真是想不到系列文章(1-6) - VB6指针技术大揭秘

目录

1

每次看大师的东西到了精彩之处，我就会拍案叫绝：\哇噻，真是想不到！\。在经过很多次这种感慨之后，我发现只要我们动了脑筋，我们自己也能有让别人想不到的东西。于是想到要把这些想不到的东拿出来和大家一起分享，希望抛砖引玉，能引出更多让人想不到的东西。

真是想不到系列之一：VB到底为我们做了什么？

关键字：VB、底层、WIN32、API、COM

难度：中级

要求：熟悉VB，会用VC调试器，了解WIN32 SDK、COM。

VB一直以来被认为有以下优缺点：优点是上手快、开发效率高；缺点是能力有限，运行效率低。这正是有些软件把VB做为首选语言，而有些软件肯定不会用VB做的原因。而很多VC，DELPHI的程序员都认为VB里搞开发不自由，它让我们做事变容易的同时，也让我们发挥的余地越来越小。的确，简单和功能强大这两者本身就是一对矛盾。那怕一行代码不写，仅仅起动运行一个空窗体这样简单动作，VB在底下就为我们做了大量复杂的工作（决不仅仅是注册窗口类、显示窗口、起动消息循环这么简单），这些工作对程序员是透明的。我们在感谢VB开发小组对我们程序员体贴入微的同时，不禁也要责怪为什么在文档中对这些底层的动作只字未提，虽然这些动作对最终的程序也许并无影响，但我们拥有知情权，更何况这些动作有时的确会影响我们的工作（我将在本系列后面的《VB多线程》中谈到这种影响）。

然而，所有希望从本文得到\未公开技术秘密\的朋友你将会很失望，因为我能够知道的和你一样多，我们所能做的一切就是站在外面来猜VB在里面做了什么？所以我决不是要带大家一起去将VB反向工程，而是想通过猜想VB的内部工作来将一些原来比较模糊的概念搞清楚。作为一个系列的第一篇文章，它的目的是为了后面的深入打下基础，所以我会在需要的时候指出我们必须掌握的知识点，如果你不清楚，请及时地学习相关书籍来补课，具体见《参考书目》。

最后，要声明我在本文中所做的各种实验和推断仅是我个人的观点，不能保证其正确性，并且不承担任何相关的法律责任。

好，开始吧！首先准备好我们的武器，我下面要使用的工具主要有：VB6中文企业版+SP5（废话），还有SPY++、Dependency Walk和OLE Viewer（以下简称SPY和DEPEND和OLEVIEW，SPY在VB光盘的common\\tools\\vb\\下的SPY目录中，OLEVIEW是其下OLETOOLS目录中的OLEVIEW.EXE，注意其下还有一个OLE2VW32.EXE功能类似，不过本文所指的是OLEVIEW.EXE，还Denpend在其下的Unsupprt\\DEPEND里）。还要用用

2

VC（上面提的工具在VC里有），因为我们还要看看VB生成的代码，搞VB高级开发的朋友一定要会用VC调试器，懂点汇编更好。当然，本文的重点不在这儿，所以没有VC也不要紧。

打开VB6新建一标准EXE工程，在\工程\引用\对话框里应该已有四个引用，简单点就是：1、Visual Basic For Application(VBA) 2、VB运行时对象库 3、VB对象库 4、OLE自动化。前面三个是任何VB工程都必须的，你想不要都不行，不信你试着去掉对它们的引用。那么这三个核心类型库各有什么用，在最终生成的可执行程序中扮演怎样的角色，这是本文要分析的第一个问题。

1）VB、VBA、VBS的区别你搞清楚了吗？

首先VBS不应该和VB、VBA放在一起比较，它是微软按照自己定义的ActiveX Scripting规范完全从头开始写成的脚本语言，虽然它的语法结构和VB非常相似，但VBS仅仅依靠自动化对象来扩充其功能（只有后期绑定），它不能用implements来实现接口，不可能在VBS里直接使用API，没有VarPtr这样能得到指针的函数，而VBS缺少的这些功能正是VB和VBA所特有的。当然，这不是说VBS不如VB或VBA，Windows已经为VBS提供了足够强大的功能，我们可以用VBS来做脚本COM组件，而且借自动化对象的能力VBS可以说能力无限，所以有病毒用VBS来写，对程序员来说VBS最重要的功能莫过于可以给自己的软件提供宏功能，就象VC中提供的VBS宏功能那样。注意，VBS是Free的，这和在Office中使用VBA来提供宏功能不同，要集成VBA需要价格不低的许可证费用，关于脚本语言可参见MSDN中Platform SDK\\Tools and Languages\\Scripting。（在本系列后面的文章《脚本功能》中我会实做一个用VBS来提供宏功能的小软件）

那么VB和VBA又有什么不同呢？好吧，眼见为实，开始我们的实验吧！

如果装了Office 2000以上版本，那么打开OLEVIEW，点击File下的View TypeLib查看位于E:\\Program Files\\Common Files\\Microsoft Shared\\VBA\\VBA6下的VBE6.dll的类型库，再用同样的方法看看MSVBVM60.dll的类型库，你会发现它们的类型库基本上一模一样，除了VBE6多了一个VBEGlobal接口和实现这个接口的Global对象，这个Global对象我们也可以在VBA编程环境（比如用WORD的VB编辑器）中用对象浏览器看到。它有二个方法Load和UnLoad，还有一个UserForms属性，这是因为VBA6使用MS Form 2.0 Form设计器(FM20.dll)来设计和使用UserForm窗体（而在VB6中，我们可以使用多个设计器。比如通过使用MS Form 2.0 Form设计器，我们就能在VB中使用VBA所使用的UserForm用户窗体）。和VBA的Global对象类似，在VB中也有GLobal对象，从VB的对象浏览器中可以知道它在vb6.olb这个类型库中，这个类型库就是每个工程都必须引用的VB对象库，所有的VB内置对象都在这里。而VBA的UserForm中使用的对象都在FM20.dll中。

除了上述不同外，VB和VBA还有一个最大的不同，就是VBA不能生成EXE可执行文件，但可以猜想在IDE环境中VBA和VB都要把代码编译成p-code来执行，后面我将用实验来证明的确是这样，虽然在具体的实现上VB和VBA有很大的不同。

从上面的分析上可以看到VB和VBA还是有很大不同的，这种不同主要体现在编程环境和对象结构上，但在本质上它们之间却有着不可割舍的血源关系。如果刚才你仔细地观察

3

了MSVBVM60.dll的类型库，你就会发现如下的片断：

```
// Generated .IDL file (by the OLE/COM Object Viewer)

[

dllname(\
```

```
uuid(35BFBDA0-2BCC-1069-82D5-00DD010EDFAA),
helpcontext(0x000f6ec4)
]
module Strings {
[entry(0x60000000), helpcontext(0x000f665f)]
short _stdcall Asc([in] BSTR String);
[entry(0x60000001), helpcontext(0x000f6e9f)]
BSTR _stdcall _B_str_Chr([in] long CharCode);
.....
}
```

什么？在MSVBVM60.dll中的对象其方法却定义在VBA6.DLL中？！VB安装目录下不就有个VBA6.DLL吗？再用OLEVIEW看看它，哇噻，真是想不到它居然和MSVBVM60.DLL的一模一样。怎么回事？赶快再拿出DEPEND来看看VBA6.dll、MSVBVM60.dll和VBE6.dll这三个DLL的输出函数。哈，又有新发现，我们可以发现在三个DLL的输出函数中从编号512到717绝大部分都是一模一样的一些以rtc开头的函数，比如595的rtcMsgBox（rtc是什么？应该是Run Time Component? Control? Code?有谁知道吗？），这说明三个DLL都有着相同的运行时VBA函数。

我们再用DEPEND来观察一下VB6.EXE, 我们可以发现VB6.EXE引入了VBA6.DLL中一些它特有的以Eb和Tip开头的函数，从这些函数的名称上可以发现它们的功能都是IDE相关的，比如79的EbShowCode和82的TipDeleteModule。VB6.EXE恰恰没有引入任何rtc开头的函数（注意一）。我们再来看看MSVBVM60.DLL，随便找一个用了MsgBox函数的编译后的文件，用DEPEND来观察它，就会发现它引入MSVBVM60.DLL输出的595号rtcMsgBox函数（注意二）。并且引入MSVBVM60.DLL中很多以下划线开头的函数，比如__vbaVarAbs（注意三）。其实从这个三个\注意\中我们已经可以进行一些猜想，无论对错，你可以先想想。

4

如果你没有跟着我做实验，而仅仅是看这篇文章的话，我猜想你应该有点昏了。如果你自己动手做了这些实验，现在你应该充满了疑问而急侍看到结论。所以请一定要亲手试一试，学习研究问题的方法比看结论更重要。

到这里至少我们可以得出结论：VB和VBA本就是同宗的姐妹，只不过姐姐VB的功夫要比妹妹VBA历害些。不过姐姐只会单打独斗是女强人；妹妹却只会傍大款。姐姐有生育能力，是真正的女人；妹妹却不会生崽，但深谱相夫之道，一番教导指挥之下可使她老公增色不少，而VBS呢，也是大户人家的女儿，不过没有VB和VBA姐妹优秀的血统，娇小玲珑干不得粗活只能指挥些自动听话的对象来干活，她乐于助人品德好不象VBA那样只认大款，VB、VBA、vbs三个女人我都喜欢。

2)Native Code(本地代码)到底做了什么?

打起精神，我们再深入一步。用OLEVIEW得到的类型库还不能正确的反映各对象方法对应的DLL中的函数入口，你应该已经发现用OLEVIEW得到的IDL文件中各个方法的entry属性值都是0x600000XX这样的假东西。要得到类型库中各方法在DLL中的真正入口，我们需要自己来写段程序。

即使在VB中我们也可以非常容易地获取类型库信息，再加上点COM初始化和调用代码，我们就能用自己的代码实现VB6才引入的CallByName函数（在本系列后面的《Hack COM》中我会更深入谈谈COM，作为一名VB程序员对COM的理解非常重要）。由于本文的关键不是指导如何在VB里使用类型库，所以下面提供的方法尽量从简。

新建一个标准EXE工程，添加对TypeLib Infomation的引用，在Form中放一个名为lblInfo的标签，然后添加如下代码:

```
'程序1
Private Sub Form_Load()
Dim oTLInfo As TypeLibInfo
Dim oMemInfo As MemberInfo
Dim sDllName As String
Dim sOrdinal As Integer
Set oTLInfo = TLI.TypeLibInfoFromFile(\
5
lblInfo = \模块包含以下方法：\
For Each oMemInfo In oTLInfo.TypeInfos.NamedItem(\
With oMemInfo
.GetDllEntry sDllName, vbNullString, sOrdinal
lblInfo = lblInfo & .Name _
& \定义在\中，\
& \其编号为\
& vbCrLf
End With
Next
End Sub
```

运行以后我们就可以知道MATH模块中的Abs方法定义在VBA6.DLL中，其编号为656。在DEPEND中查看VBA6.DLL中编号为656的函数，果然就是rtcAbsVar，用VBE6.DLL试试结果相同。

还记得前面的注意一吧，VB6.EXE没有引入rtc开头的函数这说明在IDE环境中执行的VBA方法实际上是通过COM调用VBA对象库中的方法（跟踪p-code是噩梦，所以我无法验证它用的是什么绑定方式）。而注意二中提到的最终可执行程序中引入了rtcMsgBox，如我们所料最终的程序会直接调用它，这要比COM调用快一点，但在跟踪最终程序时，我发现rtcMsgBox内部却是经过了二万五千里长征后才会去调用MessageBoxA这个API，其间有多次对其它对象的COM调用，慢！可能是因为显示的是模态对话框，在多进程多线程环境有很多需要考虑的因素吧，如果你是疯狂在意效率的程序员，你应该试试用API来重写MsgBox，绝对快不少。再来看看注意三，让我们把以下的程序编译成使用本地代码的\程序2.EXE\（为了后面的实验，可以在工程属性的编译选项卡中将它设成\无优化\和\生成

6

符号化调试信息\程序2.EXE\）：

```
'程序2
Private Declare Sub DebugBreak Lib \
Private Sub Main()
Dim i As Long, j As Long
Dim k
i = &H1234
DebugBreak
```

```
k = 1234
j = Abs(k)
j = Abs(i)
MsgBox \
j = VarPtr(i)
End Sub
```

用DEPEND观察\程序2.EXE\，我们可以发现\程序2.EXE\并没有如我们预期的一样在引入595的rtcMsgBox的同时引入656的rtcAbsVar，相反它引入了__vbaVarAbs和__vbaI4Abs，看看函数名就知道一个针对的是Variant，一个针对的是long。这说明VB在最终生成的代码中对象Abs这样的可以进一步针对不同类型优化的VBA函数进行了相应的处理，观察一下所有以__vba开头的函数绝大部分都是那些最基本最常用的VBA函数，可以说__vba开头的VBA函数是rtc开头的VBA函数的优化版本，它们基本上是VB开发小组重新写的，绝大多数在函数内部实现自身功能，而rtc开头的函数大多数是调用COM服务对象来完成工作。从这么多__vba开头的函数上可以看出VB小组在Native Code（本地代码）的优化上下了不少功夫，这决对不是吹牛。它的确高度优化了不少科学计算相关的函数，以ABS为例Native Code要比p-code快4倍以上。但是并不是所有的计算函数都经过了这样的优化，比如Rnd函数，它就没有对应的__vba开头的优化函数，而是直接对应到rtcRandomNext函数上，虽然rtcRandomNext也已经优化过，但内部依然用了COM调用，还是不如自己重写的快，我不明白为什么VB开发小组没有考虑为它写一个对应的__vbaRnd。

不要以为上面的分析没有意义，因为我们可以从现象看本质，也可以从本质来解释现象。比如我们再做一个实验，给你的代码加入一个类模块，你可以试试声明一个和内部方法同名

7

的公有的方法（这是一个很有用的技术，在本系列后面的《错误处理》中我们会用到这种方法），比如我们可以声明一个Public Function Rnd(x) as single，同样我们可以自己写一个同名的MsgBox。但是你试试能不能声明一个Public Function abs(x)，这时VB肯定会弹出一个莫名其妙的编译错误提示框告诉你\缺少标识符\，这种错误发生在你的函数名和VB关键字冲突的时候。但是为什么同样是MATH模块中的函数，abs是关键字，rnd却不是，VB文档里是不会告诉你为什么的，但如果你认真的看了我上面的实验分析，我们就能猜想这是因为VB对需要进一步优化的函数已经做了高度优化处理，VB开发小组为了保护他们的劳动成果，并显示他们对自己优化技术的自信，而禁止我们重写这些函数，同时VB开发小组也承认还有些函数有待进一步优化，所以准许我们重写之。在这里我要提出一个伟大的猜想：凡是能够被重写的函数就能够被优化，就象凡是大于2的偶数就能够被分解成两个质因数的和一样。

