2005 年第 8 期

在运行时向组合框加入值是很普通的要求。然而，当列表项的源为表时管理它却是个难事，因为组合框只允许数据被加入到单列里。当然，解决的办法是，使用弹出表单来处理编辑的要求，但把它全部集成到一个普通的类来做这些事。本月，**Andy Kramek** 和 **Marcia Akins** 将紧紧围绕这个令人头痛的小问题进行处理并带来一些现成的普通类。

在 VFP 9 表单上的 GDI+：解决绘图问题

作者：Craig Boyd

译者：fbilo

这是 Craig Boyd 写的一个关于“GDI+及其在 VFP 中的应用”的系列文章中的第一篇。这个系列的文章将大量的用到 VFP 9 中自带的 _gdiplus.vcx FFC 类库、以及一些没有被_gdiplus.vcx 封装的 GDI+ 平台 API。Craig 对这些文章的计划，是给你大量的新工具和技术，这样一来，GDI+就不再只跟 VFP 报表有关了。

关于 GDI+ 及其在 VFP 中的应用的好文章终于开始出现了。这该好好的感谢 VFP 9.0 中新增加的 FFC 类库 _gdiplus.vcx 和新的报表功能。

去年，Walter Nicholls 为 FoxTalk 2.0 写了一系列精彩的关于 _gdiplus.vcx 类库及一些其可能用途的文章（见 2004 年 8 月、9 月、10 月的杂志）。如果你没看过这些文章，那么我强烈推荐你去把它们都看一遍。它们至少能够起到启迪的作用，可以让你更快的熟悉我在这篇文章中将会涉及的一些 GDI+ 的主题。

这个系列的文章将较好的接手 Walter 没有谈到的内容，而且这篇文章将指出一些解决在 VFP 表单上使用 GDI+ 绘图时出现问题的技巧。

不罗嗦了，现在开始...

