FoxTalk 2.0
Solutions for Microsoft® Visual FoxPro® Developers

2005 年 1 月刊

轻松地绑定 Windows 事件 — Doug Hennig 著 Fbilo 译

Page.3

VFP 所缺少的在其它开发环境中的一个功能是捕捉 Windows 事件的能力。VFP 9 扩展了 BindEvent()函数的功能，现在，当 Windows 传递某些特定的消息给 VFP 窗口的时候，BindEvent()可以调用我们自己的代码。这个功能有着很广泛的用途，Doug Hennig 在这里为我们讲述了一部分。

用集合设计，第三部分 — Lauren Clarke and Randy Pearson 著 Fbilo 译

Page.14

这是由三部分组成的用集合设计系列文章的最后一个部分。在第一部分中，Randy Pearson 和 Lauren Clarke 回顾了一些他们在使用了两年的集合以后所学到的经验。在第二部分中，这些经验通过演示怎样使用集合来建立 Visual FoxPro 对象模型来被应用到实际工作中去。这个月，两位作者讨论的是建立 Collection 基类的子类的主题，以满足某些特殊的需求。

在 VFP 里使用视频捕捉 API —Anatoliy Mogylevets 著 CY 译

Page.23

你是否想过要在你的 VFP 应用程序里包含动态视频源？在这篇文章里，WindowsAPI 专家 Anatoliy Mogylevets 将为你展示如何以 VFP 来编写应用程序利用 MS 视频捕捉 API 来访问数字摄像头或其他视频源。

来自 VFP 开发团队的 TIPS —微软 VFP 开发团队 著 LQL.NET 译

Page.36

这个月的 TIPS 是 2004 年 10 月刊 TIPS 的延续，那次我们展示了 BINDEVENT 函数 VFP9 中的新功能：用 VFP 代码来控制 WINDOWS 消息。这个月的 TIPS 则展示了如何捕获电源事件，比如待机或关机。注意：这个 TIPS 不能在 VFP9 公测版下工作，这个功能是在公测版发布以后添加的－因此请在最后发布的 VFP9 上执行这些代码。

工具箱：只运行一次，就一次 —Andy Kramek & Marcia Akins 著 LQL.NET 译

Page.39

这个月，Andy Kramek 和 Marcia Akins 将由简而易地讲述如何保护用户的应用程序，如何阻止应用程序的多个副本运行。

轻松地绑定 Windows 事件

作者：**Doug Hennig**

译者：**fbilo**

VFP 所缺少的在其它开发环境中的一个功能是捕捉 Windows 事件的能力。VFP 9 扩展了 BindEvent()函数的功能，现在，当 Windows 传递某些特定的消息给 VFP 窗口的时候，BindEvent()可以调用我们自己的代码。这个功能有着很广泛的用途，Doug Hennig 在这里为我们讲述了一部分。

Windows 通过传递消息来把事件传送给应用程序。尽管 **VFP** 已经通过 **VFP** 对象中的事件为我们提供了这些消息的一部分——比如 **MouseDown** 和 **Click**——但还是有许多消息对 **VFP** 程序员来说是不可用的。

一个常见的需求是要能够检测到用户在应用程序之间切换的事件。例如，我建立了一个应用程序，该应用挂钩在一个著名的联系人管理系统 **GoldMine** 中，以为当前联系人显示更多的信息。如果用户切换到 **GoldMine**，然后选择了另一个联系人，那么，当他再次切换回我的应用程序的时候，我的应用程序就应该进行刷新，以显示新联系人的信息。不幸的是，在以前版本的 **VFP** 中是不可能做到这一点的；我只有依靠定时器来不断的检查当前显示在 **GoldMine** 中的是哪个联系人。

VFP 9 扩展了在 **VFP 8** 中增加的 **BindEvent()** 函数的功能以支持 **Windows** 消息。实现这个功能的语法是：

```
bindevent(hWnd, nMessage, oEventHandler, cDelegate)
```

在这里，**hWnd** 是接收事件的窗口的 **Windows** 句柄，**nMessage** 是 **Windows** 消息编号，而 **oEventHandler** 和 **cDelegate** 是当窗口接收到消息的时候被触发的对象和方法。与 **VFP** 事件不同的是，只有一个事件处理器（**oEventHandler**）可以被绑定给一个特定的 **hWnd** 和 **nMessage** 组合。指定第二个事件处理器对象或者代理方法会导致第一个绑定被替换为第二个。**VFP** 不会对 **hWnd** 或 **nMessage** 的值的有效性进行检查；如果其中一个无效，就什么都不会发生，因为指定的窗口将无法接收到指定的消息。

对于 **hWnd**，你可以指定 **_Screen.hWnd** 或者 **_VFP.hWnd** 来跟踪那些发送给应用程序的消息，或者指定一个表单的 **hWnd** 来跟踪发送给该表单的消息。**VFP** 控件没有 **Windows** 句柄，但是 **ActiveX** 控件有，所以你也可以绑定到它们身上。

Windows 中有着数百个 **Windows** 消息。这类的消息有：**WM_POWERBROADCAST(0x0218)**，该消息在电池电量过低、或者切换到挂起模式这样的电源事件发生时被发出、

WM_THEMECHANGED(0x031A)，它表示 Windows XP 桌面主题已经被更换；还有 WM_ACTIVATE(0x0006)，它在切换到一个应用程序、或者从应用程序切换出来的时候被触发（Windows 消息通常用一个以 WM_开头的名字来引用）。

在地址：

http://msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/windowui.asp

处有着几乎所有的 Windows 消息的文档。而作为 Platform SDK 一部分的 WinUser.H 文件中有着那些 WM_ 开头的常量的值，你可以从 www.microsoft.com/msdownload/platformsdk/sdkupdate/ 处下载这个 Platform SDK。

事件处理器的方法必须接收 4 个参数：hWnd，是接收消息的窗口的句柄；nMessage，是 Windows 消息的编号；还有两个 Integer 参数，它们的值随着 Windows 消息的不同而不同（每个消息的文档解释了这些参数的值）。方法必须返回一个 Integer，其中包含着一个结果的值。一种可能的返回值是 BROADCAST_QUERY_DENY(0x424D5144，代表字符串“BMQD”)，意思是事件的发生被阻止了。

如果你想让消息通过那种大多数事件处理器应做的常规方式来被处理，你就必须在你的事件处理器方法中调用 VFP 的 Windows 消息处理器；这是种就像是在 VFP 方法代码中使用 DODEFAULT() 这样的办法。你的事件处理器方法很象是返回 VFP 的 Windows 消息处理器的返回值。这里是一个事件处理器完成这种任务的一个例子（别的它什么都不干）：

```
lparameters hWnd, ;
 Msg, ;
 wParam, ;
 lParam
local lnOldProc, ;
 lnResult
 #define GWL_WNDPROC -4

declare integer GetWindowLong in Win32API ;
 integer hWnd, integer nIndex

declare integer CallWindowProc in Win32API ;
 integer lpPrevWndFunc, integer hWnd, integer Msg, ;
 integer wParam, integer lParam

lnOldProc = GetWindowLong(_screen.hWnd, GWL_WNDPROC)
lnResult = CallWindowProc(lnOldProc, hWnd, Msg, ;
```

```
    wParam, lParam)
    return lnResult
```

当然，事件处理器不需要每次都声明 Windows API 函数或者调用 GetWindowLog；你可以把这些代码放在类的 Init 方法中，把 GetWindowLong 的返回值保存在一个自定义属性中，然后在事件处理器要调用 CallWindowProc 的时候使用这个属性。后面的例子会用这种方法来做。

为了判定绑定的是哪种消息，可是使用 AEVENTS(ArrayName,1)。它会在指定的数组中为每个绑定分配一行和四列，并用被传递给 BINDEVENT()的参数的值对元素们进行填充。

你可以使用 UNBINDEVENT(hWnd[,nMessage])来取消事件绑定。忽略第二个参数的话，就会取消对指定窗口的所有消息的绑定。只给这个函数传递一个 0 会取消对所有窗口的所有消息的绑定。在事件处理器对象被释放以后，再次发生消息时事件也会自动取消绑定。

一个新版本的 VFP 怎么会没有一些新的 SYS() 函数呢？在 VFP 9.0 中新增了三个与 Windows 事件相关的 SYS()函数：

- SYS(2325, wHandle)为一个由传递进来的 wHandle（VFP 对 hWnd 的一个内部封装）指定的窗口返回它的客户端窗口的 wHandle（一个客户端窗口是一个在一个窗口内部的窗口；例如，_Screen 是 _VFP 的一个客户端窗口）。
- SYS(2326,hWnd)返回用 hWnd 指定的窗口的 wHandle。
- SYS(2327,wHandle)为用 wHandle 指定的窗口返回其 hWnd。这些函数的文档指出，它们是在使用了 VFP API 库结构工具包的情况下用于 BINDEVENT()函数的。不过，从下面的例子中可以看到，你也可以使用它们来获得一个 VFP IDE 窗口的客户端窗口的 hWnd。

绑定到 VFP IDE 窗口事件

包含在这个月下载文件中的 TestWinEventsForIDE.PRG 演示了怎么将事件绑定到 VFP 的 IDE 窗口。将你想要绑定事件的 IDE 窗口的 Caption 设置为 lcCaption（下面的代码将使用命令窗口来演示），然后运行这个程序。激活和取消激活该窗口，移动、缩放窗口等等；你将看到对 Screen 做出反应的 Windows 事件。试验完了的时候，在命令窗口里输入 RESUME 并按下回车来进行清理。要用这段代码对一个 IDE 窗口中的客户端窗口进行测试的话，需要将代码中被注释掉的部分反注释。

你也可以通过给这段代码添加 BINDEVENT() 语句来绑定其它的事件；为希望绑定的事件使用在 WinEvents.H 中的常量的值。注意，TestWinEventsForIDE.PRG 只工作于非可停靠 IDE 窗口，所以，在你运行这个程序之前要先在你想要测试的窗口的标题栏上单击鼠标右键、确保可停靠选项已被关闭。