说到优化，还应该谈谈直接API调用和使用API类型库的差别，还必须谈谈VB所使用的后端优化器（和VC用的是一样的优化器），还想谈谈如何尽最大可能来使用vTable绑定??（准备在本系列中另写一篇《优化》来谈这些问题）。

看了本地代码，我们再来看看p-code，要是你看了MSDN中关于p-code的原理，你肯定会头大。平心而论p-code真是一个了不起的技术，代码大小平均可以缩小50%。我们把程序2编译成p-code看看，还是用DEPEND来观察，发现它并没有引入__vba开头函数（没有使用优化的VBA函数？），却引入了CallEngine这样的东西（肯定是为了调用p-code伪码解释引擎），而且和Native Code一样都引入了rtcMsgBox（编译生成的p-code在调用MsgBox时应该比在IDE环境中运行的p-code快）。

如果你迫不及待地运行了程序2，你就会发现它将弹出一个应用程序错误对话框，说程序发生异常。别怕，这是因为调用了DebugBreak这个API的缘故，这个API其实就是产生一个Int 3中断，使得我们能够中断程序执行。如果你装了VC这样的支持即时调试的调试器，你可以在错误对话框中点击\取消\，这样可以起动调试器来调试程序。我就是这样跟踪程序运行的。如果你想看看VB生成的程序反汇编代码可以自己试试，我们可以用同样的技术在VB或VBA的IDE中来中断程序执行，比如我们完全可以在

Word的VB编辑器中运行上面程序2的代码，从而中断于Word的进程中，并可观察到VBA生成的p-code代码。比如VB和VBA在IDE中生成的p-code代码就会发现它们这间有很大的不同。

所以，IDE中运行的程序和最终生成的程序是完全不同的。用SPY++看看你在IDE中运行的窗体，你会发现它在VB的主线程下，也就是说在IDE中你用程序做出的窗体和VB IDE工作窗口一样属于VB IDE，你的程序在IDE中运行时申请的资源也属于VB IDE。有些程序在IDE中运行会让IDE死掉（在VB5中写纯API多线程就千万别在IDE中运行，定死无疑，相比之下VB6的IDE健壮得多）。还有些程序可能在IDE中能正常工作，但生成EXE后就工作不了。总之，在写系统程序时要考虑到这种不同可能引起的问题。

3)VB的编译技术，要我怎么夸你，又要我怎么骂你。

8

看了上面对Native Code的高度评价，你可能会对VB做出的东西更有信心了，腰板更直了。是的，作为VB程序员没有什么需要害羞的，一个功力深厚的VB程序员理应拿比普通VC程序员更多的工资，因为他的生产力是VC程序员的好几倍，而做出的程序在质量上和VC做的相差无几。

甚至有大师开玩笑说VB的内置对象就是用VB写出的，比如我们可以自己写Form.cls、Label.ctl，呵呵，我们还真不能排除这种可能性（虽然用VB不可能直接生成vb6.olb）。如果真是这样，看来VB小组自己都对自己的编译优化技术非常有信心。

实际上我们看看VB安装目录下的C2.exe的属性，再看看VC的C2.DLL的属性，就会发现它们是同一个东西，同样Link.exe也是VC的，所以我们完全可以对VB程序的后端优化编译器以及联结放心了。它们根本就是VC开发小组东西，或者VB、VC都是同一个编译器开发小组在做编译模块。总之，我们可以壮着胆说我们VB做的程序其二次优化和联结用的是和VC一样的技术，嘿嘿，你有的我也有，我有的你没有的（纯属诡辩）。

还有，没有任何编译器比VB编译器更快，因为在IDE中VB就是一种解释型语言，这才是VB开发效率高的关键，快得几乎感觉不得编译过程。其请求时编译，后台编译技术更是一只独秀，厉害啊！想想看，别的语言的程序员有多少时间花在了等待代码编译和重新联结上啊！

不要高兴得太早，因为最终的目的还是要生成可执行文件。在VB中没有分块编译和增量联结的功能，VB在生成可执行程序时总是编译所有模块并完全重新联结，而在别的编译语言中我们可以仅编译最近修改过的文件（分块编译），联结时将新生成的代码附在可执行程序的后面，并将原来的代吗标记为作废（增量联结，最终的可执行程序会越来越大，但联结时间大大缩短）。做实验看看，会发现在VB中每次生成可执行文件所花时间都是相同的。我不知VB开发小组为什么不提供分块编译和增量联结的功能，可能VB开发小组认为生成可执行文件在VB中不是经常要做的工作。但是实际上这种理由是说不过去的，因为如前面所说IDE中运行程序和最终程序有很大不同，如我们要经常编译出可执行文件才能真正对它进行Profile，又如我们要调试多线程程序不能在VB IDE中做，在这些情况下每次修改后都要重新生成可执行文件，我们浪费了不少时间去编译已编译过的代码，联结已联结过的程序。我猜想这是因为VB生成可执行程序时进行了全局优化，所以必须得全部重新编译联结。但提供一个新的功能让我们能够生成不进行全局优化的可以分块编译的调试版本，对Vb开发小组应该不是难事吧！（我有一个变通的解决方案，还在试验中）

在来看看VB6安装目录下的VBAEXE6.lib，怎么只有1k大一点，可以猜想里面应该不会有代码，多半是些象vTable这样的函数地址跳转表，或者是些全局常量，我也不知道。但至少说明VB可以用静态联结库了，为什么不把这个功能提供给我们，让我们有更多的选择。

再做个实验看看，做一个标准EXE工程，里面只有一个标准模块，模块里面只一个Sub Main，Sub Main里面什么也没有，将它生成为EXE文件。看看，嚯，有16k多。你要是有时间跟踪这个什么也不做的程序看看，就会知道它要做很多事，初始化Err和App对象，准备COM调用，准备VB、VBA对象库，甚至为使用ActiveX控制也做了准备，嘿嘿，看服

9

务多周到。你必须得用VB对象库中的控制，不用也不行。你再多找几个EXE工程看看，有很多东西相同，都是一个模子做出的，而且你没有选择模子自由。ActiveX工程也是一样，都是Dual双接口，你做的ActiveX控制都必须要躲在一个Extender Object后面。是的，在VB里有很多东西你没有选择的自由。如果需要这种自由要么不用VB，要么就得采取一些未公开的非官方的古怪的技巧（本系列文章最重要的目的之一，就是介绍这样的非官方技巧）。

这又到文章开头说的，VB让我们做事情变得容易的同时也让我们失去了不少自由。在最终代码的生成上则也采取了公式化的做法。当然，我们应该全面地来看待这个问题，如同生产线上生产的东西不一定比手工的精致，群养的家禽不如野味好吃的道理一样，如果需要精致的野味，意味着更多的劳动和更大的成本，这和VB所追求的更容易更便宜的目标是相违背的。

4)VB程序员也得有HACK精神。

本文的最后这个标题是严重离题了，但我想在此为本系列文章定下一个充满HACK精神的基调。HACK精神是什么？没有准确的定义，我的理解是：HACK精神 = 总想探寻未知领域的好奇心 + 凡事总想知道为什么的研究欲 + 总想拿出自己的东西的创新精神 + 解决问题的耐心和恒心。 VB的程序员也一样需要这种精神。

最后，我们都知道VB开发小组已经达上.NET的快车飞起来了，不能不说VB6以后再没有VB的新版本了。微软已经用.NET为我们划出了新的圈子，VB.NET是这个新圈子里的新产物。在圈子里面我们能够飞得更高，但是圈子外面的天空更大，所以我依然乐意站在圈子外，虔诚地祈祷真正的VB7的诞生，阿门。

10

每次看大师的东西到了精彩之处，我就会拍案叫绝：\哇噻，真是想不到！\。在经过很多次这种感慨之后，我发现只要我们动了脑筋，我们自己也能有让别人想不到的东西。于是想到要把这些想不到的东拿出来和大家一起分享，希望抛砖引玉，能引出更多让人想不到的东西。

VB真是想不到系列之二：VB《葵花宝典》--指针技术 关键字：VB、指针、动态内存分配、效率、安全 难度：中级至高级

要求：熟悉VB，掌握基本的C，了解汇编，了解内存分配原理。

想当年东方不败，黑木崖密室一战，仅凭一根绣花针独战四大高手，神出鬼没，堪称天下武林第一高手。若想成为VB里的东方不败，熟习VB《葵花宝典》，掌握VB指针技术，乃是不二的法门。

欲练神功，引刀??，其实掌握VB指针技术，并不需要那么痛苦。因为说穿了，也就那么几招，再勤加练习，终可至神出鬼没之境。废话少说，让我们先从指针的定义说起。

一、指针是什么？

不需要去找什么标准的定义，它就是一个32位整数，在C语言和在VB里都可以用Long类型来表示。在32位Windows平台下它和普通的32位长整型数没有什么不同，只不过它的值是一个内存地址，正是因为这个整数象针一样指向一个内存地址，所以就有了指针的概念。

有统计表明，很大一部分程序缺陷和内存的错误访问有关。正是因为指针直接和内存打交道，所以指针一直以来被看成一个危险的东西。以至于不少语言，如著名的JAVA，都不提供对指针操作的支持，所有的内存访问方面的处理都由编译器来完成。而象C和C++，指针的使用则是基本功，指针给了程序员极大的自由去随心所欲地处理内存访问，很多非常巧妙的东西都要依靠指针技术来完成。

关于一门高级的程序设计语言是不是应该取消指针操作，关于没有指针操作算不算一门语言的优点，我在这里不讨论，因为互联网上关于这方面的没有结果的讨论，已经造成了占用几个GB的资源。无论最终你是不是要下定决心修习指针技术《葵花宝典》，了解这门功夫总是有益处的。

注意：在VB里，官方是不鼓励使用什么指针的，本文所讲的任何东西你都别指望取得官方的技术支持，一切都要靠我们自己的努力，一切都更刺激！ 让我们开始神奇的VB指针探险吧！

二、来看看指针能做什么？有什么用？

先来看两个程序，程序的功能都是交换两个字串：【程序一】：'标准的做法SwapStr

Sub SwapStr(sA As String, sB As String) Dim sTmp As String

sTmp = sA: sA = sB: sB = sTmp End Sub

【程序二】：'用指针的做法SwapPtr

Private Declare Sub CopyMemory Lib \Alias _ (Destination As Any, Source As Any, ByVal Length As Long)

11

Sub SwapPtr(sA As String, sB As String) Dim lTmp As Long

CopyMemory lTmp, ByVal VarPtr(sA), 4

CopyMemory ByVal VarPtr(sA), ByVal VarPtr(sB), 4 CopyMemory ByVal VarPtr(sB), lTmp, 4 End Sub

你是不是以为第一个程序要快，因为它看着简单而且不用调用API（调用API需要额外的处理，VB文档明确指出大量调用API将降低程序性能）。但事实上，在VB集成环境中运行，程序二要比程序一快四分之一；而编译成本机代码或p-code，程序二基本上要比程序一快一倍。下面是两个函数在编译成本机代码后，运行不同次数所花时间的比较： 运行100000次，SwapStr需要170毫秒，SwapPtr需要90毫秒。 运行200000次，SwapStr需要340毫秒，SwapPtr需要170毫秒。 运行2000000次，SwapStr需要3300毫秒，SwapPtr需要1500毫秒。

的确，调用API是需要额外指令来处理，但是由于使用了指针技术，它没有进行临时字串的分配和拷贝，因此速度提高了不少。

怎么样，想不到吧！C/C++程序员那么依赖指针，无非也是因为使用指针往往能更直接的去处理问题的根源，更有驾驭一切的快感。他们不是不知道使用指针的危险，他们不是不愿意开卫星定位无级变速的汽车，只是骑摩托更有快感，而有些地方只有摩托才走得过去。 和在C里类似，在VB里我们使用指针也不过三个理由： 一是效率，这是一种态度一种追求，在VB里也一样；

二是不能不用，因为操作系统是C写的，它时刻都在提醒我们它需要指针；

三是突破限制，VB想照料我们的一切，VB给了我们很强的类型检查，VB象我们老妈一样，对我们关心到有时我们会受不了，想偶尔不听妈妈的话吗？你需要指针！

但由于缺少官方的技术支持，在VB里，指针变得很神秘。因此在C里一些基本的技术，在VB里就变得比较困难。本文的目的就是要提供给大家一种简单的方法，来将C处理指针的技术拿到VB里来，并告诉你什么是可行的，什么可行但必须要小心的，什么是可能但不可行的，什么是根本就不可能的。

三、 程咬金的三板斧

是的，程序二基本上就已经让我们看到VB指针技术的模样了。总结一下，在VB里用指针技术我们需要掌握三样东西：CopyMemory，VarPtr/StrPtr/ObjPtr, AdressOf. 三把斧头，程咬金的三板斧，在VB里Hack的工具。 1、CopyMemory

关于CopyMemory和Bruce McKinney大师的传奇，MSDN的Knowledge Base中就有文章介绍，你可以搜索\的文章。正是这位大师给32位的VB带来了这个可以移动内存的API，也正是有了这个API，我们才能利用指针完成我们原来想都不敢想的一些工作，感谢Bruce McKinney为我们带来了VB的指针革命。

如CopyMemory的声明，它是定义在Kernel32.dll中的RtlMoveMemory这个API，32位C函数库中的memcpy就是这个API的包装，如MSDN文档中所言，它的功能是将从Source指针所指处开始的长度为Length的内存拷贝到Destination所指的内存处。它不会管我们的程序有没有读写该内存所应有的权限，一但它想读写被系统所保护的内存时，我们就会得到著名的Access Violation Fault(内存越权访问错误)，甚至会引起更著名的general protection

12

(GP) fault（通用保护错误）。所以，在进行本系列文章里的实验时，请注意随时保存你的程序文件，在VB集成环境中将\工具\选项\中的\环境\选项卡里的\启动程序时\设为\保存改变\，并记住在\立即\窗口中执行危险代码之前一定要保存我们的工作成果。

