库的二进制替代。VB 使用 IID 资源来为所有以前的接口提供排队支持，并且它使用这些资源以使 QueryInterface 函数能正确的响应对先前的接口标识符的请求。为了支持在一个合法的接口上进行管理，VB 使用指向主接口的 Interface\{...}\Forward 键来替代 Interface\{...}\TypeLib 键。forward 键告诉类型库驱动的排队引擎要像对待当前实现的主接口一样对待这一接口。

读者完成的产品中包含了所有的兼容信息。当读者扩充接口时，资源会变得更大，这增加了注册表的花销；随着别名的增加，类型库也会随之增加。公共访问的对象之后的 QueryInterface 实现在对接口请求回答 yes 或者 no 之前，要检查更多的数据。幸运的是，在新版本的 DLL 上编译的客户不能从老版本的 DLL 中间得到它所期望的接口，也不能调用一个老版本的 VTable 不支持的方法。

观察 VB 对一个特定的接口所做的改变可以更深入的考察 VB 显示类型库的方法和在何种条件下，它试图保护读者。下面的例子是一个简单类库的三个版本。我将对第一个版本构建二进制兼容性文件，然后通过增加一个函数来修改这个类并且重命名第一个函数。我将忽略 IID 的改变而只注意其他三个字段。第一个相关的数是 VTable 偏移量；第二个是 DISPID；第三个是函数在类型库中排列的顺序。

```
'The starting class
Public Function F1() As Long
End Function
Public Function F2() As Long
```

```
End Function

'Add a function
Public Function F1() As Long
End Function
```

表 15.4 起始类中的函数布局

函数	VTable 偏移量	DISPID(hex)	函数顺序
F1	28	60030000	1
F2	32	60030001	2

表 15.5 可兼容编辑的函数布局

函数	VTable 偏移量	DISPID(hex)	函数顺序
F1	28	60030000	1
F2	32	60030001	3
F3	36	60030002	2

表 15.6 兼容性被破坏的函数布局

函数	VTable 偏移量	DISPID(hex)	函数顺序
FIEx	28	60030003	1
F2	36	60030005	3
F3	32	60030004	2

```
Public Function F1() As Long
End Function
Public Function F2() As Long
End Function

'Rename the first function, accepting the warning that you
'are breaking binary compatibility.
Public Function F1Ex() As Long
End Function
Public Function F3() As Long
```

```
End Function
Public Function F2() As Long
End Function
```

从这些数据中可以得到一些信息。首先，类型库中的函数顺序与模块中的函数顺序相同。其次，VTable 顺序首先是基于二进制兼容性文件的顺序，然后才是基于 CLS 文件中的函数顺序的。再次，破坏兼容性会重新排列 VTable(这是第一条规定和第二条规定的必然结果)。最后，VB 保证新接口中的 DISPID 不与二进制兼容接口中 DISPID 值相重叠。之所以会不重叠是因为老版本和新版本有同样的 CLSID，都支持 IDispatch 绑定。读者无需查询新产生的 IID 就可以在接口上作 IDispatch 调用。使用新的 IID 产生新的 IDispatch 有效的阻塞了对接口的先前版本的调用。

二进制兼容性遭破坏的部分原因是因为先前接口的所有信息都已经丢失了，这些信息不仅包括破坏后的接口的信息，而且包括工程中别的接口的信息。当兼容性被破坏后，所有接口（不只是读者破坏的那一个）的兼容性别名都破坏了。类型库的主版本也增加了。VB 恢复到工程兼容，只保存了 LIBID 和 CLSID 的值。

15.3.2 开发兼容性要求

VB 提供的兼容性保护是特别强大的，但是读者也要考虑一下在给一个接口增加新方法时，改变接口的 IID 带来的益处何在。在 Set 语句中，使用接口描述符来证实运行的对象支持由它的设计时间类型库描述符所约定的 IID。如果一个新的用户调用了一个不支持最新、最重要的接口的老 DLL，Set 语句就会执行失败。如果仅仅在 VTable 后部增加一个新方法而不改变 IID，Set 语句会执行成功，但随后的方法调用会在 VTable 表的结尾之外调用一个函数（这通常会引起冲突）。无论在哪一种情况下，如果用户代码调用一个不被 ActiveX DLL 的当前版本支持的方法时，程序都不会继续执行。VB 的 IID 修改过程提供了一个看似较轻的致命错误，但它仍然是致命错误。

表 15.7 对调用代码函数的兼容性要求

调用代码	VTable	DISPID	成员名	接口名
Vtable bound	Yes	No	No	-
DISPID bound	No	Yes	No	-
Late bound	No	No	Yes	-
New	-	-	-	No
CreateObject	-	-	-	Yes

在重新构建过程中经常改变 IID 会使组件成为活动的物体。这在开发周期内会带来一些问题。它不仅仅造成我刚才谈的膨胀的问题：它也迫使读者重新编译一个使用修改后的组件的应用程序，即使调用代码不使用修改过的方法。当读者在一个开发小组内工作时或在 C++中调用组件时，这些问题都会叠加起来，它迫使读者在每次目标类型库改变时都要重新产生头文件。在开发过程中，读者需要一定程度的兼容性控制，但是很少有需要 VB 提供的 iron-clad 控制的情况。穿上全副盔甲确实不会有坏处，但是始终穿着厚厚的盔甲也是一件很受罪的事。

如果读者需要 IID 在修改时保持相同，读者就应在这三种兼容性之间做出选择。读者需要的兼容性的类型完全依赖于调用应用程序的当前版本是如何绑定到对象上去的。如果调用代码是 VTable 绑定的，读者应该保持所有现在的函数 VTable 顺序和函数别名。但是函数的实际名字和 DISPID 值可以改变，因为以后也不会使用到这些东西。如果读者的代码是早期绑定的，VTable 和名字可能改变，但 DISPID 值应保持相同。如果调用代码是完全的后期绑定，必须使它们的名字保持相同。如果现存的代码完全是使用关键字 New 而不是 CreateObject，除了改变这些成员之外，读者还能改变类名。

15.3.3 二进制兼容性的编辑

PowerVB 二进制兼容性编辑器（EditCompat.Dll）在本书的 CD 中可以找到档案文本，它提供了对二进制兼容性文件的完全控制。

- 只要文件与 VB 产生的代码相兼容，就可以直接对它进行编辑。编辑工作包括改变成员的名字、删除整个类型、删除辅助接口（由实现语句产生的）、增加成员和其他类似的行为。注意到只有在编辑工程时不能选择“保存兼容性”选项时，才需要这种直接编辑能力。其中一些编辑会改变兼容性文件的 IID 资源。注意到这会使兼容性文件的资源与编译后的代码不一致：因为兼容性的原因，读者仍能使用这些文件，但是不能将它们用作可执行文件。
- 读者可以通过增加新的成员到接口上来取消任何 IID 升级。如果构建完成时没有兼容性错误，可以根据旧兼容性文件生成一个新的兼容性文件（基于新的 EXE 文件中的类型库）和 IIDs。
- 读者可以从读者的兼容性文件中自动去除所有的别名垃圾。这仅仅需要提供包含以前发行旧版本的可执行文件的目录。编辑器会对这所有的版本进行分析以确定需要支持哪一个中间接口。在以前版本中不能找到的有别名接口都被去除。

总之，二进制兼容性插件使读者可以充分的控制组件版本。读者可以做一些编辑，从兼容性文件中增加或去除兼容性支持，然后让 VB 根据当前的设置重新产生可执行文件。它允许读者做一些简单的编辑，例如改变成员的名字、增加一个新的成员或去除一个实现接口，这些操作不会给现在的代码带来问题。尽管这一插件是一个很有用的开发工具，它也有可能把事情搞得一团糟。读者应经常给最后的版本设置二进制兼容性文件，并且在构

建时不要出现兼容性错误。

15.4 后期构建类型库的修改

VB 在组件的类型库中放置了大量的信息，但是在应用程序运行之前，读者通常要发布更多的常驻类型库信息。如果读者的计算机需要这些信息，就必须和 VB 构建的二进制文件一起来分配并管理其他一些文件。当读者注册一个 VB 组件时，VB 运行时间自动的为读者注册它的类型库。读者可以通过修改原先的类型库并用用户定制的类型库来代替原先的类型库，来使 VB 注册更多（或更少）的类型库信息。在几个常见的情况下，读者会发现这种技术特别有用。这里提及的所有编辑操作都可以使用本书附带的光盘中的 PowerVB 类型库修改符来实现。本小节只是简单的看一下使用这一工具的几个原因。

- 如果读者已经使用类型库描述了实现的一个接口并且需要跨过线程、进程或机器的边界来访问这个接口，就必须注册一个包含接口定义的类型库。这通常意味着读者需要传递和注册 TLB 文件和组件。读者可以通过将外部的类型引入到读者的可执行文件的类库资源中并将所有的外部引用重定向给内部拷贝，来消除附加的文件关联性。
- 如果读者将实现作为一个内部（线程内）设计结构来使用，并且读者不希望广播这一信息，可以有两种选择。使用第一种方法，读者只需简单的将接口定义添加在类上，通过不发行引用的类型库来孤立接口定义。除非二进制码是 OCX 并且读者需要使用 VB5，这种解决方法是一种良好的方法。在这种情况下，读者必须发行引用的类型库或者结束不含有事件的控件（这一由 OCA 产生的错误已经在 VB6 中得到了更正）。而在第二种方法中，读者应该修改类型库并且删除接口引用。注意在这里实际上读者想要得到的是在 VB 中对私有实现的支持，它允许一个类支持一个接口但无需将它写入到类型库中。
- 如果使用 VBoost 聚合类型来实现一个接口，这个接口不会出现在 coclass 中。如果要将接口的实现公有化，读者应将它增加到已实现的接口的列表中去。
- 如果读者要 VB 化一个在类型库中定义的接口以有助于 Implements，这个类型库就包含一个对 VB 化的类型库的引用，而不是引用原先的类型库。如果读者不发行和注册 VB 化的类型库，这一引用就不能稳定存在并且也不可以读取接口标识符。一些应用程序通过在 coclass 上寻找已实现的接口来检验一个组件。如果读者希望能识别一个由 VB 产生的组件，需要将接口重定向到真正的类型库上而不是 VB 化的版本上。
- 在 VB 中读者不能定义一些数据类型。例如，读者希望可执行文件中包含着字符串常量或别名。使用后期构建修改，读者可以在外部的库中定义这一类型并在发行之前将它们并入到可执行文件的库中。

- 读者可能发现在类型库中定义结构和枚举类型要比在作为公共入口的类模块中定义更容易一些。它给与读者无需改变兼容性就可修改枚举类型的自由，并且它使读者的公共结构具有排列能力。如果读者在发行之前将这些类型并入到读者的主类型库中，就不必重新发布外部类库。
- 如果要改变一个类的缺省接口，需要编辑类型库。VBoost 对象能使读者将对 IDispatch 对象的 QueryInterface 请求重定向到一个辅助接口上，但是如果读者不接收 QI 或 IID_IDispatch，这样做也没有用处。通过修改类型库并重定向 QI 调用，读者就能使自己的对象的实现和类型库描述保持同步。这样读者就在多个对象上放置了一个标准的缺省接口并且可以使用类似 Open、Close、Input、Stop 等的方法来构建 VB 对象。通常不能使用这些名字，因为 VB 不会让读者使用与关键字相冲突的方法或属性名。
- 读者可以通过改变[in,out]参数设置为[in]或[out]来帮助排队。在一个数组或结构上将[in,out]参数改变为[in]参数可以在函数调用返回时节省整个数组的拷贝。将一个数组参数的属性改变为[out]可以保证当外部调用源调用数组参数时，总有一个空数组可以与之打交道。

本书提供的类型库工具是用来提高而非替代 VB 的类型库自动生成能力。在绝大多数时间内，VB 能完成读者想要的工作。本书中提供的这些工具可以使开发过程变得更简单，并且能产生有专业质量的软件产品。

第十六章

控 制 窗 口

尽管在 VB6 中广泛的使用了 COM 对象，人们通常还是把它看作一个 Windows 开发工具。最初的 VB 工程一般都是一个包含 1 至 2 个窗体的标准 EXE，而不是一个 ActiveX DLL。在 Windows 下编程的时候，我们常常在窗口、控件、以及其他用户界面的创建和交互上花费很多时间。毕竟，如果我们不能将结果显示给用户的话，那么程序也将没有任何意义。本章将主要讨论如何创建窗口以及子类化这样的问题。子类化允许人们处理从操作系统发送到一个窗口对象的消息流。窗口创建允许我们从零开始创建子定义的窗口，而不是使用 VB 已创建好的窗口。

一个窗口在很多方面都类似于一个 COM 对象。窗口对象使用内存，包装和实现了一组公用的函数，有一个特定的类型，一个惟一的标示，一个固定的生存期。窗口与 COM 对象的区别主要有：窗口有一个惟一的所有者，它是通过 API 函数来进行操作而不是通过方法，而且窗口通过一个函数就可以接受所有的事件通知。就像从外部调用一个 COM 对象时使用这个对象的指针一样，调用一个窗口对象需要使用一个窗口句柄或称为 HWND(handle to WINDOW)。我们也可以将 HWND 看作对一个窗口对象的弱引用。保存一个 HWND 并不意味着拥有这个窗口对象，拥有一个对 COM 对象的引用会保证对象的存活，而保存 HWND 却不同，Windows 系统拥有所有的窗口对象并且控制了它们的生存期。