这里是这个 PRG 的代码：

```
#include WinEvents.H


lcCaption = 'Command'
```

```
loEventHandler = createobject('IDEWindowsEvents')
lnhWnd = ;
loEventHandler.FindIDEWindow(lcCaption)

* 反注释下面的代码以接收客户端窗口的事件
*lnhWnd = ;

loEventHandler.FindIDEClientWindow(lcCaption)
if lnhWnd > 0
 bindevent(lnhWnd, WM_SETFOCUS, loEventHandler, ;
 'EventHandler')
 bindevent(lnhWnd, WM_KILLFOCUS, loEventHandler, ;
 'EventHandler')
 bindevent(lnhWnd, WM_MOVE, loEventHandler, ;
 'EventHandler')
 bindevent(lnhWnd, WM_SIZE, loEventHandler, ;
 'EventHandler')
 bindevent(lnhWnd, WM_MOUSEACTIVATE, loEventHandler, ;
 'EventHandler')
 bindevent(lnhWnd, WM_KEYDOWN, loEventHandler, ;
 'EventHandler')
 bindevent(lnhWnd, WM_KEYUP, loEventHandler, ;
 'EventHandler')
 bindevent(lnhWnd, WM_CHAR, loEventHandler, ;
 'EventHandler')
 bindevent(lnhWnd, WM_DEADCHAR, loEventHandler, ;
 'EventHandler')
 bindevent(lnhWnd, WM_KEYLAST, loEventHandler, ;
 'EventHandler')
 clear
 suspend
 unbindevents(0)
 clear
else
 messagebox('The ' + lcCaption + ;
      ' window was not found.')
```

```
endif lnhWnd > 0

define class IDEWindowsEvents as Custom
 cCaption = ''
 nOldProc = 0
 function Init
      declare integer GetWindowLong in Win32API ;
           integer hWnd, integer nIndex
      declare integer CallWindowProc in Win32API ;
           integer lpPrevWndFunc, ;
           integer hWnd, integer Msg, ;
           integer wParam, integer lParam
      declare integer FindWindowEx in Win32API;
           integer, integer, string, string
      declare integer GetWindowText in Win32API ;
           integer, string @, integer
      This.nOldProc = GetWindowLong(_screen.hWnd, ;
      GWL_WNDPROC)
 endfunc

 function FindIDEWindow(tcCaption)
      local lnhWnd, ;
      lnhChild, ;
      lcCaption
      This.cCaption = tcCaption
      lnhWnd = _screen.hWnd
      lnhChild = 0

      do while .T.
           lnhChild = FindWindowEx(lnhWnd, lnhChild, 0, 0)
           if lnhChild = 0
                exit
           endif lnhChild = 0

           lcCaption = space(80)
           GetWindowText(lnhChild, @lcCaption, len(lcCaption))
```

```
            lcCaption = upper(left(lcCaption, ;
                at(chr(0), lcCaption) - 1))
            if lcCaption = upper(tcCaption)
                exit
            endif lcCaption = upper(tcCaption)
        enddo while .T.
        return lnhChild
    endfunc

    function FindIDEClientWindow(tcCaption)
        local lnhWnd, ;
        lnwHandle, ;
        lnwChild
        lnhWnd = This.FindIDEWindow(tcCaption)

        if lnhWnd > 0
            lnwHandle = sys(2326, lnhWnd)
            lnwChild = sys(2325, lnwHandle)
            lnhWnd = sys(2327, lnwChild)
        endif lnhWnd > 0
        return lnhWnd
    endfunc

    function EventHandler(hWnd, Msg, wParam, lParam)
        ? 'The ' + This.cCaption + ;
            ' window received event #' + transform(Msg)
        return CallWindowProc(This.nOldProc, hWnd, Msg, ;
            wParam, lParam)
    endfunc
    enddefine
```

这段代码从建立 **IDEWindowsEvents** 类的实例开始。它调用 **FindIDEWindow** 方法来获得对一个其 **Caption** 储存在 **lcCaption** 变量中的窗口的句柄。然后它使用 **BindEvent()**来将所需窗口的特定事件绑定到这个类的 **EventHandler** 方法。这些事件包括激活、取消激活、缩放、移动窗口、以及在窗口中的键击。

IDEWindowsEvents 的 **Init** 方法声明了这个类要用到的 **Windows API** 函数。它还定义了用于调用 **VFP** 的 **WIndows** 消息处理器的值，并将这个值保存在 **nOldProc** 属性中；**EventHandler** 方法使用这个

值来确保常规的事件处理会正常发生。FindIDEWindow 方法使用一对 Windows API 函数来找到指定的 VFP IDE 窗口。它是通过遍历 _VFP 的每个子窗口、并将各个子窗口的 Caption 和被作为参数传递进来的 Caption 进行对比来完成这个任务的。FindIDEClientWindow 做的是类似的事情，不过使用了新的 SYS() 函数来获得对指定窗口的客户端窗口的句柄。

当你运行 TestWinEventsForIDE.PRG 的时候，你会发现并非每个 IDE 或者客户端窗口都会发生每个事件。例如，在属性窗口中你不会看到 keypress 事件的发生。这很可能是由于 VFP 实现窗口的方式与别的 Windows 应用程序不同所导致的。

注意：通常你不会在一个典型的应用程序中使用类似于这里的代码；它只是向那些想要给 VFP 的 IDE 添加自己所需行为的程序员提供的。下一个例子中用到的技术你可能会用在一个最终用户的应用程序中。

绑定到应用程序窗口和磁盘事件

WindowsMessagesDemo.SCX （见图 1）演示了挂钩到激活和取消激活以及特定的一些 Windows Shell 事件——例如插入或取出一张 CD 或者 USB 设备。下面演示了一个 Windows 事件的有趣的用途：这些代码将一个 Windows Shell 事件的子集当作是一个自定义 Windows 事件，并将_VFP 注册为接收这个自定义 Windows 事件。

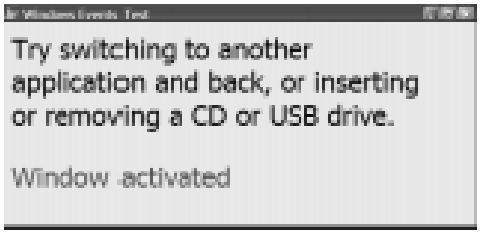

图 1、WindowsMessagesDemo.SCX 演示了你的应用程序可能会需要的几个 Windows 事件。

这个表单的 Init 方法会处理必要的设置问题。跟 TestWinEventsForIDE.PRG 一样，它声明了几个 Windows API 函数，并将用于 VFP 的 Windows 事件处理器的值保存在 nOldProc 属性中。使用自定义的消息 WM_USER_SHNOTIFY 来调用 SHChangeNotifyRegister 以让 Windows 把 _VFP 注册为接收磁盘事件、媒体的插入和移除事件、以及设备的增加和减少事件（在下面的代码中，大写的标志符是定义在 WinEvents.H 或者 ShellFileEvents.H 中的常量）。

然后这段代码为表单将激活事件和设备更动事件以及它刚刚为 _VFP 定义的自定义消息绑定到表单的 HandleEvents 方法。注意：对 SHChangeNotifyRegister 的调用要求 Windows XP 或更高的版本。如果你在使用一个更早版本的操作系统，请将为 This.nSHNotify 赋值的语句注释掉。

```
local lcSEntry


* 声明要用到的 Windows API 函数
```

```
declare integer GetWindowLong in Win32API ;
 integer hWnd, integer nIndex
declare integer CallWindowProc in Win32API ;
 integer lpPrevWndFunc, integer hWnd, integer Msg, ;
 integer wParam, integer lParam
declare integer SHGetPathFromIDList in shell32 ;
 integer nItemList, string @szPath
declare integer SHChangeNotifyRegister in shell32 ;
 integer hWnd, integer fSources, integer fEvents, ;
 integer wMsg, integer cEntries, string @SEntry
declare integer SHChangeNotifyDeregister in shell32 ;
 integer

* 为 VFP 窗口事件处理器取得一个句柄
This.nOldProc = GetWindowLong(_screen.hWnd, ;
GWL_WNDPROC)

* 注册 _VFP 来把特定的 shell 事件当作一个自定义 Windows 事件接收
lcSEntry = replicate(chr(0), 8)
This.nShNotify = SHChangeNotifyRegister(_vfp.hWnd, ;
 SHCNE_DISKEVENTS, SHCNE_MEDIAINSERTED + ;
 SHCNE_MEDIAREMOVED + SHCNE_DRIVEADD + ;
 SHCNE_DRIVEREMOVED, WM_USER_SHNOTIFY, 1, @lcSEntry)

* 绑定到我们感兴趣的 Windows 事件
bindevent(This.hWnd, WM_ACTIVATE, This, 'HandleEvents')
bindevent(_vfp.hWnd, WM_DEVICECHANGE, This, 'HandleEvents')
bindevent(_vfp.hWnd, WM_USER_SHNOTIFY, This, 'HandleEvents')

* 隐藏 VFP 主窗口以便更清楚的看到接下来会发生什么
_screen.Visible = .F.
```

HandleEvents 方法处理已注册了的事件。它使用一个 **CASE** 语句来判定发生了哪个事件，并相应的更新在表单上的状态标签的 **Caption**。特定的事件类型有着由 **wParam** 参数标识的“子事件”；这是用来准确的判定发生了什么事件用的。例如，当一个 **WM_ACTIVATE** 事件发生的时候，**wParam** 指出窗口当前是激活了的还是没有激活、以及激活的发生是否是由任务切换（例如用户按下了 **Alt+Tab**）或者在窗口上单击而

导致的。

处理自定义的 **Shell** 事件要比其它事件复杂一些。在这个案例中，**lParam** 用来识别出事件，而 **wParam** 则包含着一个内存地址，该地址指向的是用于表示被插入或者移除的驱动器的路径。然后，**SYS(2600)**被用来从地址中拷贝出地址所指向的内存中的值，自定义的 **BinToInt** 方法（代码这里没有放出来）把这个值转换成一个整型值，而 **Windows API** 函数 **SHGetPathFromIDList** 则从这个整型值中提取出真正的路径。最后，这个方法调用 **HandleWindowsMessage** 方法，后者简单的调用 **CallWindowsProc** 来获得常规的事件处理行为。这里是 **HandleEvents** 的代码：

```
lparameters hWnd, ;
Msg, ;
wParam, ;
lParam
local lcCaption, ;
lnParm, ;
lcPath

do case
* 处理一个 activate 或者 deactivate 事件
case Msg = WM_ACTIVATE
do case
    * 处理一个 deactivate 事件
    case wParam = WA_INACTIVE
        This.lblStatus.Caption = 'Window deactivated'
    * 处理一个 activate 事件 (任务切换或者在标题栏上单击)
    case wParam = WA_ACTIVE
        This.lblStatus.Caption = ;
            '窗口激活 (任务切换)'
    * 处理一个 activate 事件 (在窗口的客户端区域单击).
    case wParam = WA_CLICKACTIVE
        This.lblStatus.Caption = ;
            '窗口激活 (单击)'
endcase

* 处理一个设备更动事件
case Msg = WM_DEVICECHANGE
    do case
```

```
            case wParam = DBT_DEVNODES_CHANGED
                This.lblStatus.Caption = 'DevNodes 已变动'
            case wParam = DBT_DEVICEARRIVAL
                This.lblStatus.Caption = '设备增加'
            case wParam = DBT_DEVICEREMOVECOMPLETE
                This.lblStatus.Caption = '设备移除 ' + ;
                    '完成'
        endcase
    * 处理一个自定义 shell 通知事件
    case Msg = WM_USER_SHNOTIFY
        do case
            case lParam = SHCNE_DRIVEADD
            lcCaption = '驱动器已增加'
            case lParam = SHCNE_DRIVEREMOVED
            lcCaption = '驱动器被移除'
            case lParam = SHCNE_MEDIAINSERTED
            lcCaption = '媒体已插入'
            case lParam = SHCNE_MEDIAREMOVED
            lcCaption = '媒体已移除'
        endcase

        lnParm = This.BinToInt(sys(2600, wParam, 4))
        lcPath = space(270)
        SHGetPathFromIDList(lnParm, @lcPath)
        lcPath = left(lcPath, at(chr(0), lcPath) - 1)
        This.lblStatus.Caption = lcCaption + ': ' + lcPath
    endcase
    return This.HandleWindowsMessage(hWnd, Msg, wParam, ;
    lParam)
```