2、VatPtr/StrPtr/ObjPtr

它们是VB提供给我们的好宝贝，它们是VBA函数库中的隐藏函数。为什么要隐藏？因为VB开发小组，不鼓励我们用指针嘛。

实际上这三个函数在VB运行时库MSVBVM60.DLL（或MSVBVM50.DLL）中是同一个函数VarPtr（可参见我在本系列第一篇文章里介绍的方法）。 其库型库定义如下：

[entry(\

long _stdcall VarPtr([in] void* Ptr); [entry(\

long _stdcall StrPtr([in] BSTR Ptr); [entry(\

long _stdcall ObjPtr([in] IUnknown* Ptr); 即然它们是VB运行时库中的同一个函数，我们也可以在VB里用API方式重新声明这几个函数，如下：

Private Declare Function ObjPtr Lib \ (var As Object) As Long

Private Declare Function VarPtr Lib \ _ (var As Any) As Long

（没有StrPtr，是因为VB对字符串处理方式有点不同，这方面的问题太多，在本系列中另用一篇《VB字符串全攻略》来详谈。

顺便提一下，听说VB.NET里没有这几个函数，但只要还能调用API，我们就可以试试上面的几个声明，这样在VB.NET里我们一样可以进行指针操作。

但是请注意，如果通过API调用来使用VarPtr，整个程序二SwapPtr将比原来使用内置VarPtr函数时慢6倍。）

如果你喜欢刨根问底，那么下面就是VarPtr函数在C和汇编语言里的样子： 在C里样子是这样的：
long VarPtr(void* pv){ return (long)pv; }

所对就的汇编代码就两行：

mov eax,dword ptr [esp+4]

ret 4 '弹出栈里参数的值并返回。

之所以让大家了解VarPtr的具体实现，是想告诉大家它的开销并不大，因为它们不过两条指令，即使加上参数赋值、压栈和调用指令，整个获取指针的过程也就六条指令。当然，同样的功能在C语言里，由于语言的直接支持，仅需要一条指令即可。但在VB里，它已经算是最快的函数了，所以我们完全不用担心使用VarPtr会让我们失去效率！速度是使用指针技术的根本要求。

一句话，VarPtr返回的是变量所在处的内存地址，也可以说返回了指向变量内存位置的指针，它是我们在VB里处理指针最重要的武器之一。

13

3、ByVal和ByRef

ByVal传递的参数值，而ByRef传递的参数的地址。在这里，我们不用去区别传指针/传地址/传引用的不同，在VB里，它们根本就是一个东西的三种不同说法，即使VB的文档里也有地方在混用这些术语（但在C++里的确要区分指针和引用） 初次接触上面的程序二SwapPtr的朋友，一定要搞清在里面的CopyMemory调用中，在什么地方要加ByVal，什么地方不加（不加ByVal就是使用VB缺省的ByRef）准确的理解传值和传地址（指针）的区别，是在VB里正确使用指针的基础。 现在一个最简单的实验来

看这个问题，如下面的程序三： 【程序三】：'体会ByVal和ByRef Sub TestCopyMemory() Dim k As Long k = 5

Note: CopyMemory ByVal VarPtr(k), 40000, 4 Debug.Print k End Sub

上面标号Note处的语句的目的，是将k赋值为40000，等同于语句k=40000，你可以在\立即\窗口试验一下，会发现k的值的确成了40000。 实际上上面这个语句，翻译成白话，就是从保存常数40000的临时变量处拷贝4个字节到变量k所在的内存中。

现在我们来改变一个Note处的语句，若改成下面的语句： Note2: CopyMemory ByVal VarPtr(k), ByVal 40000, 4

这句话的意思就成了，从地址40000拷贝4个字节到变量k所在的内存中。由于地址40000所在的内存我们无权访问，操作系统会给我们一个Access Violation内存越权访问错误，告诉我们\试图读取位置0x00009c40处内存时出错，该内存不能为'Read'\。 我们再改成如下的语句看看。

Note3: CopyMemory VarPtr(k), 40000, 4

这句话的意思就成了，从保存常数40000的临时变量处拷贝4个字节到到保存变量k所在内存地址值的临时变量处。这不会出出内存越权访问错误，但k的值并没有变。 我们可以把程序改改以更清楚的休现这种区别，如下面的程序四： 【程序四】：'看看我们的东西被拷贝到哪儿去了 Sub TestCopyMemory()

Dim i As Long, k As Long k = 5

i = VarPtr(k)

NOTE4: CopyMemory i, 40000, 4 Debug.Print k Debug.Print i i = VarPtr(k)

NOTE5: CopyMemory ByVal i, 40000, 4 Debug.Print k End Sub

程序输出： 5

14

40000 40000

由于NOTE4处使用缺省的ByVal，传递的是i的地址（也就是指向i的指针），所以常量40000拷贝到了变量i里，因此i的值成了40000，而k的值却没有变化。但是，在NOTE4前有：i=VarPtr(k)，本意是要把i本身做为一个指针来使用。这时，我们必须如NOTE5那样用ByVal来传递指针i，由于i是指向变量k的指针，所以最后常量40000被拷贝了变量k里。

希望你已经理解了这种区别，在后面问题的讨论中，我还会再谈到它。

4、AddressOf

它用来得到一个指向VB函数入口地址的指针，不过这个指针只能传递给API使用，以使得API能回调VB函数。

本文不准备详细讨论函数指针，关于它的使用请参考VB文档。

5、拿来主义。

实际上，有了CopyMemory，VarPtr，AddressOf这三把斧头，我们已经可以将C里基本的指针操作拿过来了。

如下面的C程序包括了大部分基本的指针指针操作： struct POINT{ int x; int y; };

int Compare(void* elem1, void* elem2){} void PtrDemo(){ //指针声明:

char c = 'X'; //声明一个char型变量 char* pc; long* pl; //声明普通指针 POINT* pPt; //声明结构指针 void* pv; //声明无类型指针

int (*pfnCastToInt)(void *, void*);//声明函数指针: //指针赋值：

pc = &c; //将变量c的地址值赋给指针pc pfnCompare = Compare; //函数指针赋值。 //指针取值：

c = *pc; //将指针pc所指处的内存值赋给变量c //用指针赋值：

*pc = 'Y' //将'Y'赋给指针pc所指内存变量里。 //指针移动： pc++; pl--; }

这些对指针操作在VB里都有等同的东西，

前面讨论ByVal和ByRef时曾说过传指针和传地址是一回事，实际上当我们在VB里用缺省的ByRef声明函数参数时，我们已经就声明了指针。 如一个C声明的函数：long Func(char* pc)

15

其对应的VB声明是：Function Func(pc As Byte) As Long 这时参数pc使用缺省的ByRef传地址方式来传递，这和C里用指针来传递参数是一样。 那么怎么才能象C里那样明确地声明一个指针呢？

很简单，如前所说，用一个32位长整数来表达指针就行。在VB里就是用Long型来明确地声明指针，我们不用区分是普通指针、无类型指针还是函数指针，通通都可用Long来声明。而给一个指针赋值，就是赋给它用VarPar得到的另一个变量的地址。具体见程序五。 【程序五】： 同C一样，各种指针。 Type POINT X As Integer Y As Integer End Type

Public Function Compare(elem1 As Long, elem2 As Long) As Long '

End Function

Function FnPtrToLong(ByVal lngFnPtr As Long) As Long FnPtrToLong = lngFnPtr End Function Sub PtrDemo()

Dim l As Long, c As Byte, ca() As Byte, Pt As POINT

Dim pl As Long, pc As Long, pv As Long, pPt As Long, pfnCompare As Long c = AscB(\

pl = VarPtr(l) '对应C里的long、int型指针 pc = VarPtr(c) '对应char、short型指针 pPt = VarPtr(Pt) '结构指针

pv = VarPtr(ca(0)) '字节数组指针，可对应任何类型，也就是void* pfnCompare = FnPtrToLong(AddressOf Compare) '函数指针 CopyMemory c, ByVal pc, LenB(c) '用指针取值

CopyMemory ByVal pc, AscB(\用指针赋值 pc = pc + LenB(c) : pl = pl - LenB(l) '指针移动 End Sub

我们看到，由于VB不直接支持指针操作，在VB里用指针取值和用指针赋值都必须用CopyMemory这个API，而调用API的代价是比较高的，这就决定了我们在VB里使用指针不能象在C里那样自由和频繁，我们必须要考虑指针操作的代价，在后面的\指针应用\我们会再变谈这个问题。

程序五中关于函数指针的问题请参考VB文档，无类型指针void*会在下面\关于Any的问题\里说。

程序五基本上已经包括了我们能在VB里进行的所有指针操作，仅此而已。

下面有一个小测试题，如果现在你就弄懂了上面程咬金的三板斧，你就应该能做得出来。 上面提到过，VB.NET中没有VarPtr，我们可以用声明API的方式来引入MSVBVM60.DLL中的VarPtr。现在的问题如果不用VB的运行时DLL文件，你能不能自己实现一个ObjPtr。答案在下一节后给出。

四、指针使用中应注意的问题

16

1、关于ANY的问题

如果以一个老师的身份来说话，我会说：最好永远也不要用Any！是的，我没说错，是永远！所以我没有把它放在程咬金的三板斧里。当然，这个问题和是不是应该使用指针这个问题一样会引发一场没有结果的讨论，我告诉你的只是一个观点，因为有时我们会为了效率上的一点点提高或想偷一点点懒而去用Any，但这样做需要要承担风险。

Any不是一个真正的类型，它只是告诉VB编译器放弃对参数类型的检查，这样，理论上，我们可以将任何类型传递给API。

Any在什么地方用呢？让我们来看看，在VB文档里的是怎么说的，现在就请打开MSDN(Visual Studio 6自带的版本)，翻到\文档\使用Visual Basic\部件工具指南\访问DLL和Windows API\部分，再看看\将 C 语言声明转换为 Visual Basic 声明\这一节。文档里告诉我们，只有C的声明为LPVOID和NULL时，我们才用Any。实际上如果你愿意承担风险，所有的类型你都可以用Any。当然，也可以如我所说，永远不要用Any。

为什么要这样？那为什么VB官方还要提供Any？是信我的，还是信VB官方的？有什么道理不用Any？

如前面所说，VB官方不鼓励我们使用指针。因为VB所标榜的优点之一，就是没有危险的指针操作，所以的内存访问都是受VB运行时库控制的。在这一点上，JAVA语言也有着同样的标榜。但是，同JAVA一样，VB要避免使用指针而得到更高的安全性，就必须要克服没有指针而带来的问题。VB已经尽最大的努力来使我们远离指针的同时拥有强类型检查带来的安全性。但是操作系统是C写的，里面到处都需要指针，有些指针是没有类型的，就是C程序员常说的可怕的void*无类型指针。它没有类型，因此它可以表示所有类型。如CopyMemory所对应的是C语言的memcpy，它的声明如下： void *memcpy(void *dest, const void *src, size_t count);

因memcpy前两个参数用的是void*，因此任何类型的参数都可以传递给他。

一个用C的程序员，应该知道在C函数库里这样的void*并不少见，也应该知道它有多危险。无论传递什么类型的变量指针给上面memcpy的void*，C编译器都不会报错或给任何警告。

在VB里大多数时候，我们使用Any就是为了使用void*，和在C里一样，VB也不对Any进行类型检查，我们也可以传递任何类型给Any，VB编译器也都不会报错或给任何警告。

但程序运行时会不会出错，就要看使用它时是不是小心了。正因为在C里很多错误是和void*相关的，所以，C++鼓励我们使用satic_cast来明确指出这种不安全的类型的转换，已利于发现错误。

说了这么多C/C++，其实我是想告诉所有VB的程序员，在使用Any时，我们必须和C/C++程序员使用void*一样要高度小心。

VB里没有satic_cast这种东西，但我们可以在传递指针时明确的使用long类型，并且用VarPtr来取得参数的指针，这样至少已经明确地指出我们在使用危险的指针。如程序二经过这样的处理就成了下面的程序： 【程序五】：'使用更安全的CopyMemory，明确的使用指针！

Private Declare Sub CopyMemory Lib \Alias \(ByVal Destination As Long, ByVal Source As Long, ByVal Length As Long) Sub SwapStrPtr2(sA As String, sB As String) Dim lTmp As Long

Dim pTmp As Long, psA As Long, psB As Long

17

pTmp = VarPtr(lTmp): psA = VarPtr(sA): psB = VarPtr(sB) CopyMemory pTmp, psA, 4 CopyMemory psA, psB, 4 CopyMemory psB, pTmp, 4 End Sub

注意，上面CopyMemory的声明，用的是ByVal和long，要求传递的是32位的地址值，当我们将一个别的类型传递给这个API时，编译器会报错，比如现在我们用下面的语句： 【程序六】：'有点象【程序四】，但将常量40000换成了值为1的变量.