对一个窗口对象的外部处理通过 Windows API 的子集实现，他们会把 HWND 当作第一个参数或者返回一个 HWND 的值。这组用来处理 HWND 的一组 API 函数是无序、松耦合的，这与紧密联系的 COM 中的 VTable、接口、方法等完全相反。所有的 HWND 都定义在 WIN32 API 中的 User32 部分。其他两组核心的 API 函数是处理操作系统核心功能如进程和线程管理的 Kernel32 以及绘图管理 GDI32。一个 User32 控制的窗口存活于 Kerne32 创建的进程中，并且是用 GDI32 画出来的。

在本章中会有许多 API 调用，尽管这里简要介绍了它们的核心概念，但这仍然需要读者自己去了解其他的相关调用。有许多很好的关于 API 的书适合不同层次技术水平的读者参阅。另外我还建议读者保持对所拥有的 MSDN 的更新，从而得到最完整的 API 信息。

在本书中，并没有试图去创建一个完整的类型库来进行 API 调用，因为本书不是一个完整的（甚至只是部分的）关于如何在 VB 中使用 Win32 API 的综述。本章中所引用的所有工程都将依赖于 Declare 语句来访问 Windows 中的 API 函数，但是如果读者愿意的话也可以使用类型库。在这里，我推荐使用 Bruce McKinney 著的 《HardCore Visual Basic》一书中的 API TypeLib。Bruce 已经决定不再继续那本书的第三版，也不再更新这个类型库，当然读者也不必过于烦恼，因为有很多高手还在维护这个类型库，我们可以在 Http://www.TheMandebrotSet.com 上找到这个类型库的最新版本及完成的源程序。

最常用的两个 HWND 函数是 SendMessage 和 PostMessage，其中前者用来对一个窗口发送一个串行消息，并且直到这个消息得到处理后才返回；而后者则用来在目标窗口的消息队列中添加一条新的消息，并且立即返回。Windows 系统会从消息队列中一次取出一条消息并将其发送给窗口对象的当前窗口过程。这个窗口过程是一个窗口所有事件的一个惟一入口点。所有通过 HWND 调用的函数最终都会生成一个或多个消息发送给窗口过程。WM_PAINT 消息是一个窗口过程经常收到的一个典型的消息，处理这个消息可以控制在客户端一个窗口的哪些部分会被刷新。显然，控制了窗口过程就控制了窗口。

16.1 子 类 化

我们必须控制窗口过程来定制一个窗口如何对它收到的消息做出反应。由于 Windows 系统允许我们在自己的进程中使用一个新的过程来替换任何窗口的窗口过程，这样就使得这一过程十分简单。窗口过程实际上是一个函数指针，可以用 SetWindowLong API 和 GWL_WNDPROC 索引来进行替换。子类化在概念上等于 VBoost 中使用的 IUnknown 钩子技术，但是相对要容易一些。这是因为我们仅仅需要替换一个函数，而不是替换整个虚表中的所有函数。替换窗口过程时，SetWindowLong 会返回先前的函数指针。我们自己的窗口过程应当处理我们自己需要处理的消息，然后使用 CallWindowProc API 函数来将所有其他的消息发送给先前的窗口过程去处理。为了保证正常的窗口销毁，我们需要在窗口销毁前重置起初的窗口过程。最基本的窗口过程就是简单的服从下一个窗口过程：

```
Function windowProc(ByVal hWnd As Long, ByVal uMsg As Long, _
 ByVal wParam As Long, ByVal lparam As Long) As Long
 'Debug.print Hex$(uMsg) 'Put useful code here.
 WindowProc = CallWindowProc(m_wndprcNext, hWnd, uMsg, _
                             wParam, l Param)
End Function
```

这个函数有一个问题就是 m_wndprcNext 变量没有进行定义。这就导致了我们在子类化时需要考虑的第一个问题：即我们如何将一个窗口过程和这个变量结合起来。窗口过程必须在一个 BAS 模块中，这样我们才能够使用 AddessOf，而 m_wndprcNext 的值对于我们子类化的每一个窗口都是不同的。如果我们在同一个模块中定义，那么我们仅仅能够子类化一个窗口，这样的一个子类化系统显然不是一个完整的方案。

m_wndprcNext 值是真正的窗口过程所需要的数据之一。如果我们需要子类化一个控件窗口从而定制该窗口的刷新，那么我们将需要这个控件实例的当前设置以及相关的特定 HWND 值。在理想情况下，我们希望将窗口过程定义为用户控件模块中的一个函数，这样我们就可以访问控件的私有成员变量来准确的画出控件。

HWND 参数是 Windows 系统用来处理窗口过程的惟一标志，由于这个原因，将一个窗口过程映射到实例数据中就需要使用 API 函数调用及 HWND 来保存一个对实例的弱引用。实例中的数据可以用 GWL_USERDATA 索引和 SetWindowLong 函数，或者使用 GetProp 和 SetProp 这些 API 函数来进行存储。对于这些技术，William Storage 和我曾在 1997 年 2 月的“Visual Baisc Programmers Journal”中讨论过，这种技术工作的很好，但是现在已经不需要这样做了。

子类化实际上需要的就是一个函数指针，它可以用来调用一个类模块中的友元函数并且不需要执行额外的处理来获得所需的实例。我在第 11 章中的“类函数指针”一节介绍了 PushParamThunk 概念，这个小小的动态生成的汇编代码使得数据不需要和 HWND 联系起来，因为实例数据在子定义生成的窗口过程中已经存在了。子类化仅仅是使用指针来调用一个基于实例的函数的一个特例。

现在我们可以很容易的使用 SetWindowLong 和 CallWIndowProc API 函数，以及一个 PushParamThunk 生成的窗口过程来子类化一个窗口。这里惟一缺少的是 HWND 本身，它是由 VB 中窗口对象的 hWnd 属性来提供的（Form.hWnd，Text1.hWnd，UserControl.hWnd 等等）。在程序清单 16.1 中，包含了 SubClass.bas 中的帮助函数（读者可从本书所附带光盘的源代码目录下找到该文件）、对这些函数的调用、一个窗体的友元窗口过程以及用来将调用重定向到窗体实例的一个简单 BAS 函数。

程序清单 16.1　如果我们使用 PushParamThunk 和 BAS 模块中的重定向来调用一个 CLS、FRM 或 CTL 模块中的窗口过程，子类化将很容易实现

```
'Support code from SubClass.Bas
'Requires:PushParamThunk.bas
'This listing does not show the debugging support
'discussed later in the chapter.
Public Type SubClassData
    wndprocNext As Long
```

```
    wndprocThunk As PushParamThunk
End Type

Public Sub SubClass ( _
  Data As SubClassData,ByVal hWnd As Long, _
  ByVal ThisPtr As Long,ByVal pfnRedirect As Long)
   With Data
       'Make sure we aren't currently subclassed.
       If .wndprocnext Then
          SetWindowLong hWnd,GWL_WNDPROC, .wndprocNext
          .wndprocNext = 0
       End If

       'Generate the window procedure function
       InitPushParamThunk.wndprocThunk,ThisPtr,pfnRedirect

       'Establish the new window function.
        .wndprocNext = SetWindowLong _
        (hWnd,GWL_WNDPROC, .wndprocThunk.pfn)
     End With
End Sub
Public Sub UnSubClass ( _
  Data As SubClassData,ByVal hWnd As Long )
    With Data
        'Restore the windows procedure to its original value.
        If .wndprocNext Then
        SetWindowLong hWnd, GWL_WNDPOC, .wndprocNext
       .wndprocNext = 0
    End If
 End with
End Sub
```

```
'Code in Form1
 Private m_SubClassMain As SubClassData

Private Sub Form_Load()
```

```
    'Use the helper function to establish the subclass.
    SubClass m_SubclassMain, _
     Me.hWnd, ObjPtr(Me), Addressof RedirectForm1WindowProc
    End Sub

Private Sub Form_Unload(Cancel As Integer)
    UnSubClass m_SubClassMain, Me.hWnd
End Sub

Friend Function WindowProc ( _
  ByVal hWnd As Long, ByVal uMsg As Long, _
  ByVal wParam As Long, ByVal lParam As Long) As Long
    'Watch the messages ( add real code here).
    Debug.Print Hex$(uMsg)
    'Defer to the original window procedure.
    WindowProc = CallWindowProc ( m_SubClassMain.wndproNext, _
              hWnd, uMsg, wParam, lParam )
  End Function
```

```
'Code in a BAS module .
Public Funtion RedirectForm1WindowProc(ByVal This As Form1, _
  ByVal hWnd As Long, ByVal uMsg As Long, _
  ByVal wParam As Long, Byval lparam As Logn) As Long
     'Redirect to the Form1 instance provided by the thunk.
     RedirectForm1WindowProc = _
      This .Windowproc(hWnd, uMsg, wParam, lparam)
  End Function
```

单纯从 API 调用的角度，子类化一个窗口对象十分的容易。在当前的过程链中引入一个新的窗口过程会导致增加一个小小的程序开销，在程序清单 16.1 中的整个操作包括 4 个附加的调用（Windows->Thunk->Redirect->WindowProc->CallWIndowProc->original window procedure）。这里，对函数的调用是直接的，也就是说，并不需要一个查找 VTable 的过程。我们常常使用一个 Select Case 语句来判断我们所需要检查的消息。如果其 Case 值恒定为长整型，那么 Select Case 语句运行起来将非常的快。在一个窗口过程中，一个正确的窗口过程的基准就是它除了对需要处理的消息以外，对其他的消息处理没有明显的性能影响。

有一些第三方的控件声称将子类化变得更为容易。在 VB5 中引入 AddressOf 以前，子

类化需要外部的帮助。但是现在，我们不需要外部控件的帮助便可以基本满足子类化需求了。如果一旦决定使用外部的组件来实现子类化，那么还有一些问题我们需要知道：

- 实例化一个控件的消耗比直接子类化要高得多。一个 PushParamThunk 是一个很小的嵌入结构（28 字节），它只消耗了为 UserControl 实例分配的内存。加载一个独立的控件需要为 VB 和这个控件本身分配内存，所以，控件实现子类化一定会比用 API 直接进行子类化要慢。
- 如果使用一个 ActiveX DLL 而不是一个控件来实现子类化，那么消耗将会小得多，但是这需要为每个窗体和控件编写代码，和使用 SubClass.Bas 的帮助函数差不多。
- 子类化控件通常需要轮询一组消息和标志，这样在消息触发时发出通知。这种实现需要对每个消息作轮询，不管这个过程做的多好，它不会比简单的编译过的 VB 中的 Select Case 语句更快。而且如果我们对不止一个消息感兴趣的话，还需要使用 Select Case 语句。
- 我们不应该用控件来通过事件过程发送消息通知。事件是基于 IDispatch 的，所以它们比直接的函数调用或者 VTable 调用都要慢的多。但在这里，性能并不是惟一的因素。如果用事件子类化，并在 IDE 中调试时显示一个 MsbBox 或者一个模式窗口，那么被模式窗口阻塞的应用程序其他部分的所有事件都将不再触发。如果我们通过事件过程子类化，则不会得到窗口消息的通知，并且使得控件不能正常工作。这对于具有商业化质量的控件来说是不可接受的，而且现在我们完全有办法去避免这种情况。在编译好的应用中，这种事件冻结的现象一般来说并不是问题。
- 使用回调接口的控件相比使用事件过程提高了很多，但是它们仍然需要对控件进行初始化。对每个消息轮询的开销，对相关消息使用 VTable 调用的花费，显然没有直接使用 SubClass.Bas 好。
- 将 SubClass.Bas 编译到整个工程中，从而不需要其他附加的文件。没有人知道实际上是用什么技术来工作的。
- 我们已经拥有 SubClass.Bas。
- 如果开发者搞不清楚子类化所需要的那些 API，也就意味着不能在 Windows 应用中使用任何有意义的 API 调用，这不是开玩笑吧？我只是想说子类化是使用 Win32 API 所能做的最容易的事之一。

16.1.1 调试子类化窗口

除了使用控件进行子类化的那些弊端外，还有一个好处便是我们可以使用断点来进行调试。使用 AddressOf 直接子类化的问题就是 VB 不会提供一个代码的实际函数指针，而是提供了一个模拟的过程，这样就阻止了在中断模式下运行 AddressOf 函数。虽然编译成 EXE 后没有这种情况，但是不幸的是，在 IDE 中确是如此。

如果在直接子类化时进入中断模式，VB 虽然没有崩溃，但是我们已经不能使用 IDE 或窗体来进行调试了。Windows 系统要求一个窗口过程在返回控制权时进行特定的动作，但是被 AddressOf 阻塞并返回了 0。如果幸运的话，可以通过按 F5 键来继续运行程序，要不然就需要杀死整个 VB 的进程然后重新开始。

幸运的是有一种方法来避免这种情况，并且已被包括在 SubCalss.Bas 中了。这种方法实现的思路也很直观，如果 VB 在调试模式下，那么应该使用原来的窗口过程而不是首先调用 AddressOf 函数。这时就好像没有进行子类化一样。DbgWProc.Dll 实现了这种功能。DbgWProc 起初是为了上文中曾提到过的那篇 1997 年 2 月的文章而编写的，在 MSDN 中也提到了并且可以下载。最初的版本在同一时刻仅支持 100 个子类化，但是这本书中用 PushParamThunk 代码去除了这个限制。

DbgWProc 很容易使用，而且对于子类化的代码来说，它没有运行时的额外开销，请读者按照以下的步骤来实现直接于类化时的正常调试：

（1）添加一个对 Debug Object for AddressOf SubClassing 的引用，如果没有这一项的话，请浏览 PowerVB\Tools 目录来添加这个引用。