运行这个表单，然后象介绍中指出的那样在其它窗口或者桌面上单击再回到表单上以显示激活和反激活事件。插入或者移除一个某种类型的可移动驱动器，比如一个 **USB** 驱动器或者一个数码相机，以了解事件的发生情况。

在该表单中的这类代码有几个实用的用途。例如，我在本文前面提到的 **GoldMin** 插件现在可以在接收到一个 **Activate** 事件的时候刷新自己了。当我把我的带有一个 **USB** 延长线的数码相机连接到我的计算机上的时候，数码相机自带的软件就会弹出窗口提示我去下载照片。一个 **VFP** 房地产或者医学图像应用程序可以对

建筑或者伤口的照片做到类似的事情。

其它用途

绑定 **Windows** 事件还有大量其它的用途。例如，你可以在特定的条件下（比如一个重要的进程尚未完成）阻止 **Windows** 关机。在那样的情况下，你最好绑定到 **WM_POWERBROADCAST** 消息，并且在不应关机的时候返回 **BROADCAST_QUERY_DENY**。

我用 **Microsoft Money** 来管理我的家庭财务，当我从银行下载了一个帐单的时候，**Money** 立刻就会知道并且显示相应的对话框，我很喜欢这种功能。这类的行为现在在 **VFP** 应用程序中也能够做到了；现在，你的应用程序不再需要不停的去检查一个路径以搞清是否增加（或者减少、重命名）了一个文件，这种事情一发生，你的应用程序就会被通知，从而做出相应的反应。

结论

支持 **Windows** 事件绑定是 **VFP** 一个令人难以置信的增强，它让你可以挂钩到发生在 **Windows** 中的任何事情。既然 **VFP** 社群已经开始研究它的能力了，那么我期待着看到它的更酷的用法。

下载文件：**501HENNIG.ZIP**

用集合设计，第三部分

作者：**Randy Pearson & Lauren Clarke**

译者：**fbilo**

这是由三部分组成的用集合设计系列文章的最后一个部分。在第一部分中，Randy Pearson 和 Lauren Clarke 回顾了一些他们在使用了两年的集合以后所学到的经验。在第二部分中，这些经验通过演示怎样使用集合来建立 Visual FoxPro 对象模型来被应用到实际工作中去。这个月，两位作者讨论的是建立 Collection 基类的子类的主题，以满足某些特殊的需求。

上个月，我们使用集合作为“粘合剂”来建立了一个文档对象模型。尽管这个设计有不少强大的功能，但我们还没有修改 **Collection** 类本身过。无论哪个地方使用了集合，用到的都只是 **Visual FoxPro** 自带的 **Collection** 基类：

```
ADD OBJECT <name> as Collection
```

这个月，我们将突破这个限制，并考虑通过建立子类来扩展 **Collection** 基类自身的行为。

试试手

为了让每个人都适应用代码来建立子类的机制，我们先给出一个简单的例子让大家试试手。许多面对对象的语言中都有集合类，它们总是在有一个提供集合中数据项的数量的属性或者方法。某些语言（包括 **Visual FoxPro**）使用 **count**，而另一些（包括 **JavaScript**）则使用 **length**。为了让程序更友好并减少程序员的记忆不同名称的麻烦，为什么不同时支持这两种名称呢？要这么做的话，简单的添加一个 **Length** 属性，并使用一个 **access** 方法来让 **Length** 属性模仿 **Count** 属性。

```
DEFINE CLASS BaseCollection as Collection
Length = NULL
FUNCTION Length_ACCESS
    RETURN this.Count
ENDFUNC
ENDDEFINE
```

与此类似，集合有一个 **Remove()**方法可以删除单个的数据项。你可以通过给这个方法传递一个 **－1** 来删除所有的数据项。无论什么理由，我们从未喜欢过象这样的传递代码标志的方法，而更喜欢一个简单的

Reset()方法。让我们加一个吧！

```
Function Reset
    THIS.Remove(-1)
ENDFUNC
```

然后你可以在任何需要 **Collection** 基类的地方使用这个 **BaseCollection** 类了，而 **Length** 属性和 **Reset** 方法则将是一直都能用的。脑子里有了这些概念以后，让我们来继续尝试一些更重要的例子。

大小写不敏感

在这个系列文章中我们曾讨论过的一个经验是：**Visual FoxPro** 的集合是大小写敏感的。当你使用一个键来从集合中取一个数据项的时候，这就产生了一个约束，你使用的键必须与数据项被添加到集合中的时候所使用的键的大小写格式相匹配。通常这是设计时所希望的，也是其它语言中集合工作的方式。

尽管如此，你还是可能会碰到更需要大小写不敏感的设计的例子。这可能发生在某些你正在封装某些本身就不是大小写敏感的进程或者实体的情况下。比如在一个 **Visual FoxPro** 表中的字段名称们就是一个例子。如果你使用了类似于 **FIELD()** 或者 **AFIELDS()** 这样的命令生成了一个集合，那么所有的字段名称都将会是大写的。以后你就不能使用混合大小写的方式访问集合中的数据项们，尽管混合大小写的方式可能更具可读性。要建立一个大小写不敏感的集合类，需要拦截（或者覆盖）所有传递键作为参数的方法。这种类的一部分内容将会是这样的：

```
DEFINE CLASS CaseInsensitiveCollection as Collection
FUNCTION add(tvItem, tcKey)
    NODEFAULT
    DO CASE
        CASE PCOUNT() = 1 && 没有键被传递
            DODEFAULT(m.tvItem)
        CASE PCOUNT() = 2
            DODEFAULT(m.tvItem, UPPER(m.tcKey))
    ENDCASE
ENDFUNC

FUNCTION item(tvIndex)
    NODEFAULT
    IF VARTYPE(m.tvIndex) = "C" && 这是在用键作为参数来调用
        RETURN DODEFAULT(UPPER(m.tvIndex))
    ELSE
        RETURN DODEFAULT(m.tvIndex) && 这是用索引号作为参数来调用
```

```
        ENDIF
    ENDFUNC
ENDDEFINE
```

研究这段代码，你会发现为了在传递键作为参数的地方能够查找各种大小写方式，我们覆盖了 **Add()** 方法并将这些键转换成了大写。**NODEFAULT** 语句是为了阻止默认的行为，否则的话 **Visual FoxPro** 将会给集合添加两条数据项！类似的，我们覆盖了 **Item()** 方法以将任何被传递进来的键转换成大写。现在我们可以演示这个类的用法了：

```
oCI = CREATEOBJECT("CaseInsensitiveCollection")
oCI.add("Some stuff.", "AbC")
? oCI.Item("aBc") && 工作正常!
```

尽管上面列出的子类适用于大多数编码格式，基于某些原因，它还是不完整的：

- 还需要覆盖 **Remove()** 和 **GetKey()** 方法，因为它们也会接收到传递进来的键。
- **Add()** 方法还需要被扩展为能处理那种被传递进来了三个或四个参数的情况。令人惊讶的是，这类情况不能仅仅通过执行一个 **DODEFAULT(**并带上被传递进来的四个参数**)**来解决，因为除非你明确的匹配正确的调用语法，否则 **Visual FoxPro** 将触发一个错误。这就要求有四个独立的 **CASE** 语句。

这些增加的需求以及更多的说明包含在本月下载文件提供的 **CaseInsensitiveCollection** 类中。

允许冲突

关于使用集合的另一个经验是：键值必须是唯一的。没有了这个约束的话，**Item()**方法就不能正常的起作用了。虽然这个约束本身具有足够的理由，但是这种在试图给集合增加一个其键值已存在的数据项时集合的行为却可能不符合你的需要。

具体的说，**Visual FoxPro** 将触发一个 **2062** 号错误，“指定的键已经存在”。如果你的设计要求能容忍重复的键存在，你可以有几个选择：

- 在试图给集合添加一个数据项之前，先用集合的 **GetKey()** 方法来检查一下键值是否已经存在，如果存在的话再采取相应的措施。
- 在一个 **TRY/CATCH** 结果中封装对 **Add()** 的内部调用，如果触发了 **2062** 号错误再采取相应的措施。
- 建立一个封装了期望的行为的集合子类，从而让用户可以简化上述内容而只需要简单的调用 **Add()**。

前面的两种选择的结果是一样的（尽管性能所有不同），并且可能是特定条件下的最佳选择；不过，这篇文章讲的是建立集合的子类来封装行为的主题，所以让我们把关注的焦点放到第三项上来。不过首先，我们需要决定到底需要什么样的行为。或者决定给集合添加键值相同的多个数据项是否可以接受？

我们找到了两个这样的实例。第一个是“忽略”大小写，这种情况下我们假定任何重复添加的企图是多余的，因此我们放弃添加的企图——什么都不做，因为数据项“已经在那里了。”我们找到的这种需求的一个例

子是构造一个用于生成多个 **RTF(**富文本格式**)**文件的情况。这些文件在开始的时候要求有一个“色彩表”，其中包含着一个在整个文件中要用到的所有颜色的唯一列表。我们的任务是用一些部件来建立大量的文档，并且，坦白的说，如果我们需要绿色的文本，我们就只想 **Add()**它一下而不管文档中的别的部分是否已经提出了同样的需求。下面是一个简单的子类（忽略了接收到第三、第四个参数的可能性），它提供了忽略重复数据项的行为：

```
DEFINE CLASS IgnoreDupsCollection as Collection
FUNCTION Add(tvItem, tcKey)
    NODEFAULT
    IF THIS.GetKey(m.tcKey) > 0
        * 键已经存在，忽略请求
    ELSE
        RETURN DODEFAULT(m.tvItem, m.tcKey)
    ENDIF
ENDFUNC
ENDDEFINE
```

这里我们所做的就是拦截 **Add()**调用以检查数据项是否已经存在。如果是，则我们简单的什么都不做，这样就预先排除了可能会发生的错误。

第二种可能的冲突表现是“替换”的情况，在这种情况下，我们假定目标是替换以前添加的键值相同的数据项。在集合中替换一个数据项与在一个数组中替换有所不同。事实上，你必须先删除旧的数据项然后再添加这个新的。由于在普通冲突不被允许的情况下允许显式的调用替换可能也是设计时希望的，所以我们决定建立一个单独的 **Replace()**方法，根据需要可以调用这个方法。这里是提供了这么一个方法并支持冲突行为的子类：

```
DEFINE CLASS ReplaceDupsCollection as Collection
FUNCTION Add(tvItem, tcKey)
    NODEFAULT
    IF THIS.GetKey(m.tcKey) > 0
        * 键已存在
        THIS.Remove(m.tcKey)
    ENDIF
    RETURN DODEFAULT(m.tvItem, m.tcKey)
ENDFUNC

FUNCTION Replace(tvItem, tcKey)
    IF THIS.GetKey(m.tcKey) > 0
        THIS.Remove(m.tcKey)
```

```
        ENDIF
        THIS.Add(m.tvItem, m.tcKey)
    ENDFUNC
ENDDEFINE
```