Private Declare Sub CopyMemory Lib \Alias \(ByVal Destination As Long, ByVal Source As Long, Length As Long) Sub TestCopyMemory()

Dim i As Long，k As Long, z As Interger k = 5 : z = 1 i = VarPtr(k)

'下面的语句会引起类型不符的编译错误，这是好事！ 'CopyMemory i, z, 4 '应该用下面的

CopyMemory i, ByVal VarPtr(z), 2 Debug.Print k End Sub

编译会出错！是好事！这总比运行时不知道错在哪儿好！ 象程序四那样使用Any类型来声明CopyMemory的参数，VB虽然不会报错，但运行时结果却是错的。不信，你试试将程序四中的40000改为1，结果i的值不是我们想要的1，而是327681。为什么在程序四中，常量为1时结果会出错，而常量为40000时结果就不错？ 原因是VB对函数参数中的常量按Variant的方式处理。是1时，由于1小于Integer型的最大值32767，VB会生成一个存储值1的Integer型的临时变量，也就是说，当我们想将1用CopyMemroy拷贝到Long型的变量i时，这个常量1是实际上是Integer型临时变量！VB里Integer类型只有两个字节，而我们实际上拷贝了四个字节。知道有多危险了吧！没有出内存保护错误那只是我们的幸运！

如果一定要解释一下为什么i最后变成了327681，这是因为我们将k的低16位的值5也拷贝到了i值的高16位中去了，因此有5*65536+1=327681。详谈这个问题涉及到VB局部变量声明顺序，CopyMemory参数的压栈顺序，long型的低位在前高位在后等问题。如果你对这些问题感兴趣，可以用本系列第一篇文章所提供的方法(DebugBreak这个API和VC调试器)来跟踪一下，可以加深你对VB内部处理方式的认识，由于这和本文讨论的问题无关，所以就不详谈了。到这里，大家应该明白，程序三和程序四实际上有错误！！！我在上面用常量40000而不用1，不是为了在文章中凑字数，而是因为40000这个常量大于32767，会被VB解释成我们需要的Long型的临时变量，只有这样程序三和程序四才能正常工作。对不起，我这样有意的隐藏错误只是想加深你对Any危害的认识。 总之，我们要认识到，编译时就找到错误是非常重要的，因为你马上就知道错误的所在。所以我们应该象程序五和程序六那样明确地用long型的ByVal的指针，而不要用Any的ByRef的指针。

但用Any已经如此的流行，以至很多大师们也用它。它唯一的魅力就是不象用Long型指针那样，需要我们自己调用VarPtr来得到指针，所有处理指针的工作由VB编译器来完成。所以在参数的处理上，只用一条汇编指令：push [i]，而用VarPtr时，由于需要函数调用，

18

因此要多用五条汇编指令。五条多余的汇编指令有时的确能我们冒着风险去用Any。 VB开发小组提供Any，就是想用ByRef xxx As Any来表达void* xxx。我们也完全可以使用VarPtr和Long型的指针来处理。我想，VB开发小组也曾犹豫过是公布VarPtr，还是提供Any，最后他们决定还是提供Any，而继续隐瞒VarPtr。的确，这是个两难的决定。但是经过我上面的分析，我们应该知道，这个决定并不符合VB所追求的\更安全\的初衷。因为它可能会隐藏类型不符的错误，调试和找到这种运行时才产生的错误将花费更多的时间和精力。

所以我有了\最好永远不要用Any\这个\惊人\的结论。

不用Any的另一个好处是，简化了我们将C声明的API转换成VB声明的方式，现在它变成了一句话：除了VB内置的可以进行类型检查的类型外，所以其它的类型我们都应该声明成Long型。

2、关于NULL的容易混淆的问题 有很多文章讲过，一定要记在心里：

VbNullChar 相当于C里的'\\0'，在用字节数组构造C字串时常用它来做最后1个元素。vbNullString 这才是真正的NULL，就是0，在VB6中直接用0也可以。

只有上面的两个是API调用中会用的。还有Empty、Null是Variant，而Nothing只和类对象有关，一般API调用中都不会用到它们。

另：本文第三节曾提出一个小测验题，做出来了吗？现在公布正确答案： 【测验题答案】

Function ObjPtr(obj as Object) as long Dim lpObj As Long

CopyMemory lpObj, Obj, 4 ObjectPtr = lpObj End Function

五、VB指针应用

如前面所说VB里使用指针不象C里那样灵活，用指针处理数据时都需要用CopyMemory将数据在指针和VB能够处理的变量之间来回拷贝，这需要很大的额外开销。因此不是所有C里的指针操作都可以移值到VB里来，我们只应在需要的时候才在VB里使用指针。

1、动态内存分配：完全不可能、可能但不可行，VB标准

在C和C++里频繁使用指针的一个重要原因是需要使用动态内存分配，用Malloc或New来从堆栈里动态分配内存，并得到指向这个内存的指针。在VB里我们也可以自己 用API来实现动态分配内存，并且实现象C里的指针链表。

但我们不可能象C那样直接用指针来访问这样动态分配的内存，访问时我们必须用CopyMemory将数据拷贝到VB的变量内，大量的使用这种技术必然会降低效率，以至于要象C那样用指针来使用动态内存根本就没有可行性。要象C、PASCAL那样实现动态数据结构，在VB里还是应该老老实实用对象技术来实现。

本文配套代码中的LinkedList里有完全用指针实现的链表，它是使用HeapAlloc从堆栈中动态分配内存，另有一个调用FindFirstUrlCacheEntry这个API来操作IE的Cache的小程

19

序IECache，它使用了VirtualAlloc来动态分配内存。但实际上这都不是必须的，VB已经为我们提供了标准的动态内存分配的方法，那就是： 对象、字符串和字节数组

限于篇幅，关于对象的技术这里不讲，LinkedList的源代码里有用对象实现的链表，你可以参考。

字符串可以用Space$函数来动态分配，VB的文档里就有详细的说明。

关于字节数组，这里要讲讲，它非常有用。我们可用Redim来动态改变它的大小，并将指向它第一个元素的指针传给需要指针的API，如下： dim ab() As Byte , ret As long

'传递Null值API会返回它所需要的缓冲区的长度。 ret = SomeApiNeedsBuffer(vbNullString) '动态分配足够大小的内存缓冲区 ReDim ab(ret) As Byte

'再次把指针传给API，此时传字节数组第一个元素的指针。 SomeApiNeedsBuffer(ByVal VarPtr(ab(1)))

在本文配套程序中的IECache中，我也提供了用字节数组来实现动态分配缓冲区的版本，比用VirtualAlloc来实现更安全更简单。

2、突破限制

下面是一个突破VB类型检查来实现特殊功能的经典应用，出自Bruce Mckinney的《HardCore Visual Basic》一书。

将一个Long长整数的低16位作为Interger型提取出来， 【程序七】 '标准的方法，也是高效的方法，但不容易理解。 Function LoWord(ByVal dw As Long) As Integer If dw And &H8000& Then

LoWord = dw Or &HFFFF0000 Else

LoWord = dw And &HFFFF& End If End Function

【程序八】 '用指针来做效率虽不高，但思想清楚。 Function LoWord(ByVal dw As Long) As Integer

CopyMemory ByVal VarPtr(LoWord), ByVal VarPtr(dw), 2 End Function

3、对数组进行批量操作

用指针进行大批量数组数据的移动，从效率上考虑是很有必要的，看下面的两个程序，它们功能都是将数组的前一半数据移到后一半中： 【程序九】： '标准的移动数组的做法

Private Sub ShitArray(ab() As MyType) Dim i As Long, n As Long n = CLng(UBound(ab) / 2) For i = 1 To n

Value(n + i) = Value(i)

20

Value(i).data = 0 Next End Sub 【程序十】： '用指针的做法

Private Declare Sub CopyMemory Lib \ (ByVal dest As Long, ByVal source As Long, ByVal bytes As Long) Private Declare Sub ZeroMemory Lib \ (ByVal dest As Long, ByVal numbytes As Long)

Private Declare Sub FillMemory Lib \ (ByVal dest As Long, ByVal Length As Long, ByVal Fill As Byte)

Private Sub ShitArrayByPtr(ab() As MyTpye) Dim n As Long

n = CLng(UBound(ab) / 2) Dim nLenth As Long nLenth = Len(Value(1)) 'DebugBreak

CopyMemory ByVal VarPtr(Value(1 + n)), _ ByVal VarPtr(Value(1)), n * nLenth ZeroMemory ByVal VarPtr(Value(1)), n * nLenth End Sub

当数组较大，移动操作较多（比如用数组实现HashTable）时程序十比程序九性能上要好得多。

程序十中又介绍两个在指针操作中会用到的API: ZeroMemory是用来将内存清零；FillMemory用同一个字节来填充内存。当然，这两个API的功能，也完全可以用CopyMemory来完成。象在C里一样，作为一个好习惯，在VB里我们也可以明确的用ZeroMemory来对数组进行初始化，用FillMemory在不立即使用的内存中填入怪值，这有利于调试。 4、最后的一点

当然，VB指针的应用决不止这些，还有什么应用就要靠自己去摸索了。对于对象指针和字符串指针的应用我会另写文章来谈，做为本文的结束和下一篇文章《VB字符串全攻略》的开始，我在这里给出交换两个字符串的最快的方法： 【程序十一】'交换两个字符串最快的方法

Private Declare Sub CopyMemory Lib \Alias _ (Destination As Any, Source As Any, ByVal Length As Long)

Sub SwapStrPtr3(sA As String, sB As String) Dim lTmp As Long

Dim pTmp As Long, psA As Long, psB As Long

pTmp = StrPtr(sA): psA = VarPtr(sA): psB = VarPtr(sB) CopyMemory ByVal psA, ByVal psB, 4 CopyMemory ByVal psB, pTmp, 4 End Sub

对不起，为了一点点效率，又用了Any！关于StrPtr，下一篇文章我会来谈。 自己来试试吧！欲练神功，赶快行动！

21

每次看大师的东西到了精彩之处，我就会拍案叫绝：\哇噻，真是想不到！\。在经过很多次这种感慨之后，我发现只要我们动了脑筋，我们自己也能有让别人想不到的东西。于是想到要把这些想不到的东拿出来和大家一起分享，希望抛砖引玉，能引出更多让人想不到的东西。

本系列文章可见：

http://www.csdn.net/develop/list_article.asp?author=AdamBear

VB真是想不到系列之三：VB指针葵花宝典之函数指针 关键字：VB、HCAK、指针、函数指针、效率、数组、对象、排序 难度：中级

要求：熟悉VB，了解基本的排序算法，会用VC更好。

引言：

不知大家在修习过本系列第二篇《VB指针葵花宝典》后有什么感想，是不是觉得宝典过于偏重内功心法，而少了厉害的招式。所以，今天本文将少讲道理，多讲招式。不过，还是请大家从名门正派的内功心法开始学起，否则会把九阴真经练成九阴白骨爪。 今天，我们重点来谈谈函数指针的实际应用。 接着上一篇文章，关于字串的问题，听CSDN上各位网友的建议，我不准备写什么《VB字符串全攻略》了，关于BSTR的结构，关于调用API时字串在UNICODE和ANSI之间的转换问题，请参考MSDN的Partial Books里的《Win32 API Programming with Visual Basic》里的第六章《Strings》。今天就让我们先忘掉字符串，专注于函数指针的处理上来。

一、函数指针

AddressOf得到一个VB内部的函数指针，我们可以将这个函数指针传递给需要回调这个函数的API，它的作用就是让外部的程序可以调用VB内部的函数。 但是VB里函数指针的应用，远不象C里应用那么广泛，因为VB文档里仅介绍了如何将函数指针传递给API以实现回调，并没指出函数指针诸多神奇的功能，因为VB是不鼓励使用指针的，函数指针也不例外。

首先让我们对函数指针的使用方式来分个类。

1、回调。这是最基本也是最重要的功能。比如VB文档里介绍过的子类派生技术，它的核心就是两个API：SetWindowLong和CallWindowProc。

我们可以使SetWindowLong这个API来将原来的窗口函数指针换成自己的函数指针，并将原来的窗口函数指针保存下来。这样窗口消息就可以发到我们自己的函数里来，并且我们随时可以用CallWindowProc来调用前面保存下来的窗口指针，以调用原来的窗口函数。这样，我们可以在不破坏原有窗口功能的前提下处理钩入的消息。

具体的处理，我们应该很熟悉了，VB文档也讲得很清楚了。这里需要注意的就是CallWindowProc这个API，在后面我们将看到它的妙用。 在这里我们称回调为让\外部调用内部的函数指针\。

2、程序内部使用。比如在C里我们可以将C函数指针作为参数传递给一个需要函数指针的C函数，如后面还要讲到的C库函数qsort，它的声明如下：

#define int (__cdecl *COMPARE)(const void *elem1, const void *elem2) void qsort(void *base, size_t num, size_t width, COMPARE pfnCompare);

它需要一个COMPARE类型函数指针，用来比较两个变量大小的，这样排序函数可以调用

22

这个函数指针来比较不同类型的变量，所以qsort可以对不同类型的变量数组进行排序。 我们姑且称这种应用为\从内部调用内部的函数指针\。 3、调用外部的函数

也许你会问，用API不就是调用外部的函数吗？是的，但有时候我们还是需要直接获取外部函数的指针。比如通过LoadLibrary动态加载DLL，然后再通过GetProcAddress得到我们需要的函数入口指针，然后再通过这个函数指针来调用外部的函数，这种动态载入DLL的技术可以让我们更灵活的调用外部函数。

我们称这种方式为\从内部调用外部的函数指针\

4、不用说，就是我们也可控制\从外部调用外部的函数指针\。不是没有，比如我们可以加载多个DLL，将其中一个DLL中的函数指针传到另一个DLL里的函数内。 上面所分的\内\和\外\都是相对而言（DLL实际上还是在进程内），这样分类有助于以后我们谈问题，请记住我上面的分类，因为以后的文章也会用到这个分类来分析问题。

函数指针的使用不外乎上面四种方式。但在实际使用中却是灵活多变的。比如在C++里继承和多态，在COM里的接口，都是一种叫vTable的函数指针表的巧妙应用。使用函数指针，可以使程序的处理方式更加高效、灵活。

VB文档里除了介绍过第一方式外，对其它方式都没有介绍，并且还明确指出不支持“Basic 到 Basic”的函数指针（也就是上面说的第二种方式），实际上，通过一定的HACK，上面四种方式均可以实现。今天，我们就来看看如何来实现第二种方式，因为实现它相对来说比较简单，我们先从简单的入手。至于如何在VB内调用外部的函数指针，如何在VB里通过处理vTable接口函数指针跳转表来实现各种函数指针的巧妙应用，由于这将涉及COM内部原理，我将另文详述。

其实VB的文档并没有说错，VB的确不支持“Basic 到 Basic”的函数指针，但是我们可以绕个弯子来实现，那就是先从\到API\，然后再用第一种方式\外部调用内部的函数指针\来从\到BASIC\，这样就达到了第二种方式从\到 Basic\的目的，这种技术我们可以称之为\强制回调\，只有VB里才会有这种古怪的技术。

说得有点绕口，但是仔细想想窗口子类派生技术里CallWindowProc，我们可以用CallWindowProc来强制外部的操作系统调用我们原来的保存的窗口函数指针，同样我们也完全可以用它来强制调用我们内部的函数指针。

呵呵，前面说过要少讲原理多讲招式，现在我们就来开始学习招式吧！

考虑我们在VB里来实现和C里一样支持多关键字比较的qsort。完整的源代码见本文配套代码，此处仅给出函数指针应用相关的代码。 '当然少不了的CopyMemory，不用ANY的版本。 Declare Sub CopyMemory Lib \

\ ByVal numBytes As Long)

'嘿嘿，看下面是如何将CallWindowProc的声明做成Compare声明的。 Declare Function Compare Lib \

\ ByVal pElem2 As Long, ByVal unused1 As Long, _ ByVal unused2 As Long) As Integer