（2）在工程属性对话框的 Make 项下，添加一个条件编译值 DEBUGWINDOWPROC=1。

（3）像原来一样使用 SubClass.Bas 中的 SubClass 和 UnSubClass。

（4）在选择文件/编译之前，将条件编译值设为 0。如果没有这样做而在 IDE 没有加载时使用了 DbgWProc.Dll，那么便会出现下述对话框“WindowProc debugging isn't required outside the Visual Basic design environment.”。这一信息对于程序的运行来说并没有影响，但是在这里，也许您并不愿意用户看到这个信息，而且 DbgWProc.Dll 也不应当随同应用程序一起分发。

关于如何对一个活动的子类进行调试，我们可以参考光盘上 SubClass.Bas 中的代码。当然，SubClass.Bas 已经做了所有的工作，所以这段代码也许不需要复制，仅需要记住代码会由于 VB 在调试模式还是运行模式而使用不同的执行路径就可以了。

表 16.1 不同调试和子类化状态下的子类化窗口的函数调用顺序

函 数	未子类化	无钩子	有钩子（运行的）	有钩子（中断的）
WindowsProcHook function	-	-	1	1
PushParamThunk assembly	-	1	2	-
Bas module redirection	1	2	3	1
Replacement WindowProc	-	3	4	-
Original WindowProc	1	4	5	2

既然已经可以调试了，那么现在就可以开始编写自己的窗口过程了。在使用子类化时，

请记住在运行程序之前保存工程。在工具/选项对话框中的一般情况下，取消“按要求编译”选项，通常会使得工程更加稳定。另外，我们也应当避免在调试状态下用停止按钮或者 End 语句来粗鲁的终止程序运行。如果遵守这些简单的步骤，并且所有的 API 调用正确的话，那么子类化的调试还是比较健壮的。

16.1.2 WM_GETMINMAXINFO 示例

既然已经可以控制窗口过程了，那么我们也就可以控制任何感兴趣的消息。在光盘上 Samples\SubClass 目录下，有四个基本的子类化的例子。Minimal 包含了上述代码，并以其作为起点；SysteMenu 演示了如何在系统菜单中添加项以及对消息进行响应；DrawItem 演示如何捕捉 WM_DRAWITEM 消息并对一个 CheckBox 风格的 ListBox 添加图片；MinMasInfo 则包含了这里所列举工程的代码。在这些工程中，基本技术都是一致的，区别只是在于窗口过程中的 API 代码。

Windows 系统在用户想要改变或最大化一个窗口时会向窗口过程发送 M_GETMINMAXINFO 消息。在相关的 MINMAXINFO 结构中可以指定窗口最小化和最大化的大小，并且可以定制最大化窗口的大小和位置。通常使用 WM_GETMINMAXINFO 消息来突破窗体的最小值限制。在 Form_Resize 过程中通常包含了许多出错处理代码，例如用于防止用户将窗口缩得太小而无法容纳所有的控件等等。处理 WM_GETMINMAXINFO 消息通过计算窗体的最小高度和宽度来避免在 Form_Resize 中包含太多的代码。

在对 WM_GETMINMAXINFO 消息进行编程时最大的一个问题便是通过窗口过程中 lParam 参数得到的 MINMAXINFO 结构的指针。在 C 或 C++中，只需要将 lParam 赋给 MINMAXINFO*，然后直接访问内存。然而在 VB 中我们不能这样做，最常用的方法便是用 CopyMemory 把数据拷贝到本地的 MINMAXINFO 结构中，修改本地的数据，然后再拷贝回去。在 MinMaxInfo 例子中，使用了 ArrayOwner 以允许直接访问指针，而不是两次调用 CopyMemory。 这样，我们便可以方便的直接访问内存，将 ByVal Long lParam 翻译成任意的类型。

程序清单 16.2 使用 WM_GETMINMAXINFO 消息来确保一个窗口不会改变大小而覆盖一个标签控件的右下角

```
'modMinMaxInfo (SubClass\MinMaxInfo\MinMaxInfo.bas)
'External dependencies:
'    ArrayOwner.Bas or ArrayOwnerIgnoreOnly.Bas
'    PushParamThunk.Bas
'    SubClass.Bas
```

```
'API declares omitted

'Type and variable to allow direct access to
'a pointer to a MINMAXINFO structure.
Public Type OwnedMinMaxInfo
    Owner As ArrayOwner
    pSA() As MINMAXINFO
End Type
Public g_DerefMinMaxInfo As OwnedMinMaxInfo

'Project/ Properties/ Gerneral/ Startup Object: = Sub Main
Public Sub Main()
 With g_DerefMinMaxInfo
     InitArrayOwner .Owner, LenB ( .pSA( 0)), 0
  End With
 frmMinMaxInfo.Show
End Sub

'Redirection function for PushParamThunk.
Public Function RedirectMinMaxInfoWindowProc ( _
 ByVal This As frmMinMaxInfo, -
 ByVal hWnd As Long, ByVal uMsg As Long, _
 ByVal wParam As Long, ByVal lParam As Long) As Long
    RedirectMinMaxInfoWindowProc = -
     This.WindowProc( hWnd, uMsg, wParam, lParam)
End Function
```

```
'frmMinMaxInfo (SubClass\ MinMaxInfo\ MinMaxInfo.frm)
'This form has a single Label control (lblMessage)
'dropped somewhere in the middle of the form,
Private m_SubClassMain As SubClassData
Private m_MinTrackSize As POINTAPI
Private Sub Form_Load()
    'Find the minimum tracking size by using the
    'Label.AutoSize property to determine the size of the
    'control, then calculate the pixel offset of the lower
    'right corner of the label from the upper left corner fo
    'the window. Note that the Left property of a control is
    'relative to the client area of the window, which needs
```

```
        'be adjusted by Width - ScaleWidth to get the positon
        'relative to the actual corner of the window. The
        'equivalent vertical calculation is done against the
        'Top property.
      With lblMessage
            .AutoSize = True
            .Caption = _
              "Try to resize the form to cover this text, "
              m_MinTrackSize.x = ( .Left + .Width + _
                   Width - ScaleWidth)  \  Screen.TwipsPerPixelX
              m_MinTrackSize.y = (.Top + .Height +_
                                   Height  -  ScaleHeight)    \
                \Screen.TwipsPerPixelY
            End With
            SubClass m_SubclassMain, Me.hWnd, ObjPtr(Me), _
                 AddressOf RedirectMinMaxInfoWindowProc
 End Sub
Private Sub Form_Unload(Cancel As Integer)
      UnSubClass m_SubClassMain, Me.hWnd
End sub
Friend Function WindowProc( _
    ByVal hWnd As Long, ByVal uMsg As Long , _
    ByVal wParam As Long, ByVal lParam As Long ) As Long
         Select Case uMsg
              Case WM_GETMINMAXINFO
                 With g_DerefMinMaxInfo
                    .Owner.SA.pvData = lParam
                    'Read off the cached minimum tracking size.
                    'All other values are preset by Windows.
                    .pSA(0) .ptMinTrackSize = m_MinTrackSize
                  End With
                  Exit Funtion 'Return 0 per MSDN.
            'Add other messages here.
       End Select
       WindowProc = CallWindowProc (m_SubClassMain.wndprocNext, _
         hWnd, uMsg, wParam, lParam)
End Function
```

16.2 自定义窗口的创建

子类化一个现存的窗口使得我们可以完全控制该窗口。然而，完全控制这个消息流还不能够让人们修改窗口的类型以及它创建时的属性。我们必须通过指定一个窗口的各个属性，并且调用 CreateWindowEx API 来创建它。调用 CreateWindowEx 很简单，但是 VB 中并不允许创建自己的窗口。VB 也不会帮助我们在一个定制创建的窗口和 VB 本身创建的窗口之间实现交互，让 VB 和定制的窗口交互是创建定制窗口中的一个难点。

也许读者会觉得很奇怪，为什么需要调用 CreateWindowEx 来创建自己的窗口。这是因为 VB 中有丰富的内置控件，可构建定制的 ActiveX 控件，并且提供了很灵活的手段（可以子类化）。然而除此以外，主要有三个方面缺少必要的控制：

- 绝大多数内置控件的属性在运行时是只读的，那些在底层窗口上的风格设置必须在窗口创建时设定。例如，我们在 TextBox 创建后，就不能改变其 MultiLine 属性，我们也不能够用代码来改变 ListBox 的 MultiSelect、Style 或 Sorted 属性。有许多基本的控件属性在运行时是只读的（尽管 VB 已经消除了一些）。因为我们只能使用默认的属性来添加内置的控件，这样在使用 new Controls.Add 特性时就尤其显得受限制。通过创建自己的 LISTBOX 或 EDIT-style 窗口而不是使用内置的 ListBox 或 Edit 控件可以销毁和重新创建窗口，这样也就能够改变只读的窗口风格。
- 即使在运行时所有 VB 控件的属性都是可读写的，也不是所有的窗口风格在 VB 的对象中都有相对应的属性。例如，系统的 LISTBOX 窗口类支持 LBS_NODATA 风格，允许完全虚拟，这样我们就可以显示任意多的项而无需首先将所有的数据拷贝到 LISTBOX 中去。然而，VB 中的 ListBox 控件没有这样的 NoData 属性，VB 也不提供接口来生成虚拟的 ListBox 控件。类似的，common shell 控件可以创建一个完全虚拟的 ListView 控件，而在 VB 中没有这些设置。CreateWindowEx 允许我们利用系统提供高级特性来创建为自己的应用所用的高效控件。
- 所有 VB 的控件都是基于 ANSI 的，有一些版本的 Windows 提供了完全的 UNICODE 支持，如果要让一个支持 UNICODE 的应用在 Windows NT 和 Windows 2000 上运行，并且使用内置的 ANSI 控件，可能会引起很多问题。

一旦创建了自己的窗口，控制它的代码与子类化一个现存的窗口所需的代码类似。在一个新的窗口上调用 SetWindowLong 来建立自己窗口过程，在收到的消息流中忽略一些消息，将那些不需处理的消息通过 CallWIndowProc 或 DefWindowProc 发送出去。对于自创建的窗口与父窗口的交互需要额外的代码。一般来说，我们不需要创建最上层的窗口，例如窗体，这是因为在这个部分没有很多可以定制的东西。一般地，我们更倾向于在 VB 创建的父窗口上创建 WS_CHILD 类型的新窗口。

除了使用 VB 所创建的父窗口外，我们还需要另外一个窗口，它需要能够与 VB 在窗体上创建的控件进行交互。如果我们不能够将自己所创建的窗口与这样的窗口联系起来，也

就不能将自己的控件放入窗体的 Tab 顺序中去。要满足这种要求，最常用的方法便是在一个 UserControl 中调用所有的 CreateWindowEx。将窗口的创建和 UserControl 结合起来使得我们可以利用一个 UserControl 窗口来参与 Tab 顺序，与 VB 的父窗口进行交互。

这里，我们必须了解一个 UserControl 对象运行时和设计时的需求。在设计时需要一个表现完全功能的控件，很少需要创建一个额外窗口或执行一个子类化。问题在于：在 UserControl_Initialize 过程中并不知道现在究竟是在设计时还是运行时。这个信息只能通过 Ambinet 对象来获得，但是这个对象要等到控件创建过程的晚期才能够得到。

UserControl_IniProperties 和 UserControl_ReadProperties 事件在 Ambinet 属性可以得到时提供了一个可靠的入口点。当一个控件创建时，下列事件被触发：一个新控件的 InitProperties 事件；缓存了一组属性控件的 ReadProperties 事件。为了防止在这两个事件中的重复代码，我们需要在 Calss_Initialize 而不是 UserControl_Initialize 中创建一个帮助程序。程序清单 16.3 为本书中的控件示例提供了一个起点。它提供了对设计模式的检查以及用于在设计时控件上打印当前控件名字的代码。

程序清单 16.3 UserControl 的起始代码（正确的处理设计模式及运行模式）

```
Private m_fDesign As Boolean
Private Sub UserControl_InitProperties()
        InitializeMode
End Sub
Private Sub UserControl_ReadProperties (PropBag As PropertyBag)
        InitializeMode
End Sub

Private Sub SetDesignMode()
      On Error Resume Next
      m_fDesign = Not Ambient.UserMode
      If Err Then m_fDesign = True
      On Error Goto 0
End Sub
Private Sub InitializeMode()
   SetDesignMode
   If Not m_fDesign Then
         'Subclassing and aggregation calls go here.
   End If
End Sub
```

```
Private Sub UserControl_Paint()
    If m_fDesign Then
        With UserControl
            'Assume a scalemode of Pixels.
            .CurrentX = 4
            .CurrentY = 2
            UserControl.Print .Ambient.DisplayName
        End With
    End If
End Sub
Private Sub UserControl_AmbientChanged (PropertyName As String )
     If m_fDesign Then
     If PropertyName = "DisplayName" Then _
        UserControl.Refresh
  End If
End Sub
```

现在我们可以在运行时，在 UserControl 中创建窗口了。下一步是调用 CreateWindowEx，然后是子类化定制窗口和 UserControl 窗口。这里需要同时子类化以便让焦点可以通过 UserControl 窗口到达定制窗口。这里，有两种情况需要注意：一种情况是，当 UserControl 通过键盘的 Tab 键或其他加速键获得焦点时，我们需要处理主窗口的 WM_SETFOCUS 消息，然后用 SetFocus API 将控制传递给定制窗口。第二种情况是，当用户在定制窗口中通过鼠标直接激活控件时，需要拒绝 WM_MOUSEACTIVATE 消息，并将焦点设定到 UserControl，从而使得定制窗口以和第一种情况一样的方法激活。具体细节请读者参阅程序清单 16.4。