把上面的所有东西组合起来

前面我们讲的都是从 **Collection** 基类建立几个独立的子类。你也许会问，是否这些特性的一部分或者全部应该被组合到一个可重用类中去？例如，通过增加一个 **lCaseSensitive** 属性，大小写敏感可以是你的主集合子类中的一个内建的选项。与此类似，你也可以增加一个 **cOnCollision** 属性来允许将你的选择设置为忽略、替换、或者错误（默认）。你可以建立一个提供这种组合功能的 **ComboCollection** 类。

问题是，这么做是否明智。有三个理由决定了答案是“否”。首先，**Add**、**Remove** 和 **Item** 方法将变得非常的笨重以至于变得难以维护和调试。其次，由于这种复杂性，这个类将变得难以再由之派生其它子类。并且第三，无论是否用到了这些可选的功能，由于需要经常的检查每个选项是否被打开，性能也受到了影响。我们的结论是，你可能很少会用到象这样的组合型类，不过在你的军火库中有这么个东西，在某些情况下可能还是有好处的。

非字符型键？

集合的文档声明键必须是字符型的字符串。我们倾向于从字面的意思来理解它。键值当然不能是整型的，否则 **Item()**方法就不能区别键值和一个数值型索引了。

不过，同样的理由可能不适用于日期型值。但当你试图传递一个日期型键给一个集合基类的时候，**Visual FoxPro** 会触发一个 **11** 号错误。也许通过一个子类我们可以拦截任何日期，并把它们转换成字符串，这样就满足了 **Visual FoxPro** 的要求又允许程序员能够使用日期型键。从 **Add()** 方法开始的时候，事情看起来有点希望：

```
DEFINE CLASS DateKeyCollection as Collection
FUNCTION Add(tvItem, tvKey)
    NODEFAULT
    IF VARTYPE(m.tvKey) = "D"
        tvKey = DTOS(m.tvKey)
    ENDIF
    DODEFAULT(m.tvItem, m.tvKey)
ENDFUNC

FUNCTION Item(tvIndex)
```

```
        IF VARTYPE(m.tvIndex) = "D"
            tvIndex = DTOS(m.tvIndex)
        ENDIF
        RETURN DODEFAULT(m.tvIndex)
    ENDFUNC
ENDDEFINE
```

象下面这样用日期型键测试这个类：

```
loCol = CREATEOBJECT("DateKeyCollection")
loCol.Add( "Today!", DATE() ) && 成功啦!
```

但是，当我们尝试通过日期型键访问一个 **Item()** 方法的时候，事情看起来麻烦了：

```
? loCol.Item(DTOS(DATE())) && 成功
? loCol.Item(DATE()) && 失败—11 号错误
```

错误发生在 **DODEFAULT** 这行上，即使是我们已经把参数转换成字符串了还是这样。问题可以象下面演示的那样通过插入一个 **NODEFAULT** 命令来得到解决：

```
FUNCTION Item(tvIndex)
 IF VARTYPE(m.tvIndex) = "D"
     NODEFAULT
     tvIndex = DTOS(m.tvIndex)
 ENDIF
 RETURN DODEFAULT(m.tvIndex)
ENDFUNC
```

尽管我们能够让这个类工作起来，还是必须要警惕覆盖的 **Item()** 方法是否在所有情况下都能正常的工作。多在各种不同的情况下测试一下，而不要简单的假定它总是会正常的工作。

即时对象建立

想象一下一个带有一个“日期”集合的 **calendar (**日历**)**类，集合中的每个数据项来自于一个 **day** 类。每个 **day** 对象都维护着关于在那一天所发生事件的信息。基于性能的原因，你也许喜欢只根据需要来建立 **day** 对象们的实例（意思就是说，只有那些有事件的日期才会为之建立 **day** 对象）。

这种“轻量级”特性的问题是，你失去了通过引用 **day** 数据项随机访问的能力，因为你不知道要访问的那一天的 **day** 对象是否存在。例如，如果你想要给日历添加一个新年的前夜聚会事件，你也许更愿意使用象这样的一个命令：

```
Calendar.Days("20041231").Add("New Years Party!")
```

不过，这段代码依赖于特定的 **day** 对象已经存在这样一个事实。象通常一样，你可以通过在你的调用代码中首先用 **GetKey()** 来进行检查，不过另外还有一个主意：根据需要（或者即时）为集合建立对象的实例。

```
DEFINE CLASS JITCollection as BaseCollection
cItemClass = NULL && you provide
FUNCTION Item(tvIndex)
    IF VARTYPE(m.tvIndex) = "C"
        LOCAL lnKey, loObj
        lnKey = THIS.GetKey(m.tvIndex)
        IF m.lnKey = 0 && 不存在
            NODEFAULT
            loObj = CREATEOBJECT(THIS.cItemClass)
            THIS.Add( m.loObj, m.tvIndex)
            RETURN m.loObj
        ENDIF
    ENDIF
    RETURN DODEFAULT(m.tvIndex)
ENDFUNC
ENDDEFINE
```

现在，我们给它一个简单的默认类，来测试一下（见附带的代码中的 **DayWorker**）：

```
Days = CREATEOBJECT("JITCollection")
Days.cItemClass = "DayWorker"
Days("20041231").Events.Add("New Years Party!")
? Days("20041231").Events[1]

DEFINE CLASS DayWorker AS Custom
ADD OBJECT Events as Collection
ENDDEFINE
```

这个示例的简洁美来自于提供了一个简单的接口让程序员可以访问一个不存在的对象。换句话来说，当下面的代码运行的时候，在 **Days** 集合中并不存在一个 **“20041231”**数据项，但是还是能够正常运行：

```
Days("20041231").Events.Add("New Years Party!")
```

如果你还没有被发生在这里的东西所吸引，那么尝试一下在调试器中逐步跟踪代码的运行。你甚至可以重叠的使用这个概念，把事件当作是可以根据需要建立的对象，从 **JITCollection** 类中取得事件成员。

看哪！——一个数据项都没有！

使用集合的某些诉求是希望有象缺乏的功能数量那样多的接口。前者通常可以用数组、游标或者其它途径来代替；而后者则不行。上个月我们在为 **DBC** 建立一个 **DOM** 的时候讨论过对接口的需求。

但是对于你的数据本身呢？如果你可以通过象 **Customer("ALFKI")**这样的方式引用某条记录并返回一个 **Scatter** 的数据对象，将会是非常方便的。这可以通过使用一种类似于即时集合类的途径来实现，只是还要涉及到一个数据源的问题。同样的轻量级特征也将是可能的：只建立被引用到的对象。假定打开一个游标、并选择一个工作期的问题已经被处理好了，让我们来探讨一下余下的 **Item()** 方法的实现方法：

```
DEFINE CLASS DataCollection as Collection
cKeyExpr = NULL
FUNCTION Item(tvIndex)
    IF THIS.GetKey(m.tvIndex) = 0
        * 数据项不在集合中
        NODEFAULT
        LOCAL lcExpr
        lcExpr = THIS.cKeyExpr + ;
            [='] + m.tvIndex + ']
        LOCATE FOR &lcExpr

        IF FOUND()
            LOCAL loObj
            SCATTER MEMO NAME loObj
            THIS.Add(m.loObj, m.tvIndex)
            RETURN m.loObj
        ELSE
            ERROR 2061 && what they deserve
        ENDIF
    ENDIF
    RETURN DODEFAULT(m.tvIndex)
ENDFUNC
ENDDEFINE
```

发生在这里的事情是：你通过主关键字来请求一个数据项。这个类会检查该记录是否已经被作为一个数据项添加了，如果为否，则从后台的游标中查找。如果找到了，记录就被 **Scatter** 到一个对象，并将这个对象添加到集合中去。如果没找到，则使用 **Error** 命令来模仿一个合适的集合错误。

这里一切都情况良好，但是你也许会问：将数据项添加到集合中去到底是否有必要，可以直接返回 Scatter 来的数据对象呀！事实上，你可以忽略将数据项添加给集合的整个过程，而只是把集合的接口用作一个简单的对象工厂（Object factory，一种设计模式）——总是使用游标来代替。

这样做看起来很酷，但再考虑的深入一些你将认识到这么一个类的特性将不再一样了。使用修改过的类，每次同一个键被传递给 Item()的时候，一个握有当前游标中的值的新对象将被建立；而在原来的类（前面谈到的）中，你会得到对原来用 Scatter 建立的同一个对象的引用。无论哪种办法都不是“正确的”；正确的选择要根据你的需要而定。

假冒的 UDF（用户自定义函数）？

在做总结以前，让我们先进入一个过渡区域一会儿。记得 UDFs（用户自定义函数）吗？在面对对象的世界里它们备受责难，因为它们不能派生出子类，而且本质上必须是唯一的，所以会出现命名冲突的问题（例如，如果你使用了一个第三方的产品，它使用的某个自定义函数的名称与你自己的框架中某个函数的名称相重复）。

集合能够被用来解决某些这类的问题吗？通过建立一个名称与你想要替换的自定义函数名称相同的 PUBLIC 或者在作用范围内的有效 PRIVATE 变量，在某些情况下你可以使用一个“看起来象是”一个用户自定义函数的集合：

```
PRIVATE MyUdf
MyUdf = CREATEOBJECT("SomeCollection")
? MyUdf(<cParm>)
```

这么做的结果是，即使一个同名的自定义函数存在，被调用的也是该集合的 Item() 方法！乍一看，这象是一种不需要破坏已有的代码就可以将某些自定义函数迁移到类中的潜在途径。可使用 Item() 方法的限制导致了自定义函数只能接收一个字符串参数了。由于这种限制，我们决定不再继续探讨这种技术了。

结论

现在我们关于使用集合来设计的系列文章就结束了。我们希望这份材料会对你有所启迪。在日后的文章中，看到集合经常的被用作我们核心设计技术的部分时请不要惊讶。

下载文件：501PERSON.ZIP

在 VFP 里使用视频捕捉 API

原著: Anatoliy Mogylevets

翻译: CY

你是否想过要在你的VFP应用程序里包含动态视频源？在这篇文章里，WindowsAPI专家Anatoliy Mogylevets将为你展示如何以VFP来编写应用程序利用MS视频捕捉API来访问数字摄像头或其他视频源。

Windows Video Capture API可以让你从源，如DV摄像机、TV电视卡和Web摄像头，获得连续的捕捉。你可以在VFP表单或其他窗口上显示帧或视频流。你也可以保存视频帧为DIB文件或者将视频流和音频流保存为AVI文件。