'注：ByVal xxxxx As Long，还记得吧！这是标准的指针声明方法。

23

'声明需要比较的数组元素的结构 Public Type TEmployee Name As String Salary As Currency End Type

'再来看看我们的比较函数 '先按薪水比较，再按姓名比较

Function CompareSalaryName(Elem1 As TEmployee, _

Elem2 As TEmployee, _ unused1 As Long, _

unused2 As Long) As Integer Dim Ret As Integer

Ret = Sgn(Elem1.Salary - Elem2.Salary) If Ret = 0 Then

Ret = StrComp(Elem1.Name, Elem2.Name, vbTextCompare) End If

CompareSalaryName = Ret End Function

'先按姓名比较，再按薪水比较

Function CompareNameSalary(Elem1 As TEmployee, _ Elem2 As TEmployee, _ unused1 As Long, _

unused2 As Long) As Integer Dim Ret As Integer

Ret = StrComp(Elem1.Name, Elem2.Name, vbTextCompare) If Ret = 0 Then

Ret = Sgn(Elem1.Salary - Elem2.Salary) End If

CompareNameSalary = Ret End Function

最后再看看我们来看看我们最终的qsort的声明。

Sub qsort(ByVal ArrayPtr As Long, ByVal nCount As Long, _

ByVal nElemSize As Integer, ByVal pfnCompare As Long)

上面的ArrayPtr是需要排序数组的第一个元素的指针，nCount是数组的元素个数，nElemSize是每个元素大小，pfnCompare就是我们的比较函数指针。这个声明和C库函数里的qsort是极为相似的。

和C一样，我们完全可以将Basic的函数指针传递给Basic的qsort函数。 使用方式如下：

Dim Employees(1 To 10000) As TEmployee

'假设下面的调用对Employees数组进行了赋值初始化。 Call InitArray()

'现在就可以调用我们的qsort来进行排序了。

24

Call qsort(VarPtr(Employees(1)), UBound(Employees), _

LenB(Employees(1)), AddressOf CompareSalaryName) '或者先按姓名排，再按薪水排

Call qsort(VarPtr(Employees(1)), UBound(Employees), _

LenB(Employees(1)), AddressOf CompareNameSalary)

聪明的朋友们，你们是不是已经看出这里的奥妙了呢？作为一个测验，你能现在就给出在qsort里使用函数指针的方法吗？比如现在我们要通过调用函数指针来比较数组的第i个元素和第j个元素的大小。

没错，当然要使用前面声明的Compare（其实就是CallWindowProc）这个API来进行强制回调。

具体的实现如下：

Sub qsort(ByVal ArrayPtr As Long, ByVal nCount As Long, _

ByVal nElemSize As Integer, ByVal pfnCompare As Long) Dim i As Long, j As Long

'这里省略快速排序算法的具体实现，仅给出比较两个元素的方法。 If Compare(pfnCompare, ArrayPtr + (i - 1) * nElemSize, _ ArrayPtr + (j - 1) * nElemSize, 0, 0) > 0 Then

'如果第i个元素比第j个元素大则用CopyMemory来交换这两个元素。 End If End Sub

招式介绍完了，明白了吗？我再来简单地讲解一下上面Compare的意思，它非常巧妙地利用了CallWindowProc这个API。这个API需要五个参数，第一个参数就是一个普通的函数指针，这个API能够强马上回调这个函数指针，并将这个API的后四个Long型的参数传递给这个函数指针所指向的函数。这就是为什么我们的比较函数必须要有四个参数的原因，因为CallWindowProc这个API要求传递给的函数指针必须符合WndProc函数原形，WndProc的原形如下：

LRESULT (CALLBACK* WNDPROC) (HWND, UINT, WPARAM, LPARAM);

上面的LRESULT、HWND、UINT、WPARAM、LPARAM都可以对应于VB里的Long型，这真是太好了，因为Long型可以用来作指针嘛！

再来看看工作流程，当我们用AddressOf CompareSalaryName做为函数指针参数来调用qsort时，qsort的形参pfnCompare被赋值成了实参CompareSalaryName的函数指针。这时，调用Compare来强制回调pfnCompare，就相当于调用了如下的VB语句： Call CompareSalaryName(ArrayPtr + (i - 1) * nElemSize, _ ArrayPtr + (j - 1) * nElemSize, 0, 0)

这不会引起参数类型不符错误吗？CompareSalaryName的前两个参数不是TEmployee类型吗？的确，在VB里这样调用是不行的，因为VB的类型检查不会允许这样的调用。但是，实际上这个调用是API进行的回调，而VB不可能去检查API回调的函数的参数类型是一个普通的Long数值类型还是一个结构指针，所以也可以说我们绕过了VB对函数参数的类型检查，我们可以将这个Long型参数声明成任何类型的指针，我们声明成什么，VB就认为是什么。所以，我们要小心地使用这种技术，如上面最终会传递给CompareSalaryName函数的参数\只不过是一个地址，VB不会对这个地址进行检查，它总是将这个地址当做一个TEmployee类型的指针，如果不小心用成了\+ i *

25

nElemSize\，那么当i是最后一个元素时，我们就会引起内存越权访问错误，所以我们要和在C里处理指针一样注意边界问题。

函数指针的巧妙应用这里已经可见一斑了，但是这里介绍的方法还有很大的局限性，我们的函数必须要有四个参数，更干净的做法还是在VC或Delphi里写一个DLL，做出更加符合要求的API来实现和CallWindowProc相似的功能。我跟踪过CallWindowProc的内部实现，它要做许多和窗口消息相关的工作，这些工作在我们这个应用中是多余的。其实实现强制回调API只需要将后几个参数压栈，再call第一个参数就行了，不过几条汇编指令而已。 正是因为CallWindowProc的局限性，我们不能够用它来

调用外部的函数指针，以实现上面说的第三种函数指针调用方式。要实现第三种方式，Matt Curland大师提供了一个噩梦一般的HACK方式，我们要在VB里凭空构造一个IUnknown接口，在IUnknown接口的vTable原有的三个入口后再加入一个新入口，在新入口里插入机器代码，这个机器代码要处理掉this指针，最后才能调用到我们给的函数指针，这个函数指针无论是内部的还是外部的都一样没问题。在我们深入讨论COM内部原理时我会再来谈这个方法。

另外，排序算法是个见仁见智的问题，我本来想，在本文提供一个最通用性能最好的算法，这种想法虽好，但是不可能有在任何情况下都“最好”的算法。本文提供的用各种指针技术来实现的快速排序方法，应该比用对象技术来实现同样功能快不少，内存占用也少得多。可是就是这个已经经过了我不少优化的快速排序算法，还是比不了ShellSort，因为ShellSort实现上简单。从算法的理论上来讲qsort应该比ShellSort平均性能好，但是在VB里这不一定（可见本文配套代码，里面也提供了VBPJ一篇专栏的配套代码ShellSort，非常得棒，本文的思想就取自这个ShellSort）。

但是应当指出无论是这里的快速排序还是ShellSort，都还可以大大改进，

因为它们在实现上需要大量使用CopyMemroy来拷贝数据（这是VB里使用指针的缺点之一）。其实，我们还有更好的方法，那就是Hack一下VB的数组结构，也就是COM自动化里的SafeArray，我们可以一次性的将SafeArray里的各个数组元素的指针放到一个long型数组里，我们无需CopyMemroy，我们仅需交换Long型数组里的元素就可以达到实时地交换SafeArray数组元素指针的目的，数据并没有移动，移动的仅仅是指针，可以想象这有快多。在下一篇文章《VB指针葵花宝典之数组指针》中我会来介绍这种方法。

26

VB真是想不到系列之四：VB指针葵花宝典之SafeArray

关键字：VB、HCAK、指针、SafeArray、数组指针、效率、数组、排序 难度：中级或高级

要求：熟悉VB，了解基本的排序算法，会用VC更好。

引言：

上回说到，虽然指针的运用让我们的数组排序在性能上有了大大的提高，但是CopyMemory始终是我们心里一个挥之不去的阴影，因为它还是太慢。在C里我们用指针，从来都是来去自如，随心所欲，四两拨千斤；而在VB里，我们用指针却要瞻前顾后，哪怕一个字节都要用到CopyMemory乾坤大挪移，真累。今天我们就来看看，能不能让VB里的指针也能指哪儿打哪儿，学学VB指针的凌波微步。各位看官，您把茶端好了。

一、帮VB做点COM家务事 本系列开张第一篇里，我就曾说过VB的成功有一半的功劳要记到COM开发小组身上，COM可是M$公司打的一手好牌，从OLE到COM+，COM是近十年来M$最成功技术之一，所以有必要再吹它几句。

COM组件对象模型就是VB的基础，Varinat、String、Current、Date这些数据类型都是COM的，我们用的CStr、CInt、CSng等Cxxx函数根本就是COM开发小组写的，甚至我们在VB里用的数学函数，COM里都有对应的VarxxxDiv、VarxxxAdd，VarxxxAbs。嘿嘿，VB开发小组非常聪明。我们也可以说COM的成功也有VB开发小组和天下无数VB程序员的功劳，Bill大叔英明地将COM和VB捆绑在一起了。

所以说，学VB而不需要了解COM，你是幸福的，你享受着VB带给你的轻松写意，她把那些琐碎的家务事都干了，但同时你又是不幸的，因为你从来都不曾了解你爱的VB，若有一天VB对你发了脾气，你甚至不知该如何去安慰她。所以，本系列文章将拿出几大篇来教大家如何帮VB做点COM方面的家务事，以备不时之需。

想一口气学会所有COM家务事，不容易，今天我们先拿数组来开个头，更多的技术我以后再一一道来。

二、COM自动化里的SafeArray

就象洗衣机、电饭堡、吹尘器，VB洗衣服、做饭、打扫卫生都会用到COM自动化。它包含了一切COM里通用的东西，所有的女人都能用COM自动化来干家务，无论是犀利的VC、温柔的VB、还是小巧

的VBScript，她们都能用COM自动化，还能通过COM自动化闲话家常、交流感情。这是因为COM自动化提供了一种通用的数据结构和数据转换传递的方式。而VB的数据结构基本上就是COM自动化的数据结构，比如VB里的数组，在COM里叫做SafeArray。所以在VB里处理数组时我们要清楚的知道我们是在处理SafeArray，COM里的一种安全的数组。

准备下厨，来做一道数组指针排序的菜，在看主料SafeArray的真实结构这前，先让我们来了解一下C里的数组。

在C和C++里一个数组指针和数组第一个元素的指针是一回事，如对下： #include using namespace std; int main() { int a[10];

27

cout << \

cout << \ } ///:~

可以看到结果a和&a[0]是相同的，这里的数组是才数据结构里真实意义上的数组，它们在内存里一个接着一个存放，我们通过第一个元素就能访问随后的元素，我们可以称这样的数组为\真数组\。但是它不安全，因为我们无法从这种真数组的指针上得知数组的维数、元素个数等非常重要的信息，所以也无法控制对这种数组的访问。我们可以在C里将一个二维数组当做一维数组来处理，我们还可以通过一个超过数组大小的索引去访问数组外的内存，但这些都是极不安全的，数组边界错误可以说是C里一个非常容易犯却不易发现的错误。

因此就有了COM里的SafeArray安全数组来解决这个问题，在VB里我们传递一个数组时，传递的实际上COM里的SafeAraay结构指构的指针，SafeAraay结构样子如下： Private Type SAFEARRAY

cDims As Integer '这个数组有几维？ fFeatures As Integer '这个数组有什么特性？

cbElements As Long '数组的每个元素有多大？ cLocks As Long '这个数组被锁定过几次？

pvData As Long '这个数组里的数据放在什么地方？ 'rgsabound() As SFArrayBOUND End Type

紧接在pvData这后的rgsabound是个真数组，所以不能在上面的结构里用VB数组来声明，记住，在VB里的数组都是SafeArray，在VB里没有声明真数组的方法。 不过这不是问题，因为上面SFArrayBOUND结构的真数组在整个SAFEARRAY结构的位置是不变的，总是在最后，我们可以用指针来访问它。SFArrayBOUND数组的元素个数有cDims个，每一个元素记录着一个数组维数的信息，下面看看它的样子： Private Type SAFEARRAYBOUND

cElements As Long '这一维有多少个元素？ lLbound As Long '它的索引从几开始？ End Type

还有一个东西没说清，那就是上面SAFEARRAY结构里的fFeatures，它是一组标志位来表示数组有那些待性，这些特性的标志并不需要仔细的了解，本文用不上这些，后面的文章用到它们时我会再来解释。

看了上面的东西，各位一定很头大，好在本文的还用不了这么多东西，看完本文你就知道其实SafeArray也不难理解。先来看看如下的声明： Dim MyArr(1 To 8, 2 To 10) As Long

这个数组做为SafeArray在内存里是什么样子呢？如图一： cDims = 2 fFeatures =

FADF_AUTO AND FADF_FIXEDSIZE 位置 0

cbElements = 4 LenB(Long) 4 cLocks = 0 8

pvData（指向真数组） 12

28

rgsabound(0).cElements = 8 16 rgsabound(0).lLbound = 1 18 rgsabound(1).cElements = 9 22 rgsabound(1).lLbound = 2 26

cDims = 2

fFeatures =

FADF_AUTO AND FADF_FIXEDSIZE

位置 0

cbElements = 4 LenB(Long) 4

cLocks = 0 8

pvData（指向真数组） 12

29

rgsabound(0).cElements = 8 16

rgsabound(0).lLbound = 1 18

rgsabound(1).cElements = 9 22

rgsabound(1).lLbound = 2 26

图一：SafeArray内存结构

cDims表示它是个2维数组，sFeatures表示它是一个在堆栈里分配的固定大小的数组，cbElements表示它的每个元素大小是Long四个字节，pvData指向真的数组（就是上面说的C里的数组），rgsabound这个真数组表明数组二个维的大小和每个维的索引开始位置值。先来看看从这个上面我们能做些什么，比如要得到一个数组的维数，在VB里没有直接提供这样的方法，有一个变通的方法是通过错误捕获如下： On Error Goto BoundsError

For I = 1 To 1000 '不会有这么多维数的数组 lTemp = LBound(MyArr, I) Next

30

BoundErro:

nDims = I - 1

MsgBox \这个数组有\维\

现在我们知道了SafeArray的原理，所以也可以直接得到维数，如下：'先得到一个指向SafeArray结构的指针的指针，原理是什么，我后面说。 ppMyArr = VarPtrArray(MyArr)

'从这个指针的指针得到SafeArray结构的指针 CopyMemory pMyArr, ByVal ppMyArr, 4

'再从这个指针所指地址的头两个字节取出cDims CopyMemory nDims, ByVal pMyArr, 2 MsgBox \这个数组有\维\

怎么样，是不是也明白了LBound实际上是SafeArray里的rgsabound的lLbound，而UBound实际上等于lLbound +cElements - 1，现在我提个问，下面iUBound应该等于几？ Dim aEmptyArray() As Long

iUBound = UBound(aEmptyArray)