在光盘上的 Samples\CreateWindow\MinimalEdit 目录下，我们可以得到一个最简单的创建定制窗口的完整的框架。这个例子创建了一个简单的 EDIT 窗口。在这个例子中，忽略了这个控件的公共函数和属性，从而为我们自己创建工程提供了一个很好的起点。Samples\CreateWindow 下的 LboxEx 例子提供了一个虚拟的 listbox 控件。它也是对 1997 年 6 月的 "Visual Basic Programmer's Journal" 中示例的更新。最初的 LBoxEx 在 IE 中会崩溃（它们的 bug，我的问题），而且在一个 MDI 的子窗体中工作不正常。这些问题已经用本章中所讲的技术进行了修复。

程序清单 16.4 在 UserControl 窗口及 do-it-yourself 窗口之间扮演 hot-potato 的代码，节选于 MinimalEdit 样例程序

```
Private m_SubClassEdit As SubClassData
Private m_hWndEdit As Long
Private m_SubClassParent As SubClassData
Private m_hWndParent As Long

Friend Function WindowProcParent ( _
  ByVal hWnd As Long, ByVal uMsg As Long, _
  ByVal wParam As Long, ByVal lParam As Long) As Long
     Select Case uMsg
          'Special code to get the correct background
          'color on an edit control.
        Case WM_CTLCOLOREDIT
           'Bypass VB on this message. It doesn't let it
           'drop through.
           If lParam = m_hWndEdit Then
               WindowProcParent = DefWindowProc ( _
                     hwnd, uMsg, wParam, lParam)
                Exit Funtion
             End If
     End Select

     'Note: You can achieve something similar here without he
     'parent window subclass by calling SetFocusAPI m_hWndEdit
     'in UserControl_GotFocus. However, with the subclass, the
     'focus is set correctly on the child window before
     'you reach any of its events, and you will generally need
     'to handle other messages in the parent window anyway.
    WindowProcParent = CallWindowProc( _
        m_SubClassParent.wndprocNext, _
        hWnd, uMsg, wParam, lParam)
     'Bounce focus to the child edit window.
     If uMsg = WM_SETFOCUS Then SetFocusAPI m_hWndEdit
```

```
    End Function

    Friend Function WindowProc( _
        ByVal hWnd As Long, ByVal uMsg As Long, _
        ByVal wParam As Long, ByVal lParam As Long ) As Long
          Select Case uMsg
            Case WM_MOUSEACTIVATE
              If GetFocus <> m_hWndEdit Then
                'Bounce focus back to the UserControl window
                 'so that VB knows where the focus is . The
                 'parent window will then bounce control right
                 'back to us.
                  SetFocusAPI m_hWndParent
                  WindowProc = MA_NOACTIVATE
                  Exit function
              End If
        End Select
        WindowProc = CallWindowProc( m_SubClassEdit.wndprocNext, _
          hWnd, uMsg, wParam, lParam)
End Function

Private Sub InitializeMode()
      SetDesignMode
      If Not m_fDesign Then
          If m_hWndEdit Then Exit Sub
          With Usercontrol
                m_hWndEdit = CreateWindowEx( _
                   WS_EX_CLIENTEDGE, "EDIT", vbNullString, _
                   WS_CHILD Or WS_CLIPSIBLINGS Or WS_TABSTOP, _
                   0, 0, .ScaleWidth, .ScaleHeight, .hWnd, 0, _
                   App.hINstance, ByVal 0&)
                  If m_hWndEdit Then
                    SubClass m_SubClassEdit, m_hWndEdit, _
                     ObjPtr(Me), AddressOf RedirectEditProc
                    m_hWndParent = .hWnd
                    SubClass m_SubClassParent, m_hWndParent, _
```

```
                    ObjPtr(Me), AddressOf RedirectEditProcParent
                    ShowWindow m_hWndEdit, SW_SHOW
                End If
            End With
        End If
End Sub
Private Sub UserControl_Terminate()
    If Not m_fDesign Then
        If m_hWndParent Then _
            UnSubClass m_SubClassParent, m_hWndParent
        If m_hWndEdit Then _
            UnSubClass m_SubClassEdit, m_hWndEdit
        m_hWndParent = 0
        m_hWndEdit = 0
    End If
End sub
```

如果仅仅是用上面的代码运行一个简单控件的测试，可能已经完成了必需的工作。实际上，如果我们的控件是窗体上的惟一控件的话，那么一切正常。但是如果在窗体上添加了其他的控件的话，那么就会有问题了，这时你会发现用箭头键会把焦点移到窗体上的下一个控件，而不是在控件中移动。由于已经使用了子类化，我们可以很容易的监测WM_KEYDOWN消息从而做出相应的处理。

这里的问题便是：在聚合器转换过程中，丢失了所有这些先前的键击事件，并且这发生在任何窗口过程被调用之前。我们自己创建了EDIT窗口，但是VB不知道这个窗口的存在，因此VB认为在UserControl上没有其他的控件了，这时箭头键就类似于Tab键，使得焦点移到下一个控件。为了正确响应箭头键，就需要在处理键击事件前取得控制。在一个ActiveX控件中意味着在IOleInPlaceActiveObject接口上取得对TranslateAccelerator方法的控制。

对IOleInPlaceActiveObject的支持内置于VB UserControl的实现中，我们可以用Set语句来取得这个接口。通常会用VBoost对象或QIHook来覆盖这个接口的实现。不幸的是，VB不是调用QueryInterface来取得当前活动的对象，而是依赖于直接的内部调用，因此钩住IUnknown也没有什么帮助。如果VB确实是使用这个接口的话，那么我们可以在创建这个窗口时钩住它，以后就不用再考虑了。但是现在没有对IUnknown的钩子，我们就不得不获得WM_SETFOCUS消息并且深入VB编程，用定制实现替换掉当前的活动对象。当控件失去焦点时，才会结束这次替换并回到起初的状况。

本书附带光盘中的Code\InPlaceActiveObjectHook.Bas文件提供了一个对IOleInPlaceActive

Object 接口的覆盖实现。IPAOHook 是一个直接的对象实现了除 TranslateAccelerator 方法以外的其他所有实现。TranslateAccelerator 的实现是通过 OleTypes.Olb(VBoost: Ole Type Definitions)中 IHookAccelerator 接口的对应函数实现的。我们在使用 IHookAccelerator 实现前需要对 IPAOHook 对象进行初始化，关于所需的代码，请参考程序清单 16.5。

程序清单 16.5 用于在 VB 得到加速消息之前发送它们的附加代码。该代码要求一个对 OleTypes.Olb 和 InPlaceActiveObjectHook.Bas 的引用

```
    Implements OleTypes .IHookAccelerator
    Private m_IPAOHook As IPAOHook

    Friend Function WindowProc ( _
      ByVal hWnd As Long, ByVal uMsg As Long , _
      ByVal wParam As Long, ByVal lParam As Long ) As Long
       Select Case uMsg
          Case WM_SETFOCUS
              OverrideActiveObject m_IPAOHook, Me
          Case WM_MOUSEACTIVATE
        'Other code as in Listing 16.4.
    End function
  Private Sub IHookAccelerator_TranslateAccelerator ( _
    lpmsg As OleTypes.MSG, hrReturnCode As Long)
       'Return code defaults to SFALSE (1).
       With lpmsg
            If .message = WM_KEYDOWN Then
              Select Case LOWORD(lpmsg.wParam)
                Case vbKeyUp, vbKeyDown, vbKeyLeft, _
                     vbKeyRight, vbKeyPageDown, vbKeyPageUp, _
                     vbKeyHome,  vbKeyEnd
                     'Process now.
                     DoMsg .message, .wParam, .lParam
                      'Indicate that we've handled the message.
                     hrReturnCode = 0
              End Select
            End If
        End With
  End Sub
  Private Function DoMsg(ByVal uMsg As Long , _
```

```
    Optional ByVal wParam As Long = 0, _
    Optional ByVal lParam As Long = 0) As Long
        'Go straight to the previous window procedure instead of
        'calling SendMessage. This prevents unnecessarily stepping
        'through our window procedure.
        DoMsg = CallWindowProc (m_SubClassEdit.wndprocNext, _
            m_hWndEdit, uMsg, wParam, lParam)
    End Function
    Private Sub InitializeMode()
        'Other code as in Listing 16.4.
        If Not m_fDesign Then
            InitializeIPAOHook m_IPAOHOOK,Me
        End If
End Sub
```

现在我们对定制的窗口以及其父窗口的加速消息，窗口过程等都有了完全的控制，这时就可以对添加控件的属性、方法和事件与定制的 hWnd 交互并且响应消息。在现在的状况下，不足的就是创建定制的窗口使用的资源问题。我们实际上创建了比想要的窗口多一倍的窗口，如果程序在 Windows95/98 上运行，并且在许多窗体上使用了很多有窗口的控件话，则很容易导致资源不足，因为这些操作系统中窗口的数量以及设备上下文的句柄有限制。

对于窗口句柄，我们已经很熟悉了，但是对设备上下文还没有提及。设备上下文是一个代表了绘图表面的对象，例如一个打印机或者在屏幕上的一个区域。在 Windows 系统中可以使用临时的设备上下文对象（需要时创建，然后返回系统），也可以使用每个 HWND 来创建和销毁设备上下文。VB 缺省的使用后者，但是使用这种缺省设置意味了每个窗体和 UserControl 都在宝贵的系统资源中创建了两个句柄。为了克服这个问题，在 VB 中提供了 HasDC 属性。这个属性让 VB 不再缓存设备上下文，而是借用它，这样就将实际存在的句柄减少了一半。虽然这样做会导致重画速度的降低，但是在大多数情况下人们不会注意到这种区别。在这里，推荐将所有新的窗体和 UserControl 的 HasDC 属性设为 False，从而减少句柄的数量。如果感觉到了重画速度的降低，可以通过将 ClipControls 的值从缺省的 True 改为 False 来提高窗体的重画性能。

16.3 无窗口的控件

VB6 中为了减少资源的消耗，可以使用控件的两个属性，HasDC 是其中之一，另外一个是 WindowLess 属性。一个无窗口的控件使得我们可以不使用 HWND 来创建一个新的 UserControl。它只是存在于其他窗口中的一个区域。无窗口控件最初出现在 VB2 中的 Label

和 Image 控件，我们也可以为 16 位的应用创建无窗口的 VBX 控件，然而随 VB4 而来的 ActiveX 控件中没有无窗口控件。VB5 中增加了对无窗口控件的支持，不过在 VB5 中，我们只是可以使用无窗口控件而并不能够创建它们。只有 VB6，才是第一个可以真正用来创建无窗口 UserControl 对象的版本。

这种创建无窗口的能力使得我们可以在一个窗体上使用更多的控件。但是不需要 HWND 并不意味着无窗口的控件比有窗口的控件少使用很多资源，在启动时它们的消耗几乎相同。换而言之，虽然使用它们没有什么限制，但是没有理由去创建根本不需要的窗口。

在控件上进行用户自定义的绘画对无窗口的控件很有用。很花哨的绘图一般用 Window API 来实现，但是使用无窗口的控件会遇到一些正常控件没有的问题。首先，我们需要理解如何在控件的表面画图而不会超过控件的边界；第二，不能够访问窗口消息流；第三，当控件激活时，VB 没有可靠的方法来通知。尤其是第三点使得用一般的用户界面元素在无窗口控件上画图很困难，例如脱字符号或者一个高亮显示的矩形选择区域。我们可以直接访问 IOle*接口来解决这些问题，但是要用到 VBoost 的一点帮助。

16.3.1 找到矩形

在开始使用 API 画图之前，我们需要 HWND、画图的矩形区域以及对设备上下文的访问。在无窗口的控件中，UserControl.hWnd 属性总是返回 0，因此我们需要用 UserControl.ContainerHwnd 属性来判断在哪个窗口里画图。当然，仅有这些还不够，因为我们只能使用窗口的一部分，而不是全部窗口。在区域之外画图是一个非常不好的习惯，我们可以在容器的 HWND 中利用下面的函数来获得所需的矩形区域。

```
'A helper function to retrieve the in-place site. This is
'in a helper function because this interface is useful for
'more than just finding the rectangle.
Private Function WindowlessSite() As IOleInPlaceSiteWindowless
Dim pOleObject As IOleObject
    Set pOleObject = Me
    Set WindowLessSite = pOleObject. GetClientSite
End Function

'Given the site, find the rectangle relative to
'the contrainer's window.
Private Function CtlRect( _
   PSite As IOleInPlaceSiteWindowless) As RECT
Dim pFrame As IOleInplaceFrame
```

```
Dim pDoc As IOleInPlaceUIWindow
Dim ClipRect As RECT
Dim FrameInfo As OLEINPLACEFRAMEINFO
   FrameInfo.cb = LenB(FrameInfo)
   pSite.GetWindowContext _
     pFrame, pDoc, CtlRect, ClipRect, FrameInfo
End Function

'Calling code
Dim rct As RECT
    rct = CtlRect (WindowlessSite)
```

在取得了控件的矩形区域之后，我们需要一个设备上下文来调用 GDI 的绘图 API。熟悉 API 编程的读者，这时首先想到的是对容器的 HWND 用 GetDC 函数，在绘图结束后调用 ReleaseDC。但是在这里不能这样做，因为 DC 允许在容器的窗口中绘图，但是它不考虑覆盖它的其他控件。这里需要对 IOleInPlaceSiteWindowless 接口使用 GetDC 和 ReleaseDC，而不是直接调用它们。这样返回的 DC 已经被正确分割了，这时再绘图就不用担心会在不允许的地方留下什么痕迹。用这个函数返回的 DC 的原点在窗口的左边，而不是在控件的左边，所以需要用 CtlRect 函数来得到正确的矩形。