在某些情况下，视频捕捉API虽然可以因更多的目的或更强壮的原因而被其他所选择，如DirectShow API。然而在VFP代码里使用DirectShow API是很困难的，更简单的Video Capature却是更容易使用，并且只需要很少的代码。没有涉及任何的ActiveX控件，并且所需要的两个外部库文件（avicap32.dll 和 user32.dll）几乎在任何的Windows计算机上都有的。

系统要求

在本篇里所讲述的VFP Video Capture 类提供了视频捕捉函数的基本部分。为了测试，你需要VFP6以上版本和任何USB连接的Web摄像头。我对两种USB摄像头作了开发和测试：一年多的Logite(罗技)QuickCam Pro4000和更早的Dimera350C。另外，我发现这个类也可以用于我的ATI TV Wonder Pro电视卡。

注意：在本篇文章里的代码可以运行于VFP7以上，但个别代码稍有不同，在VFP6下运行的少数API有所不同，也包含在下载文件里。

捕捉窗口

视频捕捉API建立于捕捉窗口对象。这就是为什么我将在这里讨论的第一个函数capCreateCaptureWindow，以创建一个捕捉窗口。

```
DECLARE INTEGER capCreateCaptureWindow IN avicap32;
```

STRING lpszWindowName, LONG dwStyle,;

INTEGER x, INTEGER y,;

INTEGER nWidth, INTEGER nHeight,;

INTEGER hParent, INTEGER nID

第一个参数，lpszWindowName，捕捉窗口的标题。给这个参数传递一个字符串或Null。

第二个参数，dwStyle，是应用于捕捉窗口的窗口类型的值。至少有两种组合类型，WS_CHILD 和 WS_VISIBLE，被用于创建一个有效的捕捉窗口。你也可以考虑增加WS_THICKFRAME，WS_SYSMENU，WS_CAPTION，等等其他类型，组合类型通过BITOR()函数来使用。表1给出了窗口类型的列表。

参数x，y，z，nWidth, 和 nHeight，用于设置捕捉窗口的初始位置和坐标。然后，你可以调用API的SetWindowPos，MoveWindow，和 ShowWindow，来调整位置，尺寸，和捕捉窗口是否可见。

参数hParent指向一个父窗口，它是捕捉窗口的宿主。你也可以说捕捉窗口是粘在父窗口上的（这就是为什么WS_CHILD属性是很重要的）。因此，把你的捕捉窗口粘到一个表单（利用表单的HWnd属性作为hParent）或VFP主窗口（利用_Screen.HWnd属性作为hParent）。

表1：可应用于捕捉窗口的窗口类型

Style	Value	Description
WS_CHILD	0x40000000	创建一个子窗口。该类型窗口没有菜单条。该类型不可与WS_POPUP类型共用。
WS_VISIBLE	0x10000000	创建一个初始不可见的窗口。
WS_THICKFRAME	0x00040000	创建一个带缩放边框的窗口。与WS_SIZEBOX相同。
WS_SYSMENU	0x00080000	创建一个标题条带Windows菜单的窗口。WS_CAPTION类型也必须一起设置。
WS_CAPTION	0x00C00000	创建一个带标题条的窗口（包含WS_BORDER类型）。
WS_SIZEBOX	0x00040000	创建一个带缩放边框的窗口。与WS_THICKFRAME类型相同。
WS_HSCROLL	0x00100000	创建一个带水平滚动条的窗口。
WS_VSCROLL	0x00200000	创建一个带垂直滚动条的窗口。

从VFP3到VFP6，你可以通过GetFocus和GetActiveWindow API调用，获得表单和_Screen对象的窗口句柄。注意到GetActiveWindow问题返回一个等同于_Screen.HWnd的句柄（除非VFP会话不是最前面的窗口）。GetFocus返回一个获得键盘焦点的窗口的句柄。当从Form.Activate里调用时，它返回的句柄等同于Form.HWnd。

另一个的API函数，FindWindow，可以用于通过它的类或标题来返回一个顶层窗口的句柄。而

最后一个参数，nID，在需要区别多个捕捉窗口时是很重要的。你也可以把它设为零。注意，你对每个视频源不可以创建多个捕捉窗口。

调用CapCreateCaptureWindow，如果成功，将返回捕捉窗口的句柄。将此句柄存储到一个变量或一个类的成员，以便于你来引用捕捉窗口。你也需要这个句柄以便后面的调用。

与捕捉窗口通信

捕捉窗口对象，使用了类似于COM对象的“黑盒子”（封装）范例，缺少后面多样的属性和方法。取而代替的，客户端，在我们的情形里是VFP应用程序，使用SendMessage函数来与捕捉窗口通信。

注意：SendMessage API函数发送一个指定的消息给一个或多个窗口。它为特定的窗口调用窗口过程，并且一直到窗口过程已经处理该消息才返回。

```
DECLARE INTEGER SendMessage IN user32;
INTEGER hWnd, INTEGER Msg,;
INTEGER wParam, INTEGER lParam
```

第一个参数，hWnd，是窗口过程将要接收消息的窗口的句柄。正如你所记得，capCreateCaptureWindow返回一个捕捉窗口的句柄。对于所有后来的SendMessage调用，都使用这个句柄作为第一个参数。

每二个参数，Msg，是要发送的消息。Windows操作系统支持大量的消息。当接收一个消息时，窗口通常会以特定的方式对消息作出反应。它就完全象给窗口发送一个命令，说“做这”或“做那”。发送者可以是一个应用程序或是操作系统它本身。视频捕捉API使用了一个相对小的窗口消息子集（大约仅70）。参见表2的部分列表。

其它两个输入参数，wParam和lParam，是用于传递特定信息的附加消息。比如，一个捕捉窗口可以接收WM_CAP_SET_PREVIEW消息以开始或停止预览模式。SendMessage是以wParam设置为True或False来进行调用的，lParam是设置为0（不相关的）。

有时，lParam可以指向一个分配的内存缓冲。在这样的情况下，视频捕捉将会给客户端应用程序返回数据。比如，为获取捕捉驱动器的硬件性能（如调色板和叠加支持），客户端应用程序分配一个内存块，并通过SendMessage调用的lParam参数来传递这个块的地址给捕捉窗口。捕捉窗口以适当信息块来填充这个块，在SendMessage返回后对客户端应用程序就会立即变得可用。

对于双重方法的调用，lParam是个整数或指针时，就需要对SendMessage定义两次。

表2：部分视频捕捉消息

Constant	Value	Description
WM_CAP_START	0x0400	
WM_CAP_DRIVER_CONNECT	(WM_CAP_START+10)	连接捕捉窗口到捕捉驱动器。
WM_CAP_DRIVER_DISCONNECT	(WM_CAP_START+11)	断开捕捉驱动器与捕捉窗口的连接。
WM_CAP_DRIVER_GET_CAPS	(WM_CAP_START+14)	返回当前连接到捕捉窗口的捕捉驱动器的硬件性能。
WM_CAP_FILE_SAVEDIB	(WM_CAP_START+25)	复制当前帧到DIB文件。

WM_CAP_DLG_VIDEOFORMAT	(WM_CAP_START+41)	显示对话框让用户选择视频格式。
WM_CAP_DLG_VIDEOSOURCE	(WM_CAP_START+42)	显示对话框让用户控制视频源。
WM_CAP_GET_VIDEOFORMAT	(WM_CAP_START+44)	检索所使用的视频格式副本或视频格式所要求的大小（BITMAPINFO）。
WM_CAP_SET_VIDEOFORMAT	(WM_CAP_START+45)	设置捕捉视频数据的格式（BITMAPINFO）。
WM_CAP_SET_PREVIEW	(WM_CAP_START+50)	打开或关闭预览模式。
WM_CAP_SET_OVERLAY	(WM_CAP_START+51)	允许或关闭叠加模式。在叠加模式下，视频显示是使用硬件叠加。
WM_CAP_SET_PREVIEWRATE	(WM_CAP_START+52)	设置预览模式的显示帧数（毫秒）。
WM_CAP_SET_SCALE	(WM_CAP_START+53)	打开或关闭预览视频图像的滚动。
WM_CAP_SET_SCROLL	(WM_CAP_START+55)	定义显示在捕捉窗口的视频帧部分。
WM_CAP_GET_STATUS	(WM_CAP_START+54)	检索捕捉窗口的状态（CAPSTATUS）。
WM_CAP_GRAB_FRAME	(WM_CAP_START+60)	从捕捉驱动器检索并显示单帧。
WM_CAP_SEQUENCE	(WM_CAP_START+62)	启动视频流和音频流到文件。

```
DECLARE INTEGER SendMessage IN user32 As SendMessage0;
INTEGER hWnd, INTEGER Msg,;
INTEGER wParam, INTEGER lParam

DECLARE INTEGER SendMessage IN user32 As SendMessage1;
INTEGER hWnd, INTEGER Msg,;
INTEGER wParam, STRING @lParam
```

从VFP3到VFP6，你必须在你每次需要以不同接口来调用的时候重新定义SendMessage。

发送你的第一个消息给捕捉窗口—连接到捕捉驱动器

当捕捉窗口创建后，下一步是要连接到一个捕捉驱动器。这个任务的消息是WM_CAP_DRIVER_CONNECT。参数是wParam，它是捕捉驱动器的索引号，可以是0到9的范围。大部情况下，在你的机器上只有单个的捕捉驱动器可用，所以设置wParam为0。lParam对于这个调用是不相关的，于是也设置为0。

```
#DEFINE WS_CHILD 0x40000000
#DEFINE WS_VISIBLE 0x10000000
#DEFINE WM_CAP_START 0x0400
#DEFINE WM_CAP_DRIVER_CONNECT (WM_CAP_START+10)

DECLARE INTEGER capCreateCaptureWindow IN avicap32;
```

```
STRING lpszWindowName, LONG dwStyle,;
INTEGER x, INTEGER y,;
INTEGER nWidth, INTEGER nHeight,;
INTEGER hParent, INTEGER nID

DECLARE INTEGER SendMessage IN user32;
INTEGER hWnd, INTEGER Msg,;
INTEGER wParam, INTEGER lParam

hCapture = capCreateCaptureWindow(Null,;
BITOR(WS_CHILD, WS_VISIBLE),;
100,100, 320,240, _screen.HWnd, 0)

= SendMessage(hCapture, WM_CAP_DRIVER_CONNECT, 0, 0)
```

注意到前面的代码是很不完整的，虽然几乎可以肯定，如果有数码摄像头或其他视频源可用，它将能够在VFP主窗口里显示一个静止的帧。因为这个代码并没有以适当的方式来释放捕捉窗口，VFP应用程序将会被锁住。

正如你所见，capCreateCaptureWindow创建了一个捕捉窗口并把它的句柄存储在hCapture变量里。捕捉窗口是VFP主窗口的子窗口。它定位于坐标点（100，1000），并且捕捉窗口的尺寸是320x320像素。

然后WM_CAP_DRIVER_CONNECT消息被发送给捕捉窗口以试图把它连接到一个索引号为0的捕捉驱动器。大部分情况下是使用MSVIDEO驱动器。如果成功，SendMessage返回1，否则返回0。