正确的答案是-1，不奇怪，lLbound -cElements - 1 = 0 - 0 - 1 = -1

所以检查UBound是不是等于-1是一个判断数组是不是空数组的好办法。

还有SafeArray结构里的pvData指向存放实际数据的真数组，它实际就是一个指向真数组第一个元素的指针，也就是说有如下的等式： pvDate = VarPtr(MyArr(0))

在上一篇文章里，我们传给排序函数的是数组第一个元素的地址VarPtr(xxxx(0))，也就是说我们传的是真数组，我们可以直接在真数组上进行数据的移动、传递。但是要如何得到一个数组SafeArray结构的指针呢？你应该注意到我上面所用的VarPtrArray，它的声明如下： Declare Function VarPtrArray Lib \ _ Alias \

它就是VarPtr，只不过参数声明上用的是VB数组，这时它返回来的就是一个指向数组SafeArray结构的指针的指针。因为VarPtr会将传给它的参数的地址返回，而用ByRef传给它一个VB数组，如前面所说，实际上传递的是一个SafeArray结构的指针，这时VarPtrArray将返回这个指针的指针。所以要访问到SafeArray结构需要，如下三步：

用VarPtrArray返回ppSA，再通过ppSA得到它指向的pSA，pSA才是指向SafeArray结构的指针，我们访问SafeArray结构需要用到就是这个pSA指针。

现在你应该已经了解了SafeArray大概的样子，就这么一点知识，我们就能在VB里对数组进行HACK了。

三、HACK数组字串指针

这已经是第三篇讲指针的东西了，我总在说指针能够让我们怎么样怎么样，不过你是不是觉得除了我说过的几个用途外，你也没觉得它有什么用，其实这是因为我和大家一样急于求成。在讲下去之前，我再来理一理VB里指针应该在什么情况下用。

只对指针类型用指针！废话？我的意思是说，象Integer, Long, Double这样的数值类型它们的数据直接存在变量里，VB处理它们并不慢，没有HACK的必要。但是字串，以及包括字串、数组、对象、结构的Variant，还有包括字串、对象结构的数组它们都是指针，实

31

际数据不放在变量里，变量放的是指针，由于VB不直接支持指针，对他们的操作必须连同数据拷贝一起进行。有时我们并不想赋值，我们只想交换它们指针，或者想让多个指针指向同一个数据，让多个变量对同一处内存操作，要达到这样的目的，在VB里不HACK是不行的。

对数组尤其如此，比如我们今天要做的菜：对一个字串数组进行排序。我们知道，对字串数组进行排序很大一部分时间都用来交换字串元素，在VB里对字串赋值时要先将原字串释放掉，再新建一个字串，再将源字串拷贝过来，非常耗时。用COM里的概念来说，比如字串a、b的操作a=b，要先用SysFreeString(a)释放掉原来的字串a, 再用a = SysAllocString(b)新建和拷贝字串，明白了这一点就知道，在交换字串时不要用赋值的方式去交换，而应该直接去交换字串指针，我在指针葵花宝典第一篇里介绍过这种交换字串的方法，这可以大大提高交换字串的速度。但是这种交换至少也要用两次CopyMemory来将指针写回去，对多个字串进行交换时调用CopyMemory的次数程几何增长，效率有很大的损失。而实际上，指针只是32位整数而已，在C里交换两个指针，只需要进行三次Long型整数赋值就行了。所以我们要想想我们能不能将字串数组里所有字串指针拿出来放到一个Long型指针数组里，我们只交换这个Long型数组里的元素，也就相当于交换了字串指针，排好序后，再将这个Long型指针数组重新写回到字串数组的所有字串指针里，而避免了多次使用CopyMemory来一次次两两交换字串指针。这样我们所有的交换操作都是对一个Long型数组来进行，要知道交换两个Long型整数，在VB里和在C里是一样快的。 现在我们的问题成了如何一次性地将字串数组里的字串指针拿出来，又如何将调整后的字串指针数组写回去。

不用动数组的SafeArray结构，我们用StrPtr也能完成它。我们知道，字串数组元素里放的是实际上是字串指针，也就是BSTR指针，把这些指针放到一个Long型数组里很简单，用下面的方法：

Private Sub GetStrPtrs()

Dim Hi As Long, Lo As Long Hi = UBound(MyArr) Lo = LBound(MyArr)

ReDim lStrPtrs(0 To 1, Lo To Hi) As Long Dim i As Long For i = Lo To Hi

lStrPtrs(0, i) = StrPtr(MyArr(i)) 'BSTR指针数组 lStrPtrs(1, i) = i '原数组索引| Next End Sub

为什么要用2维数组，这是排序的需要，因为当我们交换lStrPtrs里的Long型指针时，原来的字串数组MyArr里的字串指针并没有同时交换，所以用lStrPtrs里的Long型指针访问字串时，必须通过原来的索引，因此必须用2维数组同时记录下每个Long型指针所指字串在原字串数组里的索引。如果只用1维数组，访问字串时就又要用到CopyMemory了，比如访问lStrPtrs第三个元素所指的字串，得用如下方法： CopyMemory ByVal VarPtr(StrTemp), lStrPtrs(3), 4

虽然只要我们保证StrTemp足够大，再加上一些清理善后的工作，这种做法是可以的，但实际上我们也看到这样还是得多次调用CopyMemory，实际上考虑到原来的字串数组MyArr一直就没变，我们能够通过索引来访问字串，上面同样的功能现在就成了： StrTemp = MyArr(lStrPtrs(1,3)) '通过原字串数组索引读出字串。

32

不过，当我们交换lStrPtrs里的两个Long型指针元素时，还要记得同时交换它们的索引，比如交换第0个和第3个元素，如下：

lTemp1 = lStrPtrs(0, 3) : lTemp2 = lStrPtrs(1, 3)

lStrPtrs(0, 3) = lStrPtrs(0, 0) : lStrPtrs(1, 3) = lStrPtrs(1, 0) lStrPtrs(0, 0) = lTemp1 : lStrPtrs(1, 0) = lTemp2

当我们排好序后，我们还要将这个lStrPtrs里的指针元素写回去，如下： For i = Lo To Hi

CopyMemory(ByVal VarPtr(MyArr(i)), lStrPtrs(0,i), 4) Next

我已经不想再把这个方法讲下去，虽然它肯定可行，并且也肯定比用CopyMemory来移动数据要快，因为我们实际上移动的仅仅是Long型的指针元素。但我心里已经知道下面有更好更直接的方法，这种转弯抹角的曲线救国实在不值得浪费文字。

四、HACK数组的BSTR结构，实时处理指针。

最精采的来了，实时处理指针动态交换数据，好一个响亮的说法。

我们看到，上一节中所述方法的不足在于我们的Long型指针数组里的指针是独立的，它没有和字串数组里的字串指针联系在一起，要是能联系在一起，我们就能在交换Long型指针的同时，实时地交换字串元素。 这可能吗？

当然，否则我花那么笔墨去写SafeArray干什么！

在上一节，我们的目的是要把字串数组里的BSTR指针数组拿出来放到一个Long型数组里，而在这一节我们的目的是要让我们Long型指针数组就是字串数组里的BSTR指针数组。拿出来再放回去的方法，我们在上一节看到了，现在我们来看看，不拿出来而直接用的方法。

这个方法还是要从字串数组的SafeArray结构来分析，我们已经知道SafeArray结构里的pvData指向的就是一个放实际数据的真数组，而一个字串数组如MyArr它的pvData指向的是一个包含BSTR指针的真数组。现在让我们想想，如果我们将一个Long型数组lStrPtrs的pvData弄得和字串数组MyArr的pvData一样时会怎样？BSTR指针数组就可以通过Long型数组来访问了，先看如何用代码来实现这一点： '模块级变量

Private MyArr() As String '要排序的字串数组

Private lStrPtrs() As Long '上面数组的字串指针数组，后面会凭空构造它 Private pSA As Long '保存lStrPtrs数组的SafeArray结构指针 Private pvDataOld As Long '保存lStrPtrs数组的SafeArray结构的原 ' pvData指针，以便恢复lStrPtrs

'功能： 将Long型数组lStrPtrs的pvData设成字串数组MyArr的pvData ' 以使Long指针数组的变更能实时反应到字串数组里 Private Sub SetupStrPtrs() Dim pvData As Long

' 初始化lStrPtrs，不需要将数组设得和MyArr一样大

33

' 我们会在后面构造它 ReDim lStrPtrs(0) As Long

'得到字串数组的pvData pvData = VarPtr(MyArr(0))

'得到lStrPtrs数组的SafeArray结构指针

CopyMemory pSA, ByVal VarPtrArray(lStrPtrs), 4

'这个指针偏移12个字节后就是pvData指针，将这个指针保存到pvDataOld ' 以便最后还原lStrPtrs，此处也可以用： ' pvDataOld = VarPtr(lStrPtrs(0)) CopyMemory pvDataOld, ByVal pSA + 12, 4

'将MyArr的pvData写到lStrPtrs的pvData里去 CopyMemory ByVal pSA + 12, pvData, 4

'完整构造SafeArray必须要构造它的rgsabound(0).cElements

CopyMemory ByVal pSA + 16, UBound(MyArr) - LBound(MyArr) + 1, 4 ' 还有rgsabound(0).lLbound

CopyMemory ByVal pSA + 20, LBound(MyArr), 4 End Sub

看不懂，请结合图一再看看，应该可以看出我们是凭空构造了一个lStrPtrs，使它几乎和MyArr一模一样，唯一的不同就是它们的类型不同。MyArr字串数组里的fFeatures包含FADF_BSTR，而lStrPtrs的fFeatures包含FADF_HAVEVARTYPE，并且它的VARTYPE是VT_I4。不用关心这儿，我们只要知道lStrPtrs和MyArr它们指向同一个真数组，管他是BSTR还是VT_I4，我们把真数组里的元素当成指针来使就行了。

注意，由于lStrPtrs是我们经过了我们很大的改造，所以当程序结束前，我们应该将它还原，以便于VB来释放资源。是的，不释放也不一定会引起问题，因为程序运行结束后，操作系统的确是会回收我们在堆栈里分配了却没有释放的lStrPtrs原来的野指针pvOldData，但当你在IDE中运行时，你有60%的机会让VB的IDE死掉。我们是想帮VB做点家务事，而不是想给VB添乱子，所以请记住在做完菜后，一定要把厨房打扫干净，东西该还原的一定要还原。下面看看怎么样来还原： '还原我们做过手脚的lStrPtr Private Sub CleanUpStrPtrs()

'lStrPtr的原来声明为：ReDim lStrPtrs(0) As Long ' 按声明的要求还原它

CopyMemory pSA, ByVal VarPtrArray(lStrPtrs), 4 CopyMemory ByVal pSA + 12, pvDataOld, 4 CopyMemory ByVal pSA + 16, 1, 4 CopyMemory ByVal pSA + 20, 0, 4 End Sub

好了，精华已经讲完了，如果你还有点想不通，看看下面的实验：

34

'实验

Sub Main()

'初始化字串数组 Call InitArray(6)

'改造lStrPtrs Call SetupStrPtrs

'下面说明两个指针是一样的 Debug.Print lStrPtrs(3)

Debug.Print StrPtr(MyArr(3)) Debug.Print

'先看看原来的字串 Debug.Print MyArr(0) Debug.Print MyArr(3) Debug.Print

'现在来交换第0个和第3个字串 Dim lTmp As Long lTmp = lStrPtrs(3) lStrPtrs(3) = lStrPtrs(0) lStrPtrs(0) = lTmp

' 再来看看我们的字串，是不是觉得很神奇 Debug.Print MyArr(0) Debug.Print MyArr(3) Debug.Print

'还原

Call CleanUpStrPtrs End Sub

在我的机器上，运行结果如下： 1887420 1887420

OPIIU

WCYKOTC

WCYKOTC OPIIU

怎么样？如愿已偿！字串通过交换Long型数被实时交换了。 通过这种方式来实现字串数组排序就非常快了，其效率上的提高是惊人的，对冒泡排序这样交换字串次数很多的排序方式，其平均性能能提高一倍以上（要看我们字串平均长度，），对快速排序这样交换次数较少的方法也能有不少性能上的提高，用这种技术实现的快速排

35

序，可以看看本文的配套代码中的QSortPointers。

本道菜最关键的技术已经讲了，至于怎么做完这道菜，怎么把这道菜做得更好，还需要大家自己来实践。

四、我们学到了什么。

仅就SafeArray来说，你可能已经发现我根本就没有直接去用我定义了的SAFEARRAY结构，我也没有展开讲它，实际上对SafeArray我们还可以做很多工作，还有很多巧妙的应用。还有需要注意的，VarPtrArray不能用来返回字串数组和Variant数组的SafeArray结构的指针的指针，为什么会这样和怎样来解决这个问题？这些都需要我们了解BSTR，了解VARIANT，了解VARTYPE，这些也只是COM的冰山一角，要学好VB，乃至整个软件开发技术，COM还有很多很多东西需要学习，我也还在学，在我开始COM的专题之前，大家也应该自学一下。

COM的东西先放一放，下一篇文章，应朋友的要求，我准备来写写内存共享。

后记：

又花了整整一天的时间，希望写的东西有价值，觉得有用就来叫个好吧！

36

真想不到之五：高效字串指针类

关键字：VB、HCAK、字串指针、BSTR、效率、内存共享 难度：中级或高级 参考文章：

1、2000年7月VBPJ Black Belt专栏文章《Modify a Varialbe's Pointer》 作者：Bill McCarthy

2、1998年4月VBPJ Black Belt专栏文章《Play VB's Strings》 作者：Francesco Balena

引言：

本想以内存共享做为VB指针专题的最后一篇，写着写着发现字串的问题应该单独谈谈。在内存共享的问题上，我尤其关心的是字串的共享，因为在我一个多月前发布的源码里用的是《HardCore VB》里Bruce Mckinney提供的CShareStr类，它实现了字串的内存共享。但是Bruce也没有突破局限，对字串的处理依然是CopyMemory的乾坤大挪移，尤其是还要进行讨厌的ANSI/DBCS和Unicode的转换。我在readme里说过它效率极低，应该采用Variant或Byte数组来实现，才能避免转换。后来又想到可以用StrPtr来做，并在VC里用DLL共享节实现了可以不进行转换的字串内存共享。不过在VC里我仍然需要用SysAllocString来建立VB能使用的BSTR。这都不是我想要的，我想要的东西要象VC里的CString的一样，只要字串够大，对其赋值就不用重新分配内存，还要象VC里CComBSTR类一样可以Attach到一个特定BSTR。