由于 UserControl 已经提供了一个内置的 hDC 属性，上面的 GetDC/RelaeseDC 看起来有点多余。内置的 DC 也已经正确分割了，但是它的原点是相对于控件而不是相对于窗口。如果 ScaleMode 设为 Pixels，那么使用 hDC 的矩形区域是{{0,0},{ScaleWidth, ScaleHeight}}。为了在一个未分割的区域内画图，需要使用正确的 DC 以及正确的矩形。内置的 DC 在一般情况下可以满足基本的需求，但是对于一些需要的功能可能性能没有优化。

- GetDC 有几个 hDC 属性没有的选项。GetDC 允许指定比整个控件小的矩形区域，它还提供了三个 hDC 属性没有的选项：OLEDC_NODRAW，OLEDC_PAINTBKGND，OLEDC_OFFSCREEN。容器可以使用或忽略这些标志。VB 只支持 OLEDC_NODRAE 标志，这个选项适合于用 DC 只是用来获得像文本宽度这样的一些信息而不是用来画图的情况。
- 在使用多个重叠的控件进行严格的测试时，我发现使用 GetDC 的屏幕闪烁要少一些。用肉眼很难判断这种类型的性能差异，因此这不是一个科学的推断，只是一个观察的结果。
- 如果控件的 BackStyle 属性设为 Transparent，在 UserControl_Paint 事件中使用 GetDC 不会有输出，此时必须用 hDC 属性画图并且与对应的矩形相关联。

16.3.2 获得消息

我们不能像子类化普通的有窗口的控件一样来子类化无窗口的控件。然而 ActiveX 控件的规范允许一个无窗口的控件通过 IOleInPlaceObjectWindowless 接口的 OnWindowMessage 方法来得到在它的区域内发生的消息。像通常的消息流一样，VB 不允许我们自动的访问这些消息，但是我们可以通过提供这个接口的定制实现来获得这些消息。与 IoleInPlaceActiveObject 在控件获得焦点时需要重新使用钩子不一样，VB 的控件容器通过 QueryInterface 来取得 IOleInPlaceObjectWindowless。我们可以使用 VBoost 聚合器，并利用第 12 章中讨论过的接口重载技术来访问这个消息。

Code\InPlaceObjectWindowLessHook.Bas 文件定义了一个称为 WindowlessSubclas 的类型，它展现了一个被称为 IntializeWindowlessSubclass 的公共函数（原型见第 8 章“聚合轻量对象”一节）。聚合所需的全部标识和对象都已经做好了，我们只需要将 AggregateData 中的一个元素及 IID 数组传递给 VBoost.AggregateUnknown 调用。这使得我们可以采用一个简单的 UnknownHook 来聚合其他的对象。这段代码使用了 delayCreation 接口的轻量实现，从而可以方便的获得接口，实现样例请参阅程序清单 16.6。

当一个活动的WindowlessSubclass对象在VB的VBoost实现上运行时，我没能够成功的对控件进行调试。如果在调试时VB崩溃了，可以将VBOOST_INTERNAL条件设为0。

无窗口控件的事件模型中存在的一个漏洞，迫使我们需要子类化钩子。当我们用 Tab 键或者在一个无窗口的控件上单击时，会得到一个 GetFocus 事件。然而如果切换到其他的程序或窗口，然后再重新激活原来的窗体，就不会接受 GetFocus（或其他）事件。这使得 GetFocus 和 LostFocus 事件不能够用来表示一个控件是否在激活状态。可是如果我们观察一下 OnWindowMessage 的消息流的话，可以发现除了用户选择窗体上的控件，窗体本身在失去焦点和重新激活时，可以得到 WM_KILLFOCUS 和 WM_SETFOCUS 事件。这使得我们可以完成像显示或隐藏焦点矩形这样的动作。我们可以使得无窗口控件的外观和感觉，就像完全可以访问消息流的普通有窗口控件一样。

程序清单 16.6　使用 WindowlessSubclass 对象来获得被激活通知要比 GotFocus 事件更为可靠。正如在一个标准子类中对 WM_SETFOCUS 消息的处理一样，我们可以利用 WM_SETFOCUS 消息来安装一个加速器钩子

```
'Member variables to get OnWindowMessage.
Implements OleTypes. IHookWindowlessMessage
Private m_WLSC As WindowlessSubclass
```

```
Private m_Hook As UnknownHook
Private Type HookData
     AggData(0) As AggregateData
     IIDs(0) As IID
End Type

'Member variables to get accelerator notifications.
Implements OleTypes, IHookAccelerator
Private m_IPAOHook As IPAOHook

'Called from InitProperties and ReadProperties,
Private Sub InitializeHooks()
Dim HookData As HookData
  If Ambient.UserMode Then
       InitializeIPAOHook m_IPAOHook, Me
       With HokData
           'Notify the hook after forwarding he message
           'to VB's implementation.
            InitializeWindowlessSubclass m_WLSC, Me, _
              nmNotifyAfter, .AggData(0), .IIDs(0), 0
            VBoost.AggregateUnknown Me, .AggData, .IIDs, m_Hook
      End With
   End If
End sub

Private Sub IHookWindowlessMessage_OnWindowMessage( _
  ByVal fBefore As Boolean, _
  ByVal uMsg As VBoostTypes.UINT, _
  ByVal wParam As VBoostTypes.wParam, _
  ByVal lParam As VBoostType.lParam, _
  plResult AS OleTypes.LRESULT, _
  hrReturnCode As Long)
     'This is always called with fBefore False because the
     'hook was established with nmNotifyAfter. nmNotifyBefore
     'can also be used to get the first shot at the message.
     'If fBefore is true, an hrReturnCode of 0 stops
```

```
        'further processing.
        If uMsg =WM_SETFOCUS Then
            OverrideAcriveObject m_IPAOHook, Me
        End If
    End Sub
    Private Sub IHookAccelerator_TranslateAccelerator( _
      lpmsg As OleTypes.MSG, hrReturnCode As Long)
    Dim plResult As LRESULT
    Dim pIPOWL As IOleInPlaceObjectWindowless
        'Return code defaults to S_FALSE (1) .
        with lpmsg
          If .message = WM_KEYDOWN Then
            Select Case .wParam And &HFFFF& 'LOWORD of wparam
              Case vbKeyUp, vbKeyDown, bKeyLeft, _
                  vbKeyRight, vbKeyHome, vbKeyEnd
                  'Forward directly to the control instead
                  'of the usual window handle.
                  Set pIPOWL = Me
                  hrReturnCode = pIPOWL.OnWindowMessage( _
                    .message, .wParam, .lParam, plResult )
            End Select
          End If
        End with
End Sub
```

使用 OnWindowMessage 和 TransalateAccelerator 钩子，我们可以使用无窗口的控件来完成任何我们想做的事情。在光盘上有两个例子使用了这种技术。第一个位于 Samples\CreateWindow\MinimalEditWindowLess 目录下，这是 MinimalEdit 工程的修改版本。该版本中使用了 CreateWindowEx，并用一个无窗口的 UserControl 作为一个有窗口的 EDIT 控件的宿主。整个程序完成了在 VB 中用一个简单的窗口句柄来创建一个定制窗口的任务。第二个例子则位于 Samples\WindowLessLightEdit 目录下，是一个完整的无窗口 Edit 控件，它提供了一个支持 UNICODE 的编辑界面，可以在任何背景上透明的使用，而且我们可以通过简单的扩展让它支持多行文本等其他特性。使用了激活和加速钩子，如果不用 SpyXX 这样的工具来扫描窗口句柄的话，无窗口的控件和有窗口的控件几乎无法分别。

附 录

VBoost 参 考

VBoost 作为一组对象和函数，提供了在 VB 中不能够直接实现的功能，VBoost 对象主要有三类：

- VB 中不能直接实现的处理数学和赋值操作的函数及操作。
- 提供变通的内存分配手段的对象。
- 用来聚合对象的对象。

除了函数本身和它们的代码，本书还提供了许多函数声明使得在 VB 中创建 VBoost 对象非常方便。同时对于书中其他一些方面的编码也提供了足够的声明。例如，这里包括了所有的 SafeArray API、GUID 操作函数、内存分配函数等。

VBoost 对象的类型声明和对象的实现是分开的。所有的类型声明都在 VBoostTypes6.olb 文件中，VBoost 的 C++声明都在 VBoost6.Dll 中。这个 Dll 文件可以任意分发，但是这个 OLB 文件不可以分发（除非您把这本书作为附带的礼物）。

除了在 VBoost6.Dll 中提供的实现外，还有一个完整的对 VBoost 对象的实现。读者可以把这些实现放到自己的工程中，或者和自己的工程一起来分发这个 DLL。另外这个 VB 实现提供了一个条件编译开关来减少最后生成的工程的大小。

为了像本书中的例子一样使用 VBoost 函数，需要在工程的启动部分初始化一个称为 VBoost 的全局变量。为了减小使用 VBoost 函数的消耗，请显式的生成 VBoost 变量，而不是采用 New VBoostRootImpl 语法。首先，使用 New 声明只能应用于 C++的实现，因为 VBoostRoot 对象在 VB 中不能用 New 来创建；第二，即使是使用 C++的实现，显式创建也可以避免额外生成的代码。

下述步骤使得可以在 C++和 VB 中实现 VBoost，而且可以在两者之间切换：

（1）打开工程/引用对话框，添加对“VBoost Object Types(6.0)”和“VBoost Object Implementation(6.0)”的引用。这些类库分别包括在 VBoostTypes6.Olb 和 VBoost6.Dll 中。

（2）打开工程/属性对话框。

（3）在通用 tab 下，选择 Sub Main 作为启动对象。

（4）在生成 tab 中，在条件编译常数字段添加 VBOOST_INTERNAL=0，注意多个值

之间是用冒号分隔的。

（5）把 VBoost.Bas 文件添加到工程中。

（6）在 Sub Main 过程中调用 InitVBoost 函数。

（7）在使用 VB 实现而不是 DLL 时，将 VBOOST_INTERNAL 的值设为 1，如果使用 VB 实现，不需要分发 VBoost6.DLL，VBoostTypes6.Olb 不能分发。

如果不想要 Sub Main，可以在任何使用它的对象中调用 InitVBoost。调用多次也可以，但是由于这本书中的许多例子都假设已经初始化了 VBoost 引用，因此应该在比较早的时候调用 InitVBoost。如果准备分发 VBoost6.Dll，那么就不需要在工程中添加 VBoost.Bas 文件，可以用如下的代码来初始化 VBoostRoot 对象。

```
Public VBoost As VBoostTypes.VBoostRoot
Sub Main()
    Set VBoost = New VBoost6.VBoostRootImpl
     'Other project specific code here.
End Sub
```

如果使用 VB 的实现，可以通过添加额外的条件编译常数来减少生成的代码。可以用 VBOOST_BLIND_#常数来减少支持的 VTable 项目数，这里#是{1024，512，256，128，64，32}。为了更好的控制 VTable 的大小，可以创建一个 VBoost.Bas 的本地版本，然后修改 cBlindVTableEntries 常数。也可以通过将 VBOOST_CUSTOM 常数设置为非零值来取消对大多数函数的支持。在下面的条件编译的常数表中，如果 VBOOST_CUSTOM 是 0 或者没有定义，那么所有 VBOOST_CUSTOM 下面的值都被忽略。如果仅仅设置 VBOOST_CUSTOM，生成的代码将会很小，在 EXE 文件的大小都不会明显的反映出来。

如果调用了一个没有包含进来的函数，将会通过 Debug.Assert 调用在 IDE 里中断。但如果引发了这一断言，可以用查看/调用堆栈对话框来判断究竟丢失了什么函数。这时必须避免用停止按钮来结束程序运行，跳出这个断言函数，用 Set Next Statement(Ctrl-F9)和 Step Over Error(Alt-F8)命令来从失败的函数调用中正确退出。如果跳过了正常的结束代码，本书中的大多数代码包括 VBoost 将会导致 IDE 崩溃或者进入一个不可用的状态。

不管选择 VB 实现还是 C++的实现，现在我们都可以使用 VBoost 对象了。C++对象在调试时要稳定一些，所以可以在开发时使用 C++对象，而在生成最终的可执行程序时使用 VB 对象。让我们来看看 VBoostRoot 对象中的所有函数，以及与之相关的结构和常数。

赋值和算术函数

在无符号的数学运算以及不同类型之间的赋值时使用第一组函数。尽管有一些函数可以在 VB 中直接实现，但是使用 VBoost 要比 VB 更快。在 VB 版本中，除了 AssignAddRef 和 SafeUnknown，其他的函数都是对汇编字节数组的指针。

附表 1 VBoost.Bas 条件编译常量

条件常数	行为
VBOOST_INTERNAL	用 VB 实现来取代 New VBoostRootImpl
VBOOST_BLIND_#	限制所产生的 blind VTable 为 {1024,512,256,128,64,32}
VBOOST_CUSTOM	移除 Assignment 和 Arithmetic 之外的所有函数
VBOOST_Aggregates	支持 AggregateUnknown 和 CreateAggregate
VBOOST_AggregateUnknown	仅支持 AggregateUnknown
VBOOST_CreateAggregate	仅支持 CreateAggregate
VBOOST_Hooks	支持 HookQI 和 HookQIAR
VBOOST_HookQI	仅支持 HookQI
VBOOST_HookQIAR	仅支持 HookQIAR
VBOOST_Memory	支持压缩和非压缩的 CreateFixedSizeMemoryManager
VBOOST_Memory_Simple	支持 CreateFixedSizeMemoryManager （fcompactible 为 False 时）
VBOOST_Memory_Compactible	支持 CreateFixedSizeMemoryManager （fcompactible 为 True 时）
VBOOST_BLIND	支持 CreateDelegator 和 BlindFunctionPointer