测试另一个类型的消息和在简单的类里封装捕捉窗口

当捕捉窗口不再需要时，它必须从捕捉驱动器断开连接然后并消毁。为从当前连接的捕捉驱动器断开连接，捕捉窗口必须接收WM_CAP_DRIVER_DISCONNECT消息。然后，正如我在后面所解释的，DestroyWindow就会处理捉窗口自身。下面是有基本功能的封装的VFP类：

```
#DEFINE WS_CHILD 0x40000000
#DEFINE WS_VISIBLE 0x10000000
#DEFINE WS_CAPTION 0x00C00000
#DEFINE WS_SYSMENU 0x00080000
#DEFINE WM_CAP_START 0x0400
#DEFINE WM_CAP_DRIVER_CONNECT (WM_CAP_START+10)
#DEFINE WM_CAP_DRIVER_DISCONNECT (WM_CAP_START+11)
```

```
DEFINE CLASS CaptureWindow As Custom
hCapture=0

PROCEDURE Init
DECLARE INTEGER capCreateCaptureWindow IN avicap32;
STRING lpszWindowName, LONG dwStyle,;
INTEGER x, INTEGER y,;
INTEGER nWidth, INTEGER nHeight,;
INTEGER hParent, INTEGER nID

DECLARE INTEGER SendMessage IN user32;
INTEGER hWnd, INTEGER Msg,;
INTEGER wParam, INTEGER lParam

DECLARE INTEGER DestroyWindow IN user32 INTEGER hWnd

THIS.hCapture = capCreateCaptureWindow(;
"VFP Capture Window",;
BITOR(WS_CHILD, WS_VISIBLE,;
WS_SYSMENU, WS_CAPTION),;
100,100, 320,240, _screen.HWnd, 0)

= SendMessage(THIS.hCapture, ;
WM_CAP_DRIVER_CONNECT, 0,0)

PROCEDURE Destroy
= SendMessage(THIS.hCapture, ;
WM_CAP_DRIVER_DISCONNECT, 0,0)
= DestroyWindow(THIS.hCapture)

ENDDEFINE
```

你可以用下面两行代码来测试这个类：

```
PUBLIC oCapWin
oCapWin = CREATEOBJECT("CaptureWindow")
```

你或许会认为，除了全局的以外，任何范围的oCapWin都将会在测试程序开始后立即触发oCapWin.Destroy代码。注意到捕捉窗口现在是可以移动的并且是有标题的。这是通过应用附加的窗口风格WS_SYSMENU和WS_CAPTION来实现的。

窗口里的图像仍然是静止的，让我们来把它动起来。

启动和停止视频预览

我将要讨论的另一个消息类型是WM_CAP_SET_PREVIEW。它根据SendMessage调用时跟随发送的wParam参数值来打开或关闭预览模式。增加两个常量和两个方法到CaptureWindow类。

```
#DEFINE WM_CAP_SET_PREVIEW (WM_CAP_START+50)
#DEFINE WM_CAP_SET_PREVIEWRATE (WM_CAP_START+52)

PROCEDURE StartPreview
= SendMessage(THIS.hCapture, ;
WM_CAP_SET_PREVIEWRATE, 30,0)
= SendMessage(THIS.hCapture, WM_CAP_SET_PREVIEW, 1,0)

PROCEDURE StopPreview
= SendMessage(THIS.hCapture, WM_CAP_SET_PREVIEW, 0,0)
```

WM_CAP_SET_PREVIEWRATE消息设置捕捉和显示新帧的毫秒数。操作系统会考虑它所能接受的设置。

保存帧到DIB文件

DIB（独立于设备的位图）是类似于BMP格式的Windows特定的位图。欲保存当前帧为DIB文件，发送WM_CAP_FILE_SAVEDIB消息给捕捉窗口，并跟随目标文件作为lParam参数。因为lParam参数现在包含一个字符型（STRING）而不是整型（INTEGER）值，这个特别的SendMessage调用将不得不重新定义。增加另一个SendMessage定义到CaptureWindow的Init方法里：

```
DECLARE INTEGER SendMessage IN user32 As SendMessage1;
INTEGER hWnd, INTEGER Msg,;
INTEGER wParam, STRING @lParam
```

再次，我想提醒你这样的双重定义在VFP7以上是支持的。在以前的VFP版本里你必须在每次你需要以不同接口来调用时重新定义SendMessage。

在CaptureWindow里增加新的常量和方法。

```
#DEFINE WM_CAP_FILE_SAVEDIB (WM_CAP_START+25)
#DEFINE WM_CAP_GRAB_FRAME (WM_CAP_START+60)

PROCEDURE GrabFrame
= SendMessage(THIS.hCapture, WM_CAP_GRAB_FRAME, 0,0)

PROCEDURE SaveToDib(cFilename)
= SendMessage1(THIS.hCapture, WM_CAP_FILE_SAVEDIB,;
0, @cFilename)
```

当Web摄像头不是处于预览模式时，在调用SaveToDib方法前先调用GrabFrame方法，以获取当前帧。注意到WM_CAP_GRAB_FRAME消息支自动关闭叠加和预览模式。

保存视频流为AVI文件

这个任务涉及发送多个消息。发送WM_CAP_FILE_SET_CAPTURE_FILE消息以为捕捉文件分配文件名，除非你对默认捕捉文件名很满意，它是C:\Capture.avi。发送WM_CAP_SET_SEQUENCE_SETUP消息以设置配置参数，类似于限制AVI文件大小和捕捉时间，或者要求的帧数。注意到SendMessage1接口将被用于发送所有这些消息。最后，WM_CAP_SEQUENCE消息启动视频流和音频流捕捉为文件。
一旦捕捉开始，它将会在按下终止键或限制时间达到时停止。默认的停止键是Esc和鼠标左键。虽然你可以用WM_CAP_SET_SEQUENCE_SETUP消息分配另一个终止键（要求调用RegisterHotKey API），和关闭或打开鼠标左键作为终止事件。

现在我们将不得不面对CAPTUREPARMS结构，它包含视频流捕捉处理的控制参数。首先，我们用WM_CAP_GET_SEQUENCE_SETUP消息对它作读取。然后，我们改变了它的两个成员，它控制限制时间（秒）。最后，我们用WM_CAP_SET_SEQUENCE_SETUP消息把它送回捕捉窗口。这是个结构，每个成员的偏移址和长度都是以字节计。正如你所见到的，所有24个成员是4字节的值（作为限制时间改变的两个成员需要带上加号）：

```
*!* typedef struct {
*!* DWORD dwRequestMicroSecPerFrame; 0:4
*!* BOOL fMakeUserHitOKToCapture; 4:4
*!* UINT wPercentDropForError; 8:4
*!* BOOL fYield; 12:4
*!* DWORD dwIndexSize; 16:4
*!* UINT wChunkGranularity; 20:4
```

```
*!* BOOL fUsingDOSMemory; 24:4
*!* UINT wNumVideoRequested; 28:4
*!* BOOL fCaptureAudio; 32:4
*!* UINT wNumAudioRequested; 36:4
*!* UINT vKeyAbort; 40:4
*!* BOOL fAbortLeftMouse; 44:4
*!* BOOL fAbortRightMouse; 48:4
*!* BOOL fLimitEnabled; 52:4 +
*!* UINT wTimeLimit; 56:4 +
*!* BOOL fMCIControl; 60:4
*!* BOOL fStepMCIDevice; 64:4
*!* DWORD dwMCIStartTime; 68:4
*!* DWORD dwMCIStopTime; 72:4
*!* BOOL fStepCaptureAt2x; 76:4
*!* UINT wStepCaptureAverageFrames; 80:4
*!* DWORD dwAudioBufferSize; 84:4
*!* BOOL fDisableWriteCache; 88:4
*!* UINT AVStreamMaster; 92:4
*!* } CAPTUREPARMS; total 96 bytes
```

为了在VFP里模仿这个结构，你最好能做的就是使用一个字符串。增加一个新的常量和方法到CaptureWindow类里。

```
#DEFINE WM_CAP_FILE_SET_CAPTURE_FILE (WM_CAP_START+20)
#DEFINE WM_CAP_SEQUENCE (WM_CAP_START+62)
#DEFINE WM_CAP_SET_SEQUENCE_SETUP (WM_CAP_START+64)
#DEFINE WM_CAP_GET_SEQUENCE_SETUP (WM_CAP_START+65)

PROCEDURE SaveToAvi(cFilename, nSeconds)
* 目标AVI文件名，捕捉的秒数
LOCAL cBuffer

* 分配CAPTUREPARAMS大小的字符缓冲
cBuffer=REPLICATE(CHR(0), 96)
```

```
* 以CAPTUREPARAMS数据填充字符缓冲
= SendMessage1(THIS.hCapture, ;
WM_CAP_GET_SEQUENCE_SETUP,;
LEN(cBuffer), @cBuffer)

* 设置fLimitEnabled为真，和wTimeLimit秒数
* 注意VFP字符的偏移址为1
cBuffer = STUFF(cBuffer, 53, 4, num2dword(1))
cBuffer = STUFF(cBuffer, 57, 4, num2dword(nSeconds))

* 发送修改的CAPTUREPARMS回给捕捉窗口
= SendMessage1(THIS.hCapture, ;
WM_CAP_SET_SEQUENCE_SETUP,;
LEN(cBuffer), @cBuffer)

* 设置AVI文件名
= SendMessage1(THIS.hCapture, ;
WM_CAP_FILE_SET_CAPTURE_FILE,;
0, @cFilename)

* 开始捕捉，并在nSeconds后自动停止
= SendMessage(THIS.hCapture, WM_CAP_SEQUENCE, 0,0)
```

函数num2dword()转换一个数值为这个结构所要求的4字节表示法。在我的代码里，这个函数不在类里面。你可以把它作为CaptureWindow类的成员。

```
FUNCTION num2dword(lnValue)
#DEFINE m0 0x100
#DEFINE m1 0x10000
#DEFINE m2 0x1000000
LOCAL b0, b1, b2, b3
b3 = Int(lnValue/m2)
b2 = Int((lnValue - b3*m2)/m1)
b1 = Int((lnValue - b3*m2 - b2*m1)/m0)
b0 = Mod(lnValue, m0)
RETURN Chr(b0)+Chr(b1)+Chr(b2)+Chr(b3)
```

注意到捕捉的AVI文件会增长的非常大，因此创建的文件仅能是几秒时间。为了使得AVI文件更小些，你可以调整捕捉窗口的视频格式。

调用对话框以改变捕捉设置

这里有4个消息，WM_CAP_DLG_VIDEOCOMPRESSION，WM_CAP_DLG_VIDEODISPLAY，WM_CAP_DLG_VIDEOFORMAT，和WM_CAP_DLG_VIDEOSOURCE，对应于4个对话框。增加一个新的常量和方法到CaptureWindow类里。

```
#DEFINE WM_CAP_DLG_VIDEOFORMAT (WM_CAP_START+41)

PROCEDURE VideoFormatDlg
= SendMessage(THIS.hCapture, ;
WM_CAP_DLG_VIDEOFORMAT, 0,0)
```