知道该怎么做，是在看了VBPJ上Bill McCarthy和Francesco Balena的两篇文章之后。Bill用修改SafeArray描述结构实现了数组的内存共享，而Francesco则对字串指针进行深入的探讨。但是Bill和Francesco的东西都没有实现我想要的字串类。

方法知道了，实现并不难，所以我决定自己来包装一个这样的东西。

正文：

使用VB里的字串类型String有两大不足：第一、它的分配是由VB运行时控制，我们不能将其分配在指定内存处；第二，任何一次对字串的赋值操作都要进行内存重新分配。要实现高效、灵活的字串处理，我们必须克服这两大不足。

对于第一个问题，通过修改String变量里放着的BSTR描述符指针可以实现；对于第二个问题，可以用Mid语句（注意是语句而不是函数）来赋值。不详细讲了，直接看下面的这个类：

```
Option Explicit
'******************************************************** 'clsBSTR.cls
'作者: 熊超 ID: AdamBear 2002年3月18日 'http://www.csdn.net/Author/AdamBear
' 你可以自由使用本类模块，不过请保留本声明
'********************************************************
37
Private Declare Sub CopyMemory Lib \Any, Source As Any, ByVal Length As Long)
'不要直接对sString赋值(可以用MID语句)，将其设为公有仅为提高效率。 Public sString As String 'BSTR描述符指针
Private pStr As Long 'BSTR地址
Private nMaxLen As Long 'BSTR最大字节数
'让本字串指向特定地址
Public Sub Attach(Addr As Long, Optional nLen As Long) pStr = Addr
'修改BSTR描述符指针，使其指向Addr CopyMemory ByVal VarPtr(sString), Addr, 4
If IsMissing(nLen) Then Exit Sub '设定最大字串字节数 nMaxLen = nLen End Sub
'还原本字串原BSTR描述符 Public Sub Detach()
CopyMemory ByVal VarPtr(sString), 0&, 4 End Sub
'让本字串指向源字串
Public Sub AttachStr(sStr As String) Attach StrPtr(sStr), LenB(sStr) End Sub
'data为缺省属性
Public Property Let data(sVal As String) Dim c As Long c = LenB(sVal)
'超过最大字串数，抛出错误。
If c > nMaxLen Then Err.Raise vbObjectError + 3000, _ \溢出\ '写字串长度
CopyMemory ByVal (pStr - 4), c, 4 '写字串
Mid(sString, 1) = sVal
38
End Property
'可以通过公有变量sString来读字串，效率更高 Public Property Get data() As String data = sString End Property
Private Sub Class_Terminate() Call Detach End Sub
```

用法如下，假设我们已通过VitualAlloc，HeapAlloc，MapViewOfFile这样的内存管理API得到了一个4k个字节的可读写的内存地址baseAddr： Dim sShare As New clsBSTR

```
'留下前4个字节用于BSTR保存字串字节数 sShare.Attach(baseAddr+4, 4096-4)
'下面的字串\会直接写到baseAddr+4字节处 sShare = \ Dim y As String
'读字串时可以用sString属性或缺省属性 y = sShare.sString
```

'用AttachStr方法Attach到一个字串。 '必须要先Detach sShare.Detach

sShare.AttachStr(y) sShare = \ Debug.Print y

'一旦AttachStr到字串y后，对sShare的修改就相当于对y的修改。 '并且以后对y的修改也只能用Mid语句 Mid(y, 1) = \

'不能直接赋值，这样VB会将原来y所指（也是sShare所指）内存释放， ' 重新分配y。这样在访问sShare时会出错。 'y = \

我也不在这里讲这个类的详细原理，可以参考我前面说的两篇文章。 使用这个类有几个需要注意的地方。

1、读字串时可以用sString属性来读，更快。

读sShare有两种方法，一种是用缺省属性Data来读，一种是直接用sString属性来读。用sString属性不重新分配内存，要快得多。

2、不要直接给sString赋值，应使用缺省的data属性来赋值。

之所以把sString属性暴露出来，是为了效率和方便。我们可以用Mid语句对其进行修改，但不要直接用\来赋值。 3、注意Attach的第二个参数，表示字串的最大字节数，不要让它超过已经分配的内存。

39

4、用AttachStr将本字串对象Attach到某个字串（比如上面的y）上后，不能再对这个字串y重新赋值，也不能将其传递到会对其重新赋值的过程。

哇，这么多需要注意的问题，用起来岂不是更不方便。的确，用它的之前要考虑是不是必须的。因为建立这个类也一样有开销。所以还有一个需要注意的问题：

5、它主要的应用还是在于将字串安放在指定内存处。虽然它也可以让同一个进程内几个的字串达到共享的目的，但是如果只是两三个很小的字串这样时做反而慢了。

后计：

数组指针和字串指针我们已经谈过了，对于普通的数值类型变量的指针没有什么Hack的必要，但是它关系到一个有用的技术，下篇文章再谈。

本文和下篇文章的代码，以及用这个类来实现的共享内存的代码，我会发布到CSDN共享软件上，名字是《内存共享和指针》。

40

真想不到之六：有用的技术和没用的指针 关键字：VB、SafeArray、数值类型指针 难度：中级 参考文章：

1、2000年7月VBPJ Black Belt专栏文章《Modify a Varialbe's Pointer》 作者：Bill McCarthy

引言：

这真的是指针专题的最后一篇了（当然，以后肯定还会提到指针）。主要是来谈谈Bill McCarthy的文章《Modify a Varialbe's Pointer》的精华。关于这篇文章的东西，在我的《VB指针葵花宝典之SafeArray》里曾谈到过，但那篇文章实际上没有写出SafeArray的精华，用SafeArray最妙的地方在于可以将一个变量建在指定的内存处，就象上一篇文章给出的那个字串类一样。

正文：

Bill McCarthy在那篇《Modify a Varialbe's Pointer》里用SafeArray实现多进程的数组共享内存，他考虑了数组变量的类型，因此可以兼容大部分数值类型的数组，是一个非常不错的东西。我这里不讲它实现的具体方法，只是想和大家一起看看SafeArray还能做什么。 修改SafeArray结构的pvData指针却是一个非常有用的技术，通过修改pvData，就能够通过数组直接访问指定的内存。

和上一篇文章包装字串指针类一样，通过修改pvData，我们也可以包装一些普通数值类型变量的指针类。

我在指针的第一篇文章里说过，要想实现C语言里'*'这个取指针所指变量值功能，必须要用CopyMemory。实际上，我说错了，我们完全可以实现和C里一样的指针，如下： //C语言

Long L;

Long* pL = &L; *pL = 12;

printf(\ *pL = %d\

'VB里

Dim pL As New pLong, L As Long pL.Attach L

'也可以 pL.Ptr = VarPtr(L) pL = 12

Debug.Print \ *pL =\

结果都能够通过修改pL指针，达到修改变量L的目的。

上面VB代码里的pLong就是一个包装好的Long型变量的指针类，下面看看如何来实现它：

Option Explicit

'** 'pLong.cls

41

'包装一个Long型指针的类

'作者: 熊超 ID: AdamBear 2002年3月18日 'http://www.csdn.net/Author/AdamBear

' 你可以自由使用本类模块，不过请保留本声明

'**

Private Declare Sub CopyMemory Lib \Any, Source As Any, ByVal Length As Long)

Private m_Arr(0) As Long

'缺省属性

Public Property Get Data() As Long Data = m_Arr(0) End Property

Public Property Let Data(ByVal Value As Long) m_Arr(0) = Value End Property

Public Sub Attach(Target As Long) Ptr = VarPtr(Target) End Sub

Public Property Let Ptr(ByVal Target As Long) Dim pSA As Long

'得到SafeArray结构指针pSA

CopyMemory pSA, ByVal VarPtrArray(m_Arr), 4 '这个指针偏移12个字节后就是pvData指针 CopyMemory ByVal (pSA + 12), Target, 4

End Property

Public Property Get Ptr() As Long Ptr = m_SA.pvData End Property

Private Sub Class_Terminate()

CopyMemory ByVal VarPtrArray(m_Arr), 0&, 4 End Sub

42

要将它改成Byte的指针类，只需要将上面的代码中m_Arr数组的类型，Data属性和Attach方法中的参数类型改为Byte型即可。

当我们这样做出pLong、pByte、pInteger后，我们就能够玩点和C里一样的花样了。 Sub Main()

Dim pB As New pByte, B As Byte Dim pI As New pInteger, I As Integer Dim pL As New pLong, L As Long

'用Attach方法将经过类型检查，直接用Ptr属性则可以绕过类型检查 pB.Attach B pI.Attach I pL.Attach L

'试试指针 B = 1

Debug.Print \ *pB =\

pB = 1

Debug.Print \ *pB =\

I = 1000

Debug.Print \ *pI =\

pI = 2000

Debug.Print \ *pI =\

L = 40000

Debug.Print \ *pL =\

pL = 60000

Debug.Print \ *pL =\

'试试C里的类型转换

'用Integer指针访问Long型变量 pI.Ptr = VarPtr(L)

Debug.Print \ End Sub

搞出这几种普通数值类型的指针类有什么用？基本上没有什么大用。不过是证明一种方法的可行性，和演示技术。这种技术还有什么用，需要的时候还会再谈。

后记：

43

本文的东西，可见CSDN共享软件上的《内存共享和指针》，

指针的专题就到这儿了，下一篇准备开始着手写VB和COM的一个系列文章，其间我准备翻译一下《VB Design Patterns》，这是一本不错的书。

44

Matthew Curland简介：

Visual Studio开发小组成员，参与开发了VB的IntelliSense和Object Browser。他是VB资深专家，对VB有非常深入的研究，堪称VB大师。所著《Advanced Visual Basice》是阐述VB高级编程技巧的一本好书。 本文英文原著可见2000年2月份《Visual Basic Programmer's Journal》（VB程序员月刊）里的《Call Function Pointers》，这是他发表的妙文之一，他的书里的第11章和本文同名，本文应该是这一章节的精华。

之所以推荐此文，是因为它综合运用了VB里的不少技术。我们可从中看到Matt大师对VB的深刻理解，而各位技术的综合运用正体现了他深厚的功力。

本 文 原 文 ： http://www.devx.com/premier/mgznarch/vbpj/2000/02feb00/mc0200/mc0200.asp (要先注册成premier用户) 本文配套代码：

http://www.devx.com/free/mgznarch/vbpj/code/2000/02feb00/vb0002mc_p.zip

关键字：函数指针，COM、对象、接口，vTalbe，VB汇编，动态DLL调用。 级别：高级

要求：了解VB对象编程，了解汇编。

调用函数指针 通过使用函数指针，我们能够动态地在代码中插入不同行为的函数，从而使代码拥有动态改变自身行为的能力。

作者：Matther Curland

要求：使用本文的示例代码，你需要VB5或VB6的专业版或企业版。

从Visual Basic 5.0开始Basic语言引入了一个重要的特性：AddressOf运算符。这个运算符能够让VB程序员直接体会到将自己的函数指针送出去的快感。比如我们在VB里就能够得到系统字体的列表，我们能够通过标准的API调用来进行子类化。一句话，我们终于可以象文档里所说的那样来使用Win32 API了。

不过，这个新玩具只能给我们带来短暂的快感，因为这个礼物并不完整。我们可以送出函数指针，但却没人能将函数指针送给我们。事实上，我们甚至不能给我们自己送函数指针，这使我们不能够体验送礼的真正乐趣（译者：呵呵，光送礼却不能收礼的确没趣）。AddressOf让我们看到了广袤天地的一角，但是VB却不让我们全面地探索它，因为VB根本就不让我们调用函数指针，我们只能提供函数指针（译者：可以先将函数指针送给API，然后让API回调自己的函数指针来完成函数指针调用的功能，但这还是要先把礼物送给别人）。其实，我们能够自己来实现调用函数指针的功能，我们可以手工将一个对COM接口的vTable绑定调用变成一个函数指针调用。最妙的是：我们能够在纯VB里写出调用函数指针的代码，不需要任何辅助的DLL。

告诉编译器函数指针是什么样子，是使VB能够调用任何函数的关键。将参数类型和返回值类型交给VB编译器，让编译器将我们的函数调用编译到我们的程序里，这样程序才能在运行时知道怎样去定位函数。在程序被编译后，一个函数就是内存里一串汇编字节流，通

45

过CPU解释执行而形成我们的程序。调用一个函数指针，首先需要程序获得指向这个函数字节流的指针，再通过x86汇编指令call将当前指令指针(译注：即x86汇编里的IP寄存器)转到函数所在的字节流上。在函数完成后，再用ret指令返回给调用此函数的程序来继续操作。

我下面将要提到的方法，利用了VB自己的函数调用方式，所以我先来解释一下VB是怎样来实现函数调用的。VB内部使用三种函数指针，但是，在本质上，不论VB是如何来定位这几类函数指针，调用它们的方法却是一样的。VB编译器必须知道准确的函数原型才能生成调用函数的代码。

第一类，最常见的函数指针类型，就是VB用来调用函数的普通指针，这样的函数定义在标准模块内（或类模块里的友元函数和私有函数）。调用友元函数和私有函数时，调用指令定位在当前指令指针的一个偏移地址处，或者先跳到一个记录着函数位置的查找表里，再跳到函数内（译者：即先\绝对地址\跳到一个跳转表内，表里的每个入口都是一个\到函数）。这些函数都在同一个工程内，联结器总是将所有的模块联结在一起，所以总是知道在内存何处能够找到VB内部函数，因此转移控制到内部函数时，其运行时开销是很少的。

VB对某些函数指针的调用却困难得多

对于另两类函数指针，VB必须在运行时进行额外的工作才能够找出它们。

第二类，VB调用一个COM对象接口里的方法。我们可能认为建立COM对象的工作是相当复杂的，如果完全用VB来为我们建造COM的所有组成部分的话，但事实上并不是这样。按照COM的二进制标准，一个COM对象是一个指针，这个指针指向一个结构，这个特定结构的第一个元素是一个指向函数指针数组的指针。这个函数指针数组（又叫虚拟函数表，简称vTable）里的前三个指针，一定是标准QueryInterface，AddRef，Release函数。vTable里接下来的函数符合给定的COM对象接口定义里的函数定义（见图一）