1. Assign(pDst As Any, pSrc As Any)

赋值函数对于本书中的代码来说是基本的，Assign 在功能上等同于“CopyMemory pDst, pSrc,4” 但是要更快。最常见的赋值的大小是 4，当然我们还可以用 CopyMemory 来处理其他的大小。Assign 在两个变量之间传递数据，不进行类型检查。可以用四种方法来调用这个函数，请参阅下面的例子（含 C 实现）。

```
Dim pDst As Long, pSrc As Long
' (void*) pDst = (void*)pDst
VBoost.Assign pDst, pSrc
'*(void**)pDst = (void*)pSrc
VBoost.Assign ByVal pDst, pSrc
'(void*)pDst = *(void **)psrc
VBoost.Assign pDst, ByVal pSrc
'*(void**)pDst = *(void*)pSrc
VBoost.Assign ByVal pDst, ByVal pSrc

'Equivalent statements
pDst = pSrc
VBoost.Assign pDst, pSrc
VBoost.Assign ByVal VarPtr(pDst), ByVal varPtr(pSrc)

'More equivalent statements.This demonstrates an assignment
'across types. Note that Assign is faster because it doesn't
'perform an AddRef/Release on the Me object.
pDst = ObjPtr(Me)
VBoost.Assign pDst, Me

'Assign a 10-element array directly to an array variable.
'SafeArrayCreateVectorEx returns a Long value which is
'assigned direcly to the array variable's memory.
Dim x() As Long
VBoost.Assign ByVal VarPtrArray(x),_
  SafeArrayCReateVectorEx(vbLong, 0,10)
```

由于这个函数直接引用了内存，如果使用不当的话会很容易的导致崩溃。请记住经常保存所做的工作，第三个语法很常用，因此 VBoost 有特殊的函数来协助这个操作。

2. Deref(ByVal Ptr As Long) As Long

解除对 Ptr 内存的引用，返回值。这是 VarPtr 函数的反向功能实现。

```
Dim  lVal As long
Dim Ptr AS Long
Ptr=varptr(lval)
Debug.print VBoost.Deref(Ptr)=lVal'prints True
```

```
'use Deref in line to get the dimensions of an array
Dim CDims As Long
Dim X(0.0) As long
cDims=SafeArrayGetDim(VBoost.Deref(VarPtrArray(x)))
'cDims=2
```

3. AssignZero(pDst As Any)

AssignZero 用来清除指针，它在功能上等同于 Assign pDst, 0&，但是 AssignZero 生成的代码更少，因为 VB 不需要构造一个临时的变量来保存 0&常量（不需要记住 0 后面的&符号）。

4. AssignAddRef(pDst As Any, pSrc As Any)

AssignAddRef 是 Assign 的一个特殊版本，它的输入数据是一个对象。在函数返回前调用 AddRef 函数。因为 pDst 标识为输出参数，在调用 AssignAddRef 前不需要将 pDst 设成 Nothing。由于 As Any 类型，这个函数不会进行 QI，只进行 AddRef。我们可以从一个长整型值（对一个对象的指针）或者其他的对象赋值。

```
Dim Me1 As Class1
Dim Me2 As Class2
Dim pUnk As stdole.IUnknown
VBoost.AssignAddRef Me1, Me
VBoost.AssignAddRef Me2, ObjPtr(Me)
VBoost.AssignAddRef pUnk, Me

'This outputs four equivalent values.
Debug.print ObjPtr(Me), ObjPtr(Me1),ObjPtr(Me2), _
  ObjPtr(pUnk)
```

使用 UserControl 变量时，有一种使用 AssignAddRef 很方便，但是又不正式的用法。UserControl 在 VB 中称为"私有基类"，也就是说基类的成员在 Me 对象中不可用。这就是为什么一个空的"UserControl,Me"对象没有任何成员，但是在一个窗体 Me 对象中可以访问所有的窗体成员。VB 的编译器不允许将一个私有基类赋给一个变量或者传递给一个通常的参数。编译器在判断一个赋值语句是产生 AddRef 还是 QueryInterface 时做这个检查，当使用 ByRef As Any 传递参数时就会跳过这个检查，因此可以用 AssignAddRef 将 UserControl

的引用存到一个变量中。但是必须保证 UserControl 的引用是在它自身的生存期内。

```
Dim UC As UserControl
    'Failing Code
     Set UC = Usercontrol
      'suceeding code
      VBoost.AssignAddRef  UC, UserControl
```

5. AssignSwap(pLeft As Any,pRight As Any)

AssignSwap 可以用来在两个 4 字节长的变量之间交换值（Long, String, object 等），而输入的数据没有被释放或被修改。AssignSwap 不是一个通常的交换逻辑，如果传递两个 Variants，那么通过交换 Variants 的类型，数据就丢失了。如果传递两个整型值，结果将没有定义。如果用 AssignSwap 在两个相同类型的变量间传递内存值，而不是用 Assign，那么在调用之前就不用先把目标变量清空。

```
'Exchange two String values.
Dim str1 As String, str2 As String
Str1 = "ped"
Str2 = "Swap"
VBoost.AssignSwap str1, str2
Debug.Print str1; str2
'output: Swapped

'Swap an array variable into a function name
Private m_Cache() As Long
Public Function FillArray() As Long()
   'Call helpers to fill m_Cache.
   'Set return value with AssignSwap.
  VBoost.AssignSwap _
     ByVal VarPtrArray(FillArray), _
     ByVal VarPtrArray(m_Cache)
   'Equivalent code without AssignSwap. Note that AssignSwap
   'into an empty variable is faster than Assign/assignZero.
   Erase FillArray
```

```
    VBoost.Assign _
       ByVal VarPtrArray(FillArray), _
       ByVal VarPtrArray(m_Cache)
     VBoost Assignzero ByVal VarPtrArray(m_Cache)
End Function
```

6. MoveVariant(pDst As Variant, pSrc As Variant)

MoveVariant 允许把一个 Variant 赋给另外一个，而不管它的类型。通常在给可能有多种情况的 Variants 赋值时，需要用 Set 语句或者 IsObject 来进行分支处理。如果不用 Set 语句给一个对象类型的 Variant 赋值时，VB 会试图指定对象的缺省值而不是对象本身。MoveVariant 通过在参数中的[out]属性来清除 pDst Variant 中的数据，把所有 16 字节的数据从 pSrc 移到 pDst，然后将 pSrc 置为空。用这种方法不仅避免了检查数据类型，而且消除了数据的冗余。

```
Public Function GetData () As Variant
   VBoost.MoveVariant GetData, HelperFunc
End Function
```

如果不使用 MoveVariant，在每次调用 HelperFunc 时需要知道它的返回类型，甚至需要写一个带输出参数的 HelperFunc，而不是直接返回值。在 Variants 之间移动很大的数组或字符串时，MoveVariant 这个很酷的工具可以避免数据的不必要冗余。

7. SafeUnknown(ByVal pUnknown As Unknown) As stdole.IUnknown

在把一个对象赋给一个 IUnknown 变量的时候，VB 通常会产生一个 QueryInerface 的调用或者 IID_IUnknown。将这个赋值过程用 SafeUnknown 包装起来会避免 QueryInterface 调用，并且将当前的指针而不是控制的 IUnknown 进行赋值。

```
Dim pUnk1 As stdole.IUnknown
Dim pUnk2 As stdole.IUnknown
Set pUnk1 = Me
Set pUnk2 = VBoost, SafeUnknown(Me)

Debug.Print ObjPtr(pUnk1) = ObjPtr( pUnk2)  'False
```

8. [UAdd|UDif|UDiv](ByVal x As Long, ByVal y As Long)

这个函数用来将 x 和 y 的和（UAdd）、差（UDif）、乘积（UDiv）的结果用无符号的整型值返回。UAdd 和 UDif 函数主要用来进行安全的指针运算，虽然 Udiv 很少用，它是为了完整性而添加进来的。Uadd 和 Udif 没有进行溢出检查。没有提供乘法的函数是因为：任何乘法运算（除了用 1 和一个大于&H7FFFFFFF 的值相乘）都会导致溢出，所以这个函数没有用。

```
'Add 1 to the maxium positive signed long value.
Debug.Print Hex$(VBoost.UAdd(&H7FFFFFFF, 1)  '80000000
```

9. [UGT|UGTE](ByVal x As Long, ByVal y As Long) As Boolean

将 x 和 y 的值当作无符号的长整型用大于（UGT）或大于等于(UGTE)来比较，没有提供 ULT 和 ULTE，只要将参数的顺序颠倒即可。

内存分配对象

VBoost 提供了两种指定大小的内存管理器。基本原型 FixedSiceMemoryManager 直到对象本身释放时才会将内存释放。在内存管理器的生存期中可以将内存回收，但是不能将它们返回给系统。第二个内存管理器 CompactibleFixedSizeMemoryManager，在分配和释放内存时要比第一个要慢一些，但是在将资源交还给系统时不需要将对象本身销毁。在 VBoost 对象中有一个函数，CreateFixedSizeMemoryManager 可以创建这两种类型的对象。

CreateFixedSizeMemoryManager 接受三个参数，返回一个 FixedSizeMemoryManager 对象。如果指定了可压缩的内存，那么可以将返回的对象赋给 CompactibleFixedSizeMemoryManager 变量来得到压缩所需要的额外的属性和方法。首先来看一下创建这个对象时所需要的三个参数：

ElementSize(Long) 指定了每个分配单元中内存的字节数。由于性能的原因，这个值会被调整到能被 4 整除，所以从 ElementSize 属性读出的值可能比这里指定的值要大。

ElementsPerBlock(Long) 是在一个分配单元中先要保存的元素的数目，内存会从未分配的内存中一次分配 ElementSize*ElementsPerBlock 个字节。

FCompactible(Boolean) 表示是否支持压缩。缺省值是 False。如果这个值是 True 的话，返回的对象就同时支持 FixedSizeMemoryManager 和 CompactibleFixedSizeMemoryManager 两个接口。

下面的属性和方法是这两个接口都支持的：

Alloc，没有参数返回的内存指针。在内存分配时 Alloc 会使用一个大小参数，这里并不需要是因为这些对象中的内存块的大小是一致的。如果所有以前分配的内存都在使用，Alloc 会到未分配的堆栈中取得额外一个 ElementsPerBlock 对象所需的内存，这个函数也可能返回内存溢出的错误。

Free 将 Alloc 返回的指针作为参数，这个指针在 Alloc 后就可以使用。Free 不检查输入的指针也不返回错误，所以请确信输入的指针正确，否则会导致崩溃。

ElementSize 这个只读属性是 Alloc 返回的分配空间的大小。它可能和传递给 CreateFixedSizeMemoryManager 的 ElementSize 参数相同，但是通常它会被调整至 4 的倍数。例如要求 10 字节的请求返回的 ElementSize 是 12。保证能被 4 整除给处理器带来的好处完全值得消耗那些额外的内存。

ElementsPerBlock 这个只读属性，返回每个内存块中的项目数。

下列属性只应用于可压缩的对象：

CompactOnFree 这个属性可读写，它表明在调用 Free 函数时是否将内存返还给系统堆栈，还是等待一个显式的 Compact 调用。缺省值是 False。

BufferBlocks 这个可读写的属性表示在调用 Compact 函数时保存多少空的内存块。如果交替使用 Alloc 和 Free 函数，那么就会在内存块的边界停住。这时 Free 函数释放了一个内存块，下一个 Alloc 需要更多的内存然后重新分配了和刚刚释放的内存一样大小的内存块。显然，这种情况只在 CompactOnFree 是 True 时发生。为了避免这样的情况，我们可以指定在压缩时不将最后一个 BufferBlocks 返回给系统堆栈。改变 BufferBlocks 不会自动触发一次压缩，而必须通过显式的对 Compact 的调用来释放空的内存块。它的缺省值是 1。

Compact 在决定将内存还给系统堆栈时调用。真正释放的内存与 BufferBlocks 设置有关。如果 CompactOnFree 的值是 True，而且最近没有减少 BufferBlocks 的值，那么就不需要显式的调用 Compact。BufferBlocks=0 后调用 Compact 会将所有没有使用的内存返还给系统。

聚合函数

VTable 委派和 IUnknown 钩子是两个核心的技术，VBoost 用它们使得一个对象好像多个对象一样。尽管 VBoost 展现了 HookQI，HookQIAR 和 CreateDelegator 函数，这里仍然建议调用 AggregateUnknown 和 CreateAggregate,因为它们自动处理了基于指定数据的 QueryInterface 的交互和委派者的创建。仅仅用 HookQI 和 CreateDelegator 来聚合对象既笨重又没有太大用处。

我希望读者将大多数时间花在这两个聚合函数上，所以这里会先讨论这两个函数然后再回来看 HookQI，HookQIAR 和 CreateDelegator。每个 COM 对象都有一个控制 IUnknown 接口，而且用一个 IUnknown 接口就可以区分同一个对象的不同接口引用。聚合的过程实际上就是通过给两个或多个 COM 对象分配相同的控制 IUnknown 从而将多个 COM 对象聚

合起来。而最终的使用者则不会意识到最后的对象是由多个对象聚合起来的。

控制 IUnknown 在聚合中扮演着核心的角色，它用来定义 COM 对象的标识，所以聚合的第一步就是得到这个控制 IUnknown。为什么这里需要两个聚合函数是因为 AggregateUnknown 钩住了一个现存的 IUnknown，在其基础上实现聚合；而 CreateAggregate 虽然使用相同的数据，但是重新创建了一个控制 IUnknown 对象。这两个函数最重要的参数是 pData 和 pIIDs 数组。AggregateData 结构和对应的标志是 VBoost 聚合的核心。