图1是你调用VideoFormatDlg方法后的显示内容。其他三个对话框调用也是同样的风格。

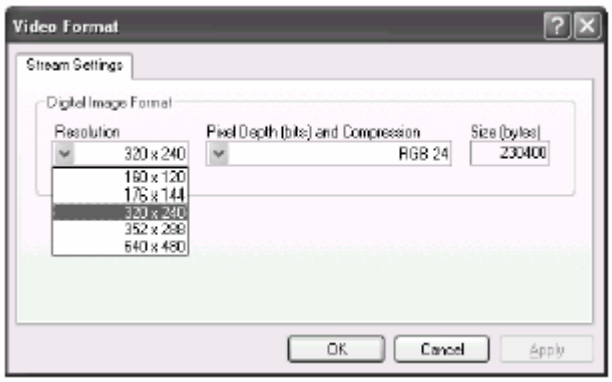

图1：通过发送WM_CAP_DLG_VIDEOFORMAT消息给捕捉窗口以调用视频格式对话框。

在BITMAPINFO结构里获取视频格式参数

另一个视频捕捉消息是WM_CAP_GET_VIDEOFORMAT。对于这个消息，视频捕捉返回一个填充以所使用的视频格式参数的BITMAPINFO结构。还有另一个结构，BITMAPINFOHEADER，在BITMAPINFO结构的开头部分，包含最有趣的视频格式参数。每个成员的依稀址和长度的字节娄如下。

```
*!* typedef struct tagBITMAPINFOHEADER{
*!* DWORD biSize; 0:4
*!* LONG biWidth; 4:4
*!* LONG biHeight; 8:4
```

```
*!* WORD biPlanes; 12:2
*!* WORD biBitCount; 14:2
*!* DWORD biCompression; 16:4
*!* DWORD biSizeImage; 20:4
*!* LONG biXPelsPerMeter; 24:4
*!* LONG biYPelsPerMeter; 28:4
*!* DWORD biClrUsed; 32:4
*!* DWORD biClrImportant; 36:4
*!* } BITMAPINFOHEADER, *PBITMAPINFOHEADER; total 40 bytes
```

增加CaptureWindow类的新属性capWidth 和capHeight，如下面的常量和方法。

```
#DEFINE WM_CAP_GET_VIDEOFORMAT (WM_CAP_START+44)
#DEFINE BITMAPINFOHEADER_SIZE 40

PROCEDURE GetVideoFormat
LOCAL cBuffer, nBufsize

nBufsize=4096
cBuffer = PADR(num2dword(BITMAPINFOHEADER_SIZE),;
nBufsize, CHR(0))

= SendMessage1(THIS.hCapture, WM_CAP_GET_VIDEOFORMAT,;
nBufsize, @cBuffer)

THIS.capWidth = buf2num(SUBSTR(cBuffer, 5,4))
THIS.capHeight = buf2num(SUBSTR(cBuffer, 9,4))
```

函数buf2num()从外部函数转换字符缓冲为VFP数值。类似于num2dword()，它也是不在类里。

```
FUNCTION buf2num(lcBuffer)
RETURN Asc(SUBSTR(lcBuffer, 1,1)) + ;
BitLShift(Asc(SUBSTR(lcBuffer, 2,1)), 8) +;
BitLShift(Asc(SUBSTR(lcBuffer, 3,1)), 16) +;
BitLShift(Asc(SUBSTR(lcBuffer, 4,1)), 24)
```

图2显示了使用CaptureWindow类的VFP表单。

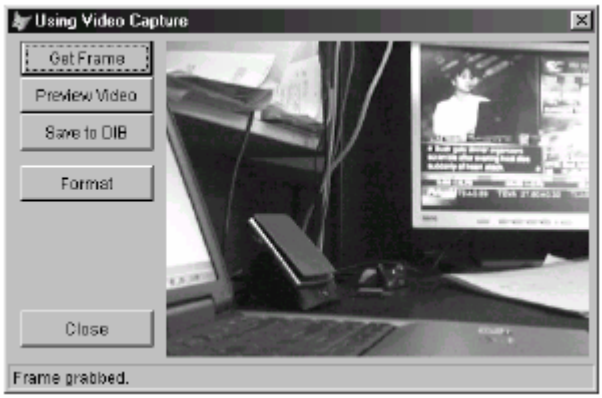

图2：使用CaptureWindow类的VFP表单。

更多

这篇文章讲述了现存的视频捕捉消息的四分之一，虽然我试着突出最有用的部分。查找其他消息可以扩展你的经验并给你带来主意。你可以连接多个Web摄像头，试验不同的视频格式，学习如何保存信息块到AVI文件里，并且试着缩放和滚动捕捉窗口。同样，参见下载文件里的在本文里使用的部分公共术语词汇表。

来自 VFP 开发团队的 TIPS

原著: 微软 VFP 开发团队

翻译: LQL.NET

这个月的 TIPS 是 2004 年 10 月刊 TIPS 的延续，那次我们展示了 BINDEVENT 函数 VFP9 中的新功能：用 VFP 代码来控制 WINDOWS 消息。这个月的 TIPS 则展示了如何捕获电源事件，比如待机或关机。注意：这个 TIPS 不能在 VFP9 公测版下工作，这个功能是在公测版发布以后添加的－因此请在最后发布的 VFP9 上执行这些代码。

绑定 VFP 代码到 WINDOWS 消息，第二部分

下面的代码演示了当 WINDOWS XP 桌面主题改变时 VFP 代码如何执行。你可以用这个例程作为起点去探索这个重要的功能。另外你也可以在本期 Doug Hennig 的文章中读到更多关于 WINDOWS 消息的知识。

```
CLEAR
#define GWL_WNDPROC (-4)
#define WM_QUERYENDSESSION 0x0011
#define WM_POWERBROADCAST 0x0218
#define PBT_APMQUERYSUSPEND 0x0000
#define PBT_APMQUERYSTANDBY 0x0001
#define PBT_APMQUERYSUSPENDFAILED 0x0002
#define PBT_APMQUERYSTANDBYFAILED 0x0003
#define PBT_APMSUSPEND 0x0004
#define PBT_APMSTANDBY 0x0005
#define PBT_APMRESUMECRITICAL 0x0006
#define PBT_APMRESUMESUSPEND 0x0007
#define PBT_APMRESUMESTANDBY 0x0008
#define PBTF_APMRESUMEFROMFAILURE 0x00000001
#define PBT_APMBATTERYLOW 0x0009
#define PBT_APMPOWERSTATUSCHANGE 0x000A
#define PBT_APMOEMEVENT 0x000B
#define PBT_APMRESUMEAUTOMATIC 0x0012
* Return this value to deny a query.("BMQD"):
```

```
#define BROADCAST_QUERY_DENY 0x424D5144
PUBLIC oPowerHandler
oPowerHandler=NEWOBJECT("PowerHandler")
DEFINE CLASS PowerHandler AS session
dwOrigWindProc=0
PROCEDURE init
DECLARE integer GetWindowLong IN WIN32API ;
integer hWnd, ;
integer nIndex
DECLARE integer CallWindowProc IN WIN32API ;
integer lpPrevWndFunc, ;
integer hWnd,integer Msg,;
integer wParam,;
integer lParam
THIS.dwOrigWindProc = ;
  GetWindowLong(_VFP.HWnd,GWL_WNDPROC)
BINDEVENT(0, WM_QUERYENDSESSION,this,"HandleMsg")
BINDEVENT(_VFP.hWnd, WM_POWERBROADCAST, ;
  this,"HandleMsg")
?"Monitoring power"
PROCEDURE HandleMsg(hWnd as Integer, ;
  msg as Integer, wParam as Integer, lParam as Integer)
LOCAL nRetvalue
nRetvalue=0
DO case
CASE msg=WM_QUERYENDSESSION
?PROGRAM(),hwnd,GetWIndowTitle(hwnd)
IF MESSAGEBOX("End Session?",4+32+4096, ;
  "Fox Rocks!")=6 && 6 = yes
nRetvalue= 1 && 0 means don't allow quit
CallWindowProc(this.dwOrigWindProc , ;
  hWnd,msg,wParam,lParam)
ELSE
nRetvalue=0
ENDIF
```

```
RETURN nRetvalue
CASE msg=WM_POWERBROADCAST
?"WM_POWERBROADCAST",SECONDS(),hWnd,msg,wParam,lParam
DO CASE
CASE wParam=PBT_APMQUERYSUSPEND
IF MESSAGEBOX("'Stand by' machine?", ;
4+32+4096,"Fox Rox!")=6
nRetvalue=CallWindowProc(this.dwOrigWindProc, ;
  hWnd,msg,wParam,lParam)
ELSE
nRetvalue=BROADCAST_QUERY_DENY && "BMQD"
ENDIF
OTHERWISE
nRetvalue=CallWindowProc(this.dwOrigWindProc, ;
  hWnd,msg,wParam,lParam)
ENDCASE
OTHERWISE
nRetvalue= CallWindowProc(this.dwOrigWindProc , ;
  hWnd,msg,wParam,lParam)
ENDCASE
RETURN nRetvalue
ENDDEFINE
```

代码见：501TEAMTIPS.ZIP

工具箱：只运行一次，就一次

原著：Andy Kramek / Marcia Akins

翻译：LQL.NET

这个月，Andy Kramek 和 Marcia Akins 将由简而易地讲述如何保护用户的应用程序，如何阻止应用程序的多个副本运行。

Andy：我在客户那边遇到了点麻烦。我有个 VFP 程序一个客户端有好几个人使用，用户们最小化程序后忘了，于是就又重新启动了这个程序一次。

Marcia：嗯我想这样他们的程序运行起来会比较慢。

Andy：嗯用户们经常给维护人员打电话抱怨“这程序太慢了”。 维护人员跑去一看，才发现他们重复运行了整整一打应用程序的副本。

Marcia：阿！ 呵呵不过解决也很简单吧－改下程序启动代码让它只允许启动一次就好啦。

Andy：说起来倒容易，VFP 能让我在启动时设置 DOS 环境变量么？

Marcia：不容易！VFP 有一个 GetEnv()函数用来接受 DOS 环境变量，但却没有 SetEnv()这样的函数。你可以用 RUN 命令，但这确实不是个好办法因为你不能确定程序安装过程中环境变量是否被建立在本机上。

Andy：但他们是在指向文件服务器的网络映射盘上运行程序的，程序没有安装过程。如果我在本机上建立一个零字节的标记文件来代替环境变量你觉得这办法怎么样？这么做很容易而且效果是一样的。

Marcia：什么意思？

Andy：我是说在程序启动代码中这么写：

```
IF NOT FILE( 'c:\alreadyrunning.txt' )
  STRTOFILE( "", "c:\alreadyrunning.txt" )
```

```
ELSE
  RETURN
ENDIF
```

如果文件不存在，我们允许程序启动时建立它，否则我们就知道程序已经运行了就退出。当然，在程序退出代码中我们必须删掉这个文件：

```
IF FILE( 'c:\alreadyrunning.txt' )
  DELETE FILE c:\alreadyrunning.txt
ENDIF
```

Marcia: 不行不行，用这个办法不行。

Andy: 嗯？这可以阻止他们运行多个应用程序的，因此运行速度就不会慢下来了，他们也就不会去麻烦维护人员啦。

Marcia: 想不想打赌？有两个问题。第一，这种情况：用户运行了程序并最小化，然后他要再运行一个副本，双击桌面图标，会怎么样？

Andy: 嗯。。。什么也不会发生。

Marcia: 再试一次，又会怎么样？

Andy: 呵呵知道你的意思了。现在他就会打电话给维护人员了因为他肯定觉得程序坏了。第二个问题呢？

Marcia: 应用程序非正常退出时会怎么样？

Andy: 你说的“非正常退出”是指什么？

Marcia: “非正常退出”是指不执行你的退出程序代码。比如死机或停电或用户忘了关应用程序直接就关机了，等等。

Andy: 哦，知道了。标记文件还留在那里，所以程序再也不能启动了。这是个硬问题！

Marcia: 这样用户必须叫维护人员过来把这个标记文件删掉才行。更严重的是你让用户知道了这个文件

是用来控制程序启动的。

Andy: 我同意,这不是个好办法。那么用 FoxTools 怎么样？Is there anything in there that would help?