图一：

函数指针代理是怎么工作的？click here

当VB通过一个对象类型的变量来调用一个COM对象的方法或属性时，这个变量里存放着对这个COM对象接口的引用。VB要定位函数时，首先要通过COM引用的第一个元素来获得指向vTalbe的指针，然后才能在vTable里定位函数指针。对一个vTable调用来说，编译器提供了COM引用和函数指针在vTable里的偏移量。这样函数指针才能在运行时被动态地选出来。这种双向间接的方式——两种指针都必须被计算（译注：指向vTalbe的指针和vTable里的函数指针都必须在运行时才确定）——使得vTable调用比同一个工程内的直接调用慢得多，因为直接调用不需要任何在运行时才能进行的指针间接指定。

VB对待同一个工程里的类的公有方法和对待外部COM对象里方法完全一样，都需要查找vTable，这就是为什么在同一个对象内调用一个友元函数会比调用一个公有函数快得多的原因。但是，查找vTable是COM的基础，它使得VB能够使用从外部库里载入的COM对象，也是象Implements这样的编程概念的实现基础。动态载入不可能通过静态联结来实

46

现，查找vTable的花费是使用动态载入必须付出的代价。

通过Object型变量来进行的后期绑定调用不同于vTable绑定调用。当然，这种差别不在于VB用没用vTable，这种差别是因为对后期绑定调用VB使用了不同的vTable。当进行后期绑定调用时，编译器会调用IDispatch接口的GetIDsOfNemes和Invoke。这需要两次vTable调用和相当多的参数传递，所以这样的处理非常慢，而且必须不断地定位Invoke，才能通过类型信息调用到真正的函数指针（译者：真正慢的原因还是Invoke所进行的参数调整。当拥有相应对象的接口类型库信息时，VB会进行另一种后期绑定——DispID绑定，它只需要在第一次访问对象时调用GetIDsOfNemes，来获得所有属性和方法的DispID，以后的调用只需要对Invoke进行一次vTalbe调用，但由于Invoke才是慢的原因，所以DispID绑定比一般后期绑定快不了多少）。毋庸置疑，当在同一个线程里调用COM对象时，后期绑定将比vTalbe绑定慢几个数量级（译者：同线程内要慢数百倍。由于跨边界的调配开销，随跨线程、跨进程、跨机器，两种绑定方式在速度上的差别将越来越小）

第三类，通过Declare语句来使用函数指针。Declare使得VB能够动通过LoadLibraray API来动态载入特定的DLL，并通过GetProcAddress API和函数名（或函数别名）来得到DLL里特定的函数指针。声明在类型库里的函数指针是在程序装入时通过import table（输入表）来载入的，而通过Declare语句声明的函数指针是在此函数第一次被调用时装入（译者：这两种方式各有优缺点。使用Declare在调用时载入，一来VB运行时直接支持，使用简单，二来当需要载入的DLL不存在时可以在运行时通过错误捕获来处理。而使用类型库一次性载入，一是会增加载入时间，二是当相应的DLL找不到时程序根本就无法起动，但是通过类型库调用API可以绕过VB运行时动态的DLL载入过程，这在某些时候很有必要）。

动态指定函数指针

无论是Declare还是库型库，当函数载入后，VB调用函数指针的方式是一样的。指针已经因为先前的调用而被载入了，所以第二次调用会更快，并且速度接近调用静态联结的函数。Declare语句是VB调用动态载入的函数指针的最自然的方法。但是，函数指针由VB决定而不是由我们来指定（译者：此为原文直译，意思应该是：函数指针只能在编译前指定，由VB来载入，而不能在运行时指定由我们自己动态载入的函数指针），所以我们不能用Declare语句来调用任意的函数指针。Declare语句的限制使我们只能载入在设计时通过Lib和Alias字句指定的函数。

到这里，我已经解释了VB是怎么样来调用自己的函数指针的。对VB本身没有的功能进行扩展都应该通过VB本身提供的工具来实现（译者：看来作者Matt是一位VB纯粹论支持者）。静态联结不用考虑——如果你喜欢自己修改PE文件头的话，请自便（译者：关于修改PE头来Hook输入函数的方法，在1998年2月MSJ专栏Bugslayer里，John Robbins大师就用纯VB实现了HookImportedFunctionsByName，不过用来调用函数指针那是杀鸡用牛刀）。我们不可能静态地指定函数指针，所以Declare语句也不用考虑。但是，我们能够在VB里自己用LoadLibaray和GetProcAddress这两个API来从外部DLL里获取函数指针，就象Declare为我们做的那样。vTable调用

是唯一一种让VB自己绑定函数的调用方式。我们的任务是建一个符合COM二进制标准的结构，再将这个手工建立的COM对象的引用放到一个对象类型的变量里，然后调用手工建立的vTable入口。通过调用这个vTable里的函数，就能够直接代理到要调用的函数指针。我称这个对象为FunctionDelegotor（函数代理者）。

47

这个方法需要我们解决三个特有的问题。第一，vTalbe调用有额外的参数（this指针），我们不想将它也传给我们的函数指针。所以我们需要一个通用的代理函数来将这个额外的this指针处理掉，然后才能进行调用。第二，我们需要建立一个vTable里有这个代理函数的COM对象。第三，我们需要一个接口定义才能让VB编译器知道我们的函数指针的样子。接口定义应该将函数原型也包括在vTable里，并且和代理函数在对象vTable里的位置一样（译者：当通过接口调用函数指针时，只有这样才能够让代理函数处理掉做为函数参数压在栈里的this指针）。

我们可以用汇编代码很容易地的写出代理函数（译者：对作者Matt来说的确很容易，因为他对在VB里插入线内汇编代码有相当深入的研究。其实作者这里的容易也是相对于Alpha平台来说的）。在Intel平台，所有传递给COM对象或标准API调用的参数都是通过堆栈来传的。不幸的是，对Alpha平台的VB来说不是这样，它不能提供一种简单的方法来写出同样功能的汇编代码（译注：Alpha平台是一个RISC精简指令集系统，其参数传递多直接使用寄存器，要在这个平台上手工写汇编代码要难得，从他的书的目录里知道他在书里专门拿出一节介绍Alpha平台下的汇编代码）。 压栈

只要我们知道栈是什么样子，我们就可以很清楚的知道汇编代码需要做什么。VB仅仅支持符合stdcall调用规范的函数。这种调用规范，参数总是从右向左压入栈中，并且是由调用者来负责栈的清理。清理的义务跟本文没什么关系，但是压栈的顺序却很重要。尤其要注意的是COM类里的this指针（在VB类里称为Me），它总是作为最左边的参数压栈的。当函数被调用时，函数返回地址（函数返回后程序继续执行的地方）也被call指令本身压入栈中。在任何COM接口输出函数被执行前，栈的样子如下：

parameter n （第n个参数，最右边的参数） ...

parameter 2

parameter 1 （第1个参数）

this pointer（暗藏的this指针才是最左前的参数） return address （返回地址）

但是，我们只想调用函数指针，并不需要暗藏的相关联的this指针。调用一个符合vTable调用却没有额外参数的函数，需要我们将this指针从栈里挤出来，然后才能将控制转移到目标函数指针。让this指针在栈里放着的好处是因为它指向结构。考虑我们定义了一个结构，它的第二个成员是一个函数指针。这个成员距结构开始位置的偏移是4个字节。那么将这个函数指出挤出来并通过代理函数调用它的汇编代码如下：

;弹出返回地址到临时的ecx寄存器， ;后面还要将它恢复。 pop ecx

;从栈里弹掉this指针（译注：做为后面跳转的基址） pop eax

48

;重新将ecx寄存器里保存的返回地址压栈 ;以使得函数指针调用后知道返回到哪儿 push ecx

;将控制转移到函数指针，

;它在this指针后偏移4个字节处。 jmp DWORD PTR [eax + 4]

这四条指令的连在一起需要6个字节：59 58 51 FF 60 04。我们在后面补两个Int3指令（CC CC）以凑足8个字节，这正好可以一个VB的Currency变量内。这样一个Currency变量的地址里会放着如下的magic number（幻数）——368956918007638.6215@ ——这个Currency变量是指向代理函数的函数指针。这个代理函数挤掉this指针，并可跳到任何函数，而不用考虑函数的参数。这就是说，我们可

以用同样的汇编代码来代理任何函数指针。我们现在需要一个vTable来包含这个指向字节流的指针，它实际上是一个函数。（译者：即用vTable的某个入口包含代理函数指针）。

使用代理函数需要用到一个结构，它偏移4字节处是我们要调用的函数指针。我们还需要它偏移0个字节处是一个指向vTable的指针，这样才能让这个结构和一个COM对象一样，只有这样VB才能调用到vTable里的函数。我们并没必要为了一个简单的函数指针调用而在堆里分配内存；相反，我们仅需在调用代码的某个地方声明一个FunctionDelegator结构的变量。虽然我们提供了AddRef和Release函数，但它们不做任何事，只不过是迁就一下VB（译者：VB她对我们的对象引用进行严格的跟踪。每当我们新增一个对我们对象的引用，她就会调用一次我们对象里的AddRef，以准确计录对象被引用的次数；每当我们的一个引用和对象分手，她又会调用Release来通知我们的对象减少引用计数。VB她这样做是为了当我们所有的引用都和对象分手后，对象能够在内存里被干净地抛弃。为了迁就VB她的这个习惯，哪怕我们手工建立的对象并不动态分配内存，我们的对象也必须提供AddRef和Release）。所以第四个vTable入口是一个指向代理函数汇编代码的指针。函数代理的代码里声明了一个UDT来包含一个vTalbe数组指针。（代码见Listing1）

将结构转换成COM对象

当我们将一个指向合法vTable的指针传给FunctionDelegator结构，并将这个结构拷贝到一个对象变量里，这个结构就成为合法的COM对象了。这个对象的QueryInterface（译者：以下简称QI）函数相信我们所要求的接口vTalbe的第四个入口的函数原型总是和函数指针相符的。如果不支持所要求的接口，QI函数通常返回E_NOINTERFACE错误。这个错误状态在VB里表现出来就是在停在Set语句上的类型不符错误。FunctionDelegator对象的这种信任的设计要求我们必须自己来保证类型安全，我们永远不要向这个对象请求一个不符合函数指针原型的接口。如果我们破坏了这个规则，对我们的惩罚就将是崩溃而不是类型不匹配错误了（译者：要体会这种惩罚，可以试着将Listing1代码里的InitDelegator返回的接口用VB里的任意接口来引用，比如用Shape，由于其第四个接口定义不符，崩溃）。

FunctionDelegator的vTalbe不进行任何引用计数，所以我们不用编写任何tear-down（严重错误处理）或内存释放代码。当栈越出它的scope时（译者：此处的scope是指FunctionDelegator对象变量的变量范围，即声明和使用它的过程级或模块级范围），COM对象所使用的内存会自动从栈里清除，这意味着InitDelegator所返回的COM对象必然在结构

49

自己销毁之前（或同时）被销毁。

在VB能够调用到代理函数之前，还有一个步骤：我们必须为我们想要调用的函数指针定义一个接口。通过使用mktylib工具来生成对象定义语言（ODL）文件，我们能够非常容易地做到这一点。尽管mktylib.exe是midl.exe的一个官方的功能简化版本，但当我们要生成给VB使用的严格的类型库时，mktylib.exe相对更容易使用。而且，不同于midl.exe，mktylib.exe它是和单独的VB产品一起销售的。我们的接口定义必须继承自IUnknown并且有一个附加的函数。当我们仅仅使用ODL待性而不使用oleautomation特性时，我们能够避免OLE自动化在注册表里的HKCR\\Interface主键下写入不必要的注册键值。虽然我们的QI函数忽略uuid，但是它还是需要我们建立类型库。（译者：虽然可以通过ActiveX工程来生成包含类型库的组件，这样可以不用外部工具就能生成类型库，但是VB里所有的组件都是支持OLE自动化的，它们必须在注册表里注册键值。更重要的是，VB所生成的接口都继承自IDispatch，其vTable并不符合本文的要求。如果不想使用对象定义语言，而想用更纯的VB地来做，就必须修改代理函数的实现，因为继承至IDispatch后，我们只能在vTable的第八个入口里放代理函数指针。虽然这种做法可行，但是实现起来很复杂，因为需要手工建立能迁就VB的IDispatch，而这决不象本文手工建立 IUnknown接口这么简单。虽然可能，但这个弯子绕得太大了）

作为例子，这里定义了三种函数。第一种是在排序算法中回调的标准的比较函数原型。第二种函数指针调用能够返回COM HRESULT错误代码，比如DllRegisterServer。第三种是一个即没有参数也没有返回值的函数。我们可以按照自己的需要来加入函数声明。保存经过我们修改的FuncDecl.odl文件，并且执行mktylib FuncDecl.odl，然后再将FuncDecl.tlb的引用加入我们的工程。（见Listing2里的ODL）

我们能够看到，通过调用下面的一对函数，我们的确是可以实时调用函数指针了，而很长时间以来，对VB程序员来说，想使用这对函数是不可能的，这对函数就是DllRegisterServer和DllUnregisterServer。通过访问这两个标准的ActiveX DLL和OCX入口函数，可以让我们的EXE按照自己的需要来定位和注册自己的组件（译者：这个技术还是有相当价值的。虽然能够通过Shell语句调用RegSvr32.exe来注册组件，但是它仅支持标准的入口：DllRegisterServer和DllUnregisterServer。而使用这里的技术，我们就能够调用非标准的入口，在ATL工程里将两个两个输出函数换个名字，我们在VB里依然可以注册，这样简单的操作就能起到一定的保护组件的作用）。对这样的外部函数来说，我们是通过LoadLibrary和GetProcAddress调用来从外部DLL获取函数指针，并将这个函数指针移到FunctionDelegator结构里以使我们能够调用这个函数指针本身。（见Listing3）

使用函数指针来排序 （译者：这里原文用了几段来演示如何通过函数指针回调的方法来进行数组排序。仅就本文要谈的函数指针调用来说，这和Listing3里的处理方式类似，因为此处省略这几段。）

我们能够在很多方面使用这种调用函数指针技术。比如，我们可以通过在运行时插入具有不同行为的函数来动态改变某段代码的行为。我们也可以通过这种技术在VB里实现type casting（强制类型转换）（译者：通过VarPtr得到一个变量的无类型指针，然后将这个指针做为参数，将这个指针传给不同的类型转换函数指针，并调用之，即可实现强制类型转换）。我不可能把所有可能的应用都列出来，但是这里我再来演示一段小程序。

50

1