1. AggregateData 结构

聚合函数利用 AggregateData 结构数组来决定最终对象所支持的对象和 IIDs。在 AggregateData 中的 FirstIID 和 LastIID 字段是 pIID 数组（用 pData 数组传递）从 0 开始的索引。PObject 字段保存将要放入控制 IUnknown 的对象。 Flags 字段是 ADFlags 类型，决定聚合器如何来翻译其他的三个字段。

adIgnoreIIDs(默认为 0)，表示没有使用 FirstIID 和 LastIID 字段。在运行时，VBoost 按照 IID 的列表来进行一个个处理，再根据指定的 adIgnoreIID 向每个对象发送 QueryInterface 的请求。首先成功响应的对象会封装到一个防风墙中以便使用，如果对象没有的 IID 请求指定过滤器，则称为“盲聚合”。

adUseIIDs 指定 FirstIID 和 LastIID 包含一组标识与在 pObject 中指定的对象对应。如果 LastIID 比 FirstIID 小就会被忽略。如果想要通过早些放入列表来保证 pObject 映射到指定的 IID，还可以在后面的 AggregateData 数组把相同的对象当作盲聚合包含进来。

adMapIID 指定用 FirstIID 索引的 IID 和 LastIID 指定的 IID 对应。IID 映射可以让我们通过一个 implements 语句来实现多个继承的接口。例如如果 IFooEx 从 IFoo 继承，那么对象就应该同时响应 IID_IFooFx 和 IFoo。通过在 pIIDs 数组中将 FirstIID 设为 IID_IFoo，并将 LastIID 设为 IID_FooFx，就可以避免用 Implements 造成的冲突。在使用 adMapIID 时，应将 pObject 设为 Nothing。

adBlockIIDs 指定一组不会被接受的 IID。这个需求很少，然而仍然支持。PObject 需要设为 Nothing，如果 LastIID 比 FirstIID 小的话会被忽略。

adPrimaryDispatch 在接受 IID_IDispatch QI 时为对象提供了第一机会。由于从 IDispatch 可以继承多个接口，但是一个 COM 对象只能返回其中一个接口的引用，所以这里需要用 QI 钩子来改变作为主 Dispatch 的接口（请参阅在第三章讨论的 IUnknown）。这个标志可以指定一个聚合对象而不是指定一个控制 IUnknown 作为主 Dispatch。AdPrimaryDispatch 的一个特殊的特性是如果不设置 pObject、adUseIIDs 和 FirstIID，那么聚合器会使用从控制 IUnknown 取回的 FirstIID 作为主 Dispatch。这允许我们将主 Dispatch 指定成从类中直接实现的接口。可以用 adPrimaryDispatch 和 adMapIIDs 来重定向 IID_IDispatch，或者在 IQIHook 接口的 QIHook 函数中更改。AdPrimaryDispatch 最简单是因为这些工作我已经做好了。（关于如何改变主接口，请参阅第 15 章中“Post-Build Type Library Modifications”一节。设置

主 IDispatch 是很关键的一步。）

adBeforeHookedUnknown 让聚合器在将 QI 调用发送给主对象之前先发送到 pObject。缺省情况下，被钩住的对象是 QI 链中的第一个，成功的话就不再处理。这个标志可以用来在 VB 接受这个调用前覆盖支持的接口。在使用 CreateAggregate 时，由于没有指定的控制 IUnknown，这个标志没有用处。

adDontQuery 指定对象可以引用但是不可以查询。这个标志大大简化了复杂对象的创建，因为这需要强引用。关于如何使用这个标志的例子，请参考第六章中的“Hierarchical Object Models”部分。

adNoDelegator 使得聚合器在返回的接口上不设置一个盲代理。当然，代理者提供了 QueryInterface 的覆盖来将调用返回控制 IUnknown，这就意味着丢失了 COM 标识。不生成代理有三个好处：首先没有代理的额外开销；其次，对象是一个纯粹的 VB 对象，不可以使用 Friend 函数；第三，由于和上面相同的原因，调试器将返回的对象当作 VB 对象，从而可以在实现显示私有变量等。如果对一个公共的 COM 对象做聚合的话，请遵守 COM 的公约，对所有公共的接口允许代理。然而对内部对象用 adNoDelegator 进行聚合可以提供对直接 Set 语句的支持。

adDelayCreation 告诉聚合器传送的对象是 IDelayCreation 接口。IDelayCreation 有一个 Create 函数，可以用 IID 参数返回一个对象。延迟创建可以用来按需创建对象，而不是在创建聚合器前创建对象。在使用延迟创建时也需要指定 adUserIID 和 FirstIID。如果 LastIID 也设置了，那么通过 IDelayCreation_Create 返回的对象可以支持所有指定的 IID。VBoost 支持在聚合控制 Unknown 对象时实现 IDelayCreation，这样就通过自动处理循环引用来避免对象的生存期问题。

```
'IFoo is delay created by AggObj.
Dim IFoo As IFoo
'Calls IDelayCreation_Create.
Set IFoo = AggObj
Set IFoo = Nothing
'IDelayCreation_Create is not called a second time
'because the reference is cached.
Set IFoo = AggObj
```

adDelayDontCacheResolved 会让聚合器不去缓存 IDelayCreation_Create 返回的对象。在使用 adDelayCreation 时，IDelayCreation 只用一次然后恢复为一个普通对象。聚合器保存了一个对延迟创建对象的引用，直至这个对象被销毁。如果这个接口很少使用，或者在一段时间内保证接口的正确性有问题，或者返回的实现中包含一个对控制对象的引用。也许 adDelayDontCacheResolved 标识是一个更好的选择。AdDelayDontCacheResolved 体现了

adDelayCreation，因此不需要同时指定这些标识。同样的 adDelayDontCacheResolved 会让聚合器不保持未解析的引用，所以即使正在使用这个 IID 的引用，对同一个 IID 的多次请求也会引起多次 IDelayCreation_Create 调用。通常在聚合需要合并的对象时不使用这个标志，因为 COM 的合并器会首先缓存接口，外界只会看到请求。

adFullyResolved 让聚合器在返回一项后不去调用 QueryInterface。如果 adUseIID 没有设置，并且只指定了一个 IID 的话，将会忽略 adFullyResolved。在封装一个指定对象的接口时，不会 QI 这个对象来得到封装后的接口，因为已经有未封装的接口了。通过让聚合器跳过 QI 可以对封装后的对象进行聚合。可以使用这个标志作为对其他接口的优化，但是在设置 pObject 字段时需要十分小心。通常会用 VBoost.AssignAddRef 来避免 QueryInterface 调用。从 IDelayCreation_Create 返回的对象也会服从这个标志。

AdWeakRefBalanced 告诉聚合器添加的对象已经有对控制 IUnknown 的引用了。在这种循环调用的情况下，UnknownHook 对象本身就有对父类的引用，因此不能正确释放。为了打破这种循环引用，避免在聚合对象本身包含不安全的引用，需要在控制 IUnknown 释放一个引用，在释放聚合对象前减小引用数。将 adNoDelegator 和 adWeakRefBalancedj 结合起来很危险，因为返回的引用在控制 IUnknown 中没有引用，很容易崩溃。

adWeakRefRaw 让聚合器将引用当作基本的弱引用，控制 IUnknown 应该有对弱引用的引用。关于 adNoDelegatorde 的注意事项同时适用于 efRaw 和 adWeakRefBalanced。

2. AggregateUnknown

AggregateUnknown 函数在现存 IUnknown 基础上通过钩住控制 IUnknown 的 VTable 来进行聚合。钩子一直在起作用，直到 AggregateUnknown 返回的 UnknownHook 对象被设为 Nothing。通常 UnknownHook 对象本身被控制的 IUnknown 所有，即使钩子正在运行也可以正确拆卸，这样就可以只用 Class_Initialize 就可以实现聚合。下面是 AggregateUnknown 的参数。

pUnk(ByVal stdole.IUnknown)作为控制 IUnknown 的对象。在传递这个参数时不要使用 SafeUnknown，这里需要使用 IUnknown 返回的 IUnknown QI。通常会把 Me 传递给这个参数。

PData(AggregateData array)是 AggregateData 项的数组，用来描述聚合的输出。

PIIDs(VBGUID array)是用 FirstII 和 DLastIID 指向 pData 的 IIDs 的列表。这个数组可能是为空，但是必须提供这个参数。

ppOwner(UnknownHook, out parameter)使用一个声明为 UnknownHook 的成员变量，让它来使用钩子。在 UnknownHook 有效时钩子将一直存在。可以在任何时候通过将这个变量设为 Nothing 来取消钩子。

```
'Aggregate a Form with a collection object, then retrieve
```

```
'the collection from the form and display and item.
Private m_Hook As UnknownHook
Private Sub Command1_Click()
Dim Coll As Collection
    Set Coll = Me
       MsgBox Coll(1)
    If TypeOf Coll Is Form Then
       MsgBox "(With a new identity)"
    End If
End Sub

Private Sub Form_Initialize()
Dim Coll As Collection
Dim pIIDs() As VBGUID
Dim AggData(0) As AggregateData
    Set Coll = New Collection
    Coll.Add "I'm alive!"
    With AggData(0)
       Set .pObject = Coll
       .Flags = adIgnoreIIDs
    End With
    VBoost.AggregateUnkown Me, AggData, pIIDs, m_Hook
End Sub
```

3. CreateAggregate

CreateAggregate 使用和 AggregateUnknown 一样的数据来进行聚合，但是它不钩住一个已存在的控制 IUnknown 对象，而是创建了一个全新的控制 IUnknown 对象。CreateAggregate 返回一个 stdole.IUnknown 的引用，可以将它赋给一个更确定的对象类型。

pData 与 AggregateUnknown 相同。

pIIDs 与 AggregateUnknown 相同。

pOwner(Optional,ByVal Long)是一个可选的参数用来跟踪聚合的对象是否依然存活。指定 pOwner 可以为每个请求返回一个现存的聚合而不是重新创建一个。如果想要这样的功能的话，可以将 VarPtr 作为一个对象的成员变量来传递，这些对象是在 pData 数组的 pObject 字段中指定的。当聚合对象销毁时，内存空间被释放。由于聚合时保存了对传递 pData 的 pObject 对象的引用，所以在聚合时可以保证内存地址的正确性。

```
'CreateExternal is a specific version of the function shown in
'in "Hierarchical Object Models" section of Chapter 6. In
'this scenario, a Child object has a weak reference on its
'parent object. When the Child object is passed to an
'external client, it needs to protect its parent. This is
'easily done with CreateAggretgate by establishing the child
'as a blindly aggregated object and the parent as a reference
'that isn't queried.
'Child is the main object, ParentPtr is the ObjPtr of its
'parent, and ExternalPtr is a member vaiable of child. This
'function is in ExternalChild.Bas on the CD.
Function CreateExternal(ByVal Child As IUnknown, _
   Byval ParentPtr AS Long, ExternalPtr As Long) As IUnknown
Dim AggData(1) As AggregateData
Dim IIDs() As VBGUID
   If ExternalPtr Then
      VBoost.AssignAddRef CreateExternal, ExternalPtr
   Else
      'Defer all QueryInterface calls to the child.
      With AggData(0)
         Set .pObject = Child
         .Flags = adIgnoreIIDs
      End With

      'Let the aggregator hold the parent reference for us,
      'but don't forward QueryInterface calls to it.
      With AggData(1)
         VBoost.AssignAddRef  .pObject, ParentPtr
         .Flags = adDontQuery
      End with

      Set CreateExternal = VBoost.CreateAggregate( _
         AggData, IIDs, VarPtr(ExternalPtr))
   End If
End Function
```

4. HookQI 函数，IQIHook 接口

HookQI 可以用来监测和修改 QueryInterface 调用而不是使用提供的聚合器来实现。对接口请求的监测是通过 IQIHook 接口来实现的。在想要监测对对象的请求是否成功时，HookQI 非常有用。在调试时 QI 监测很有帮助，如果一个失败的 QI 请求并且导致了错误的话，通常表明实现了其他的接口。可以通过在注册表中 HKEY_CLASSES_ROOT\Interfaces 部分查找 IID 来区分大多数接口。

（1）下面是 HookQI 的参数：

pUnk(ByVal stdole.IUnknown)是控制 IUnknown 被钩住的对象。在传递这个参数时不要使用 SafeUnknown，这里需要使用 IUnknown 返回的 IUnknown QI。通常会把 Me 传递给这个参数。

pQIHook(ByVal IQIHook)是对一个 IQIHook 实现的引用。PQIHook 可以和 pUnk 有相同的控制 IUnknown，也就是可以钩自己。

uhFlags(ByVal UnkHookFlags)是一个集合，这个集合可以包括空的、一些或者全部 uhBeforeQI，uhAfterQ 和 uhMapIIDs 的值。设置这些值表示什么时候调用那些 IQIHook 函数。这个参数提供了初始的 Flag 设置，可以在任何时候通过在返回的 UnknownHook 对象的 Flags 属性来修改。注意，Flags 属性是 UnknownHook 的惟一属性，在使用 AggregateUnknown 而不是使用 HookQI 和 HookQIAR 创建时是只读的。

ppOwner(UnknownHook, out parameter)在返回后这个参数保存拥有钩子的对象。在 UnknownHook 有效时钩子将一直存在，除非超过了有效范围或者将它设为了 Nothing。这里传递的变量通常是被钩住的 Unknown 的一个成员变量。