Marcia: 嗯这个有点意思，VFP 6.0 以后对 FoxTools 的支持就不是很积极了。不过 VFP 访问这个库调用 WINDOWS 函数比 API 还是要来得容易。主要的问题是它的标准帮助文件(FoxTools.chm)没有公开任何函数的细节。幸运的是，George Tasker 为我们作了这个伟大的工作，你可以从 UT 上下载到他提供的帮助文件叫 ToolHelp.hlp。

Andy: 真不错！不过它能帮我们做什么呢？

Marcia: 好，FoxTools 中有个函数叫“_WFindTitl”这个有用。它可以用窗口标题作为参数返回句柄给 VFP 应用程序（WHandle)。你就可以知道你的应用程序是否已经被运行：

```
*** First Save the title because we don't want the
*** instance that we are trying to start!
lcTitle = _Screen.Caption
_Screen.Caption = SYS(2015)
*** Now look for the original title
SET LIBRARY TO FOXTOOLS ADDITIVE
lnWHandle = _WFindTitl( lcTitle )
IF lnWHandle > 0
  *** Application exists!
  RETURN
ELSE
  *** Reset title
  _Screen.Caption = lcTitle
ENDIF
*** Continue with application startup
```

Andy: 有一个问题。我的应用程序标题是根据客户自己的设置和运行模块而定的，因此没有办法知道当前程序的标题是什么。

Marcia: 哦那可以换个办法。你可以直接用 WINDOWS API 来找到已经运行的程序。FindWindow 函数返回真正的 WINDOWS 句柄(HWnd)。

Andy: 下一步怎么办呢？到现在这个方法还没看到效果阿，不过我觉得有戏。

Marcia: 你想要什么效果呢？这个方法看起来比你的初衷更加全面，这难道不是你真正想要的么？来，看看：

首先检查你的应用程序是否已经运行，如果是，阻止它再次被运行，并且激活已运行的程序将其打开。

Andy: 这正是我想要的。

Marcia: 别急还有很多事要做呢。为了解决这个问题，我们必须做两件事情。第一，当应用程序的第一个实例启动时我们要建一个标记，这时用户要启动第二个实例，这个已存在的标记就会告诉我们应用程序已经运行了。不过这还不能帮我们定位并激活这个应用程序。因此第二件事就是在启动的时候“烙印”你的应用程序，因此第二次运行的时候我们就能定位到它了。你可以通过给窗口加个属性的办法来达到这样的效果。一旦我们找到这个窗口我们就可以定位它。

Andy: 看起来有点道理哦。不过呢，用户要非正常退出该怎么办？

Marcia: 呵呵看清楚不是标记文件是标记，用 Windows API 建的标记，标记随进程建立而建立，随进程终结而释放。这样进程是否由应用程序正常终止就无关紧要了。

Andy: 好棒！具体怎么做？

Marcia: 用 mutex 对象（互斥对象）来实现标记的工作。MSDN 对 mutex 对象定义如下：

mutex 是一个同步对象，当它不被任何线程拥有时其状态为“发信”，当它被某线程拥有时变为“不发信”状态。同一时刻只能有一个线程拥有着 mutex 对象，它的名称对独占访问公共资源相当有用。

Andy: 我明白了。Mutex = 互相排斥的独占。

Marcia: 没错。我们用 CreateMutex 这个 API 函数创建一个 mutex 对象

```
DECLARE INTEGER CreateMutex IN WIN32API ;
        INTEGER lnAttributes, ;
        INTEGER lnOwner, ;
```

```
STRING @lcAppName
```

第一个参数是指向一个结构的指针，这个结构决定返回的句柄是否能被子进程继承下来。你若不想被继承，你可以简单地给出 NULL 指针（即参数为 0）。第二个参数用来指定调用 mutex 对象的进程是否是它的主人（拥有它）。这个参数应该总被设为 TURE（即参数为 1）。最后一个参数是这个对象的名称。

Andy: 我知道了。用这个方法，应用程序没运行的时候 mutex 对象不存在，而当应用程序运行的时候 mutex 对象则被创建，对么？

Marcia: 嗯，但还不完全是。如果你调用 CreateMutex 时 mutex 对象已经存在，这个函数则从对象返回一个新的句柄。因此我们需要另一个 API 函数（GetLastError）来决定我们是否真的要自己创建这个对象。

Andy: 这有点复杂，我们需要亲自做这些活儿么？

Marcia: 如果你真的想用这个解决方案，是的。不过幸运的是，这些代码非常通用：

```
DECLARE INTEGER GetLastError IN WIN32API
*** Try and create a new MUTEX with the
*** name of the passed application
lnMutexHandle = CreateMutex( 0, 1, @lcAppName )
*** Check to see if we created a new object
*** or got the handle to an existing object
IF GetLastError() = 183
  *** Code to find and activate the window
ELSE
  *** Add a property to the window that is
  *** hosting the application so we can find it
  SetProp( _vfp.Hwnd, @lcAppName, 1)
  llRetVal = .T.
ENDIF
```

Andy: 等一下，什么是 SetProp(_vfp.Hwnd, @lcAppName, 1)？

Marcia: 这时另一个 API 函数，用来给指定的窗口添加一个属性并赋值。这就是我说的给应用程序“烙

印”。

Andy: 好的。我们现在运行一个应用程序的实例，当用户再运行一个的时候我们该怎么做？

Marcia: 嗯做这事儿我们要用到 5 个 API 函数。

Andy: 阿！

Marcia: 什么是大人物？他们并不拘泥于 VFP 内置函数，而且换你也不会考虑重新用 VFP 再做这 5 个函数的功能了对吧？

Andy: 就算你说得对，这看起来也很麻烦。

Marcia: 第一个函数我们需要 GetDesktopWindow。它返回 WINDOWS 桌面的句柄。我们得把它和第二个函数联合使用，GetWindow，在桌面找到第一个运行的应用程序窗口。

```
DECLARE INTEGER GetDesktopWindow IN WIN32API
DECLARE INTEGER GetWindow IN USER32 ;
        INTEGER lnhWnd, ;
        INTEGER lnRelationship
*** Get the handle of the first top level window
*** on the Windows Desktop.
lnhWnd = GetWindow( GetDesktopWindow(), 5 )
```

Andy: 为什么第二个参数要设成 5?

Marcia: 这个参数定义目标窗口与桌面怎么关联。这是一个常量值，你可以在 Visual C++附带的文件中找到说明（见表一）

表一：GetWindow 关系常量

常量　　值　说明

GW_HWNDFIRST　0　与指定窗口同级的第一个窗口。

GW_HWNDLAST　　1　　与指定窗口同级的最后一个

GW_HWNDNEXT　　2　　与指定窗口同级的下一窗口。

GW_HWNDPREV　　3　　与指定窗口同级的前一窗口。

GW_OWNER　　4　　指定窗口的父窗口。

GW_CHILD　　5　　指定窗口内的第一个子窗口。

Andy: 我们获取了第一个应用程序窗口，然后呢？

Marcia: 现在我们要在这个窗口中寻找“烙印”。这需要 GetProp 函数：

```
DECLARE INTEGER GetProp IN WIN32API ;
        INTEGER lnhWnd, ;
        STRING @lcAppName
```

Andy: 我们使用 GetWindow 函数返回的句柄来调用而且我们已经知道我们设置的这个属性名称。如果它存在就返回属性值，如果不存在就返回 NULL 值。

Marcia: 没错。假设我们现在已经找到了我们想要的窗口，下一步便是激活它，这需要用到 2 个函数：

```
DECLARE BringWindowToTop IN Win32API ;
        INTEGER lnhWnd
DECLARE ShowWindow IN WIN32API ;
        INTEGER lnhWnd, ;
        INTEGER lnStyle
```

Andy: BringWindowToTop 不用解释了，ShowWindow 里的“style”参数是什么意思？

Marcia: 它指定了那个将被显示的窗口的状态。它的值范围是 0 到 10，分别对应窗口的隐藏、最小化、最大化、还原.....等等。这里我们感兴趣的只是最大化我们的应用程序窗口（值为 3)。激活窗口的代码如下：

```
*** Loop through the windows.
DO WHILE lnhWnd > 0
  *** Is this the one that we are looking for?
  *** Look for the property we added the first
  *** time we launched the application
  IF GetProp( lnhWnd, @lcAppName ) = 1
    *** Activate the app and exit stage left
    BringWindowToTop( lnhWnd )
    ShowWindow( lnhWnd, 3 )
    EXIT
  ENDIF
  lnhWnd = GetWindow( lnhWnd, 2 )
ENDDO
```

Andy: 嗯不错不错真的。

Marcia: 还有件事儿。还记得上面说过 CreateMutex 函数在对象已存在时会返回一个新的句柄么？因此如果我们已经拿到这个句柄则我们必须先释放它。这是我们所需的最后一个 API 函数：

```
DECLARE INTEGER CloseHandle IN WIN32API;
        INTEGER lnMutexHandle
```

Andy: 我很奇怪为什么我们需要保存由 CreateMutex 返回的句柄。到后来我们又必须检查错误代码，我不明白为什么我们需要保存它－那现在我们这么做了。这就是权过程么？

Marcia: 是的！其他的你可以不管拉，现在你需要做的事情就是把这些代码放到你的程序或一个名为“FirstTime”的进程中－并且在你的启动模块中调用它。

```
IF NOT FirstTime( PROGRAM() )
  RETURN
ENDIF
```

Andy: 酷！我的客户一定很高兴拉！这些代码的完全版含在下载文件中，它能在 WINDOWS95 以后的版本上正常工作。

代码在：501KITBOX.ZIP