首先，让我们来看看大多数人试图在 VFP 中用 GDI+ 在表单上绘图的一个简单示例。带有图像的结果表单大致会像是图 1 这样：

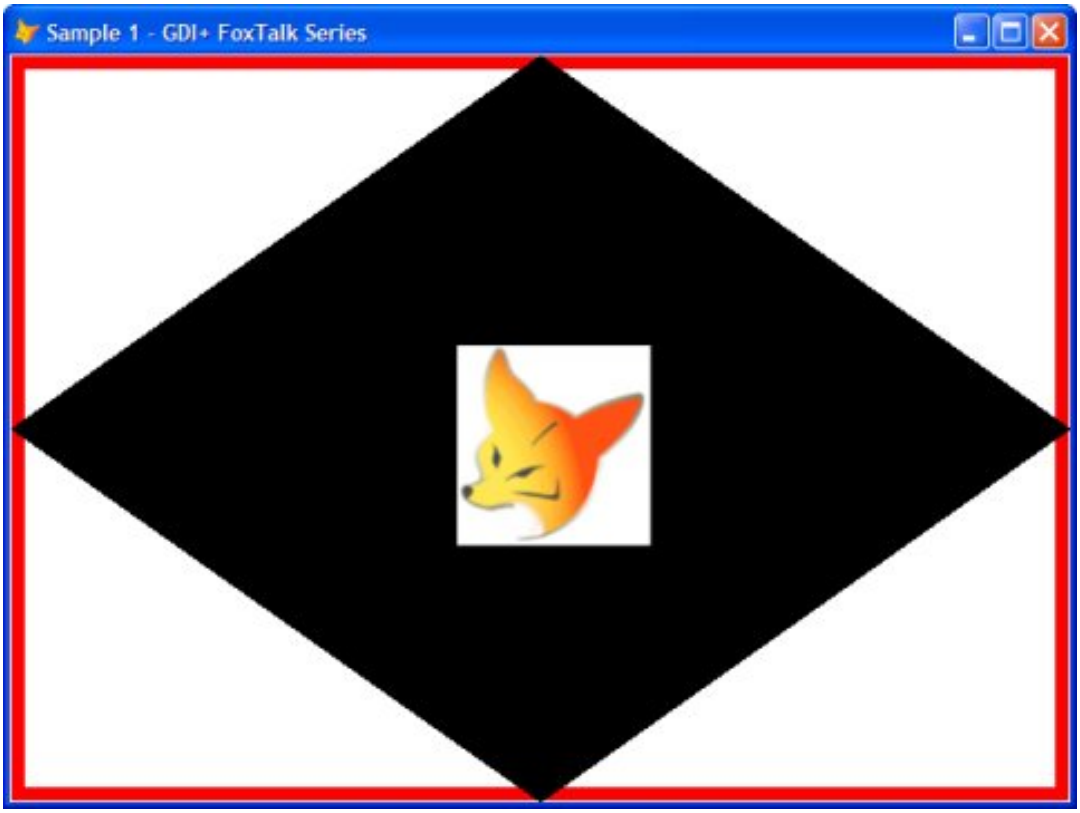

图 1、一个在 VFP 中使用 GDI+ 的快速示例。所有的元素都是分别被绘制的。

```
PUBLIC osample1
osample1=NEWOBJECT("sample1")
osample1.SHOW

DEFINE CLASS sample1 AS FORM

    HEIGHT = 447
    WIDTH = 633
    CAPTION = "Sample 1 - GDI+ FoxTalk Series"
    surfacex = 0
    surfacey = 0
    originalxcoord = 0
    originalycoord = 0
    NAME = "sample1"

PROCEDURE drawshapes
    *!* #DEFINE UNITDISPLAY 1
    #DEFINE UNITPIXEL 2 && This is the one I'll be using
    *!* #DEFINE UNITPOINT 3
    *!* #DEFINE UNITINCH 4
    *!* #DEFINE UNITDOCUMENT 5
    *!* #DEFINE UNITMILLIMETER 6
    #DEFINE IMAGEOFFSET 50
```

```
    DIMENSION aryPoints(4,2)
    LOCAL loGPColor, loGPPen, loGPRect, loGPGraphics, ;
        loGPImage

    WITH THIS
        loGPColor = CREATEOBJECT("gpColor", 255, 0, 0, ;
            255) && INIT Params (Red, Green, Blue, ;
            and Alpha)
        loGPPen = CREATEOBJECT("gpPen", loGPColor.ARGB, ;
            8, UNITPIXEL) && INIT Params (ARGB Color, ;
            Pen Width, Unit of Measure)
        loGPRect = CREATEOBJECT("gpRectangle", ;
            .surfacex + 5, .surfacey + 5, .WIDTH - 10, ;
            .HEIGHT - 10) && INIT Params(X-Coordinate, ;
            Y-Coordinate, Width, Height)
        loGPGraphics = CREATEOBJECT("gpGraphics")
        loGPGraphics.CreateFromHWND(.HWND)
        loGPGraphics.CLEAR(0xFFFFFFFF)
        loGPGraphics.DrawRectangle(loGPPen, loGPRect)
        aryPoints(1,1) = .WIDTH/2 + .surfacex
        aryPoints(1,2) = .surfacey
        aryPoints(2,1) = .WIDTH + .surfacex - 1
        aryPoints(2,2) = .HEIGHT/2 + .surfacey
        aryPoints(3,1) = aryPoints(1,1)
        aryPoints(3,2) = .HEIGHT + .surfacey - 1
        aryPoints(4,1) = .surfacex
        aryPoints(4,2) = aryPoints(2,2)
        loGPColor.Red = 0
        loGPSolidBrush = CREATEOBJECT("gpSolidBrush", loGPColor)
        loGPGraphics.FillPolygon(loGPSolidBrush, @aryPoints, 1, 1)
        loGPImage = CREATEOBJECT("gpImage")
        loGPImage.CreateFromFile(HOME(4) + "Gifs\morphfox.gif", .T.)
        loGPGraphics.DrawImageAt(loGPImage, .WIDTH/2 - ;
            IMAGEOFFSET + .surfacex -10, .HEIGHT/2 - ;
            IMAGEOFFSET + .surfacey -10)
    ENDWITH
ENDPROC

PROCEDURE MOUSEMOVE
    LPARAMETERS nButton, nShift, nXCoord, nYCoord
    IF nButton=2
        WITH THIS
        .surfacex = .surfacex + (nXCoord -  .originalxcoord)
        .surfacey = .surfacey + (nYCoord -  .originalycoord)
```

```
            .originalxcoord = nXCoord
            .originalycoord = nYCoord
            .drawshapes()
            ENDWITH
        ELSE
            THIS.MOUSEPOINTER= 0
        ENDIF
    ENDPROC

    PROCEDURE MOUSEDOWN
        LPARAMETERS nButton, nShift, nXCoord, nYCoord
        IF nButton = 2
            WITH THIS
                .MOUSEPOINTER= 15
                .originalxcoord = nXCoord
                .originalycoord = nYCoord