IQIHook 接口有两个函数：MapIID 和 QIHook。对这些函数的调用次数依赖于 UnknownHook.Flags 属性的当前设置。如果指定了 uhMapIIDs 那么在 QueryInterface 的早期阶段就会触发 MapIID 函数，这时可以交换不同的 IID 或者阻断这次调用。如果设置了 unBeforeQI，在 QueryInterface 试图访问控制 IUnknown 前会调用 QIHook 函数，这样就可以对特定的 IID 进行抢占处理。如果设定了 uhAfterQI，将会在 QueryInterface 调用后调用 QIHook，允许校验或替换返回的对象。在上述函数调用中。IUnknown 钩子没有激活，被钩住的 IUnknown 暂时和本来的状态一样。

MapIID 使用一个 IID 作为 VBGUID 参数。可以在 QueryInterface 调用中改变这个 IID 或者通过产生一个错误来阻断整个函数调用。虽然要有阻断一个 IID 的需求，但是改变 IID 的映射常见于下面两种情况。一种情况是可以将 IID_IDispatch IID 映射为一个实现了的接口而不是控制 IUnknown 上的主接口。第二种情况是如果一个接口对多个 IID 的值响应，最简单的方法就是通过 IID 映射来支持所有的 IID。（在前面的 aMapIID 标志讨论中已经提到这个问题）

（2）**QIHook** 可以用来观察和修改从控制 IUnknown 的 QueryInterface 中返回的引用。它有四个参数：

IID 是被访问的 IID。如果 MapIID 修改这个 IID，那么这就是修改后的版本。在调用中不应该修改 IID 的值。

uhFlags(ByVal UnkHookFlags) 是 uhBeforeQI 或 uhAfterQI，表明什么时候调用了 QIHook。

PResult(ByRef stdole.IUnknown)是 QI 函数的返回值。对于一个 uhBeforeQI 调用，pResult 会以一个空值开始。如果在返回前设置它，那么原来对象的 QI 函数就不会调用。PResult 可以在 uhAfterQI 调用开始时设置，也可以任意修改。如果在 uhAfterQI 调用后的 pResult 是空的话，QueryInterface 是失败的，并且返回了 E_NOINTERFACE。设置 pResult 的值时，请确信是用 SafeUnknown 或者 AssignAddRef 赋值。不然的话，对 IUnknown 变量赋值会导致 QueryInterface 返回一个错误的接口，从而导致崩溃。CreateDelegator 函数常用来创建结果对象，返回一个 stdole.IUnknown 可以用 Set 语句直接赋给 pResult。

HookedUnknown(ByVal stdole.IUnknown)是被钩住的对象的控制 IUnknown。钩子在调用时并不存在，HookUnknown 使得创建控制 IUnknown 的 VTable 代理者更加方便。

```
'Watch all QueryInterface calls coming into this object and
'whether they succeeded.
Implements IQIHook
Private m_Hook As UnknownHook
Private Sub Class_Initialize()
    VBoost.HookQI Me, Me, uhAfterQI, m_Hook
End Sub
Private Sub IQIHook_MapIID(IID As VBoostTypes, VBGUID)
    'Not called
End Sub

Private Sub IQIHook_QIHook(IID As VBoostTypes.VBGUID, _
    ByVal uhFlags As VBoostTypes.UnkHookFlags, _
    pResult As stdole.IUnknown, _
    ByVal HookedUnknown As stdole.IUknown)
Dim GuidString As String * 38
    StringFromGUID2 IID,GuidString;
    Debug.Print "QI for"& GuidString
    If pResult Is Nothing Then
        Debug.Print "failed."
    Else
        Debug.Print "succeeded."
    End If
End Sub
```

5. HookQIAR 函数，IQIARHook 接口

HookQIAR 是 HookQI 的扩展。AR 表示除了 QueryInterface 以外，AddRef 和 Release 函数也会发出通知。在真正生成代码时 AddRef 和 Release 函数没有多大帮助，但是在开发过程中非常有用。如果能够在 AddRef 和 Release 调用发生时观察它们，那么就可以跟踪对象是否被引用和释放。

这里是 HookQIAR 的参数。

（1）**pUnk** 和 HookQI 相同。

pQIARHook(ByVal IQIARHook)是对一个 IQIARHook 实现的引用。IQIARHook 将 AfterAddRef 和 AfterRelease 函数添加到 IQIHook 接口中。作为 IQIHook，可以将实现 IQIARHook 作为观察的对象的一部分。但是采用这种方法时不会在最后释放时得到通知。在这种情况下实际上最后的释放也没有必要，因为 Class_Terminate 是和最后的释放函数等同的通知。

UhFlags 支持最多五个不同的值。除了 IQIHook 接口支持的三个标志外，IQIARHook 增加了 uhAddRef 和 uhRelease。和 HookQI 一样，可以对返回的 UnknownHook 对象使用 Flag 属性在对象的生存期内来动态改变当前的标志。但是不能通过改变 flag 值来在 QIHook 和 QIARHook 之间切换。用 HookQI 建立的 UnknownHook 对象会忽略这些标志。

ppOwner 和 HookQI 相同。

（2）IQIARHook 接口对 QIHook 和 MapIID 函数添加了 AfterAddRef 和 AfterRelease 通知。它们是严格的通知函数：不能用它们来修改对象的行为，这与 QI 的函数不一样。IUnknown 钩子在调用 IQIARHook 接口时失去焦点。

AfterAddRef 使用两个 ByVal 的 Long 型参数，Result 和 pHookedUnknown。Result 是对被钩住的对象 AddRef 调用的返回值。pHookedUnknown 与传递给 QIHook 函数的 HookedUnknown 参数中的 ObjPtr 类似。这些值可以用来在调试时跟踪。

AfterRelease 接受和 AddAddRef 相同的参数。当然，Result 的值是从 Release 返回而不是从 AddRef 返回。如果得到的 Result 是 0，那么 pHookedUnknown 就是刚刚销毁的对象的指针。为了真正得到 0 这个结果，需要在除了应用钩子的对象外其他的对象上应用 IQIARHook。

除了使用 HookQIAR 来跟踪实例，它在观察 VB 和 IUnknown 接口的交互时也很有帮助。例如，在单步执行完 HookQIAR 函数时可以得到两次 Release 调用。这些调用是由一些临时变量产生的，这些变量用来按照 HookQIAR 的要求将 Me 传递给 IUnknown 和 IQIARHook 接口。在一个类中单步执行公共或者友函数时会观察到 AddRef/Release 的顺序，但是对一个私有函数则观察不到。

```
'Output all object creation and destruction as well as all
'QueryInterface, AddRef, and Release calls.
Implements IQIARHook
```

```
Private m_Hook As UnknownHook
Private Sub Classs_Initialize()
Dim pUnk As IUnknown
    Set pUnk = Me
    Debug.Print "Initialize: " & Hex$(ObjPtr(pUnk))
    Set pUnk = Nothing
    VBoost.HookQIAR -
        Me, Me, uhAfterQI or uhAddRef or uhRelease, m_Hook
End Sub
Private Sub Class_terminate()
Dim pUnk As IUnknown
    Set m_Hook = Nothing
    Set pUnk = Me
    Debug.Print "Terminate:" &Hex$(ObjPtr(pUnk))
End Sub
Private Sub IQIARHook_AfterAddRef( _
    ByVal Result As Long, ByVal pHookedUnknown As Long)
    Debug.Print "AddRef="; Result; "On: ";_
        Hex$(pHookedUnknown)
End Sub
Private Sub IQIARHook_AfterRelease( _
    ByVal Result As Long, ByVal pHookedUnknown As Long)
    Debug.Print "Release="; Result; "On: ";_
        Hex$(pHookedUnknown)
End Sub
Private Sub IQIARHook_MapIID(IID As VBoostTypes.VBGUID)
    'Not called
End Sub
Private Sub IQIARHook_QIHook(IID As VBoostTypes.VBGUID), _
    ByVal uhFlags As VBoostTypes.UnkHookFlags,_
    pResult As stdole.IUknown,_
    ByVal HookedUnknown As stdole.IUnknown)
        If Not pResult Is Nothing Then
        Debug.Print "QI Successful on: ";
        Debug.Print Hex$(ObjPtr(HookedUnknown))
    End If
End Sub
```

6. CreateDelegator

CreateDelegator 函数用来在一个内部对象和控制 IUnknown 之间创建一个盲 VTable 代理者。可以在 IQIHook_QIHook 中用这个函数来聚合函数。它需要两个参数，三个在使用高级代理者时的可选参数

punkOuter(ByVal stdole.IUnknown) 是盲代理者使用的是在盲代理者中 QueryInterface 调用所遵从的控制 Unknown。

punkDelegatet(ByVal Unknown) 是要封装的接口。注意这个参数的接口类型是 Unknown，因此 VB 在将对象赋给这对参数时不会调用 QueryInterface 函数。

pIID(Optional,ByVal VBGUIDPtr) 是一个可选的参数，它指定了一个 IID 用来查询被代理者，并决定需要封装的正确接口。如果 punkDelegatee 是对需要封装对象的一个引用而不是正确的接口，将 VarPtr(IID_Wanted)传递给 pIID 参数。

pEmbedded(Optional, ByVal BlindDelegatorPtr) 允许 VTable 代理者作为一个大结构的一部分来分配。为了使用这个参数，在结构中包含一个 BlindDelegator 类型的成员并传递 VarPtr(.BDMember)。如果传递 pEmbedded 的话，还可以在 pfnDestroyEmbedded 中指定一个函数地址。

PfnDestroyEmbedded(Optional,ByVal Long) 是在 BlindDelegator 销毁时调用的函数。如果没有指定 pEmbedded，在分配一个 BlindDelegator 结构时会使用标准的分配方法。当代理者的引用释放时这个结构被删除。析构函数保存在 BlinkDelegator 结构的 pfnDestroy 字段中，能够在对象的生存期中修改。例如，AggregateUnknown 聚合器可以比钩子生存期更长。当钩子发现有额外的对象时，它会为额外的 IID 表中的项分配对 FixedSizeMemoryManager 的引用，用来为他们分配内存，修改在 BlindDelegator 结构中的 pfnDestroy 字段来在对象释放时释放内存。尽管这是一个高级特性，为了在 VB 中可以更改 VBoost 对象，以及为了定制生成 BlindDelegator 都需要这个特性。例子请参阅第 12 章中的“接口封装”和“封装我们自己的接口”部分。

7. BlindFunctionPointer

BlindFunctionPointer 函数可以对一个轻量的对象使用 VTable 代理来覆盖现有接口的函数。例如在使用 CreateWindowEx 来创建非常定制化的控件时，需要覆盖 IoleInPlaceActive Object.TranslateAccelerator 函数。这仅是一个相关函数，然而这需要为在 VTable 中的其他所有函数提供代理代码。BlindFunctionPointer 可以用来使用 VTable 的代理代码，这样就不需要为每个函数编写实现代码了。为了使用 BlindFunctionPointer，内部的对象需要放在结构中 VTable 指针的后面。

```
Public Type IPAOHookStruct
    lpVTable As Long 'VTable pointer
```

```
    lpAOReal As IOleInPlaceActiveObject 'Inner object, offset
    'four
    'Other fields
End Type

Private Type IPAOVTable
    VTable(9) As Long
End Type
Private m_IPAOVTable As IPAOVTable

'VTable layout without BlindFunctionPointer
With m_IPAOVTable
    .VTable(0) =FuncAddr(Addressof  QueryInterface)
    .VTable(1) =FuncAddr(Addressof  AddRef)
    .VTable(2) =FuncAddr(Addressof  Release)
    .VTable(3) =FuncAddr(Addressof  GetWindow)
    .VTable(4) =FuncAddr(Addressof  ContextSensitiveHelp)
    .VTable(5) =FuncAddr(Addressof  TranslateAccelerator)
    .VTable(6) =FuncAddr(Addressof  OnFrameWindowActivate)
    .VTable(7) =FuncAddr(Addressof  OnDocWindowActivate)
    .VTable(8) =FuncAddr(Addressof  ResizeBorder)
    .VTable(9) =FuncAddr(Addressof  EnableModeless)
End With

'VTable layout without BlindFunctionPointer
With m_IPAOVTable
    .VTable(0) =FuncAddr(Addressof  QueryInterface)
    .VTable(1) =FuncAddr(Addressof  AddRef)
    .VTable(2) =FuncAddr(Addressof  Release)
    .VTable(3) =VBoost.BlindfunctionPointer(3)
    .VTable(4)=VBoost.BlindfunctionPointer(4)
    .VTable(5) =FuncAddr(Addressof  TranslateAccelerator)
    .VTable(6) =VBoost.BlindfunctionPointer(6)
    .VTable(7) =VBoost.BlindfunctionPointer(7)
    .VTable(8) =VBoost.BlindfunctionPointer(8)
    .VTable(9) =VBoost.BlindfunctionPointer(9)
End With
```

注意在代理 VTable 中的前三项和 BlindDelegator 结构的成员直接交互。由于这个原因，需要像在第 12 章中“更瘦的接口封装”那样，提供自己的 IUnknown VTable 函数或者直接使用 BlindDelegator 结构。可以向 BlindFunctionPointer 传递一个特殊的值-1，来得到对整个 VTable 的指针而不仅仅是表中的一项。可以使用 CopyMemory 中的值，用一个调用来复制大量的 VTable 项。例如上面的代码可以修改为如下所示。

```
'VTable layout with BlindFunctionpointer(-1)
With m_IPAOVTable
   CopyMemory .VTable(0), _
      ByVal VBoost.BlindFunctionPointer(-1), _
      LenB(.VTalbe(0)) * (UBound(.VTable) + 1)
   .VTable(0) =FuncAddr(Addressof  QueryInterface)
   . VTable(1) =FuncAddr(Addressof  AddRef)
   .VTable(2) =FuncAddr(Addressof  Release)
   .VTable(5) =FuncAddr(Addressof  TranslateAccelerator)
End With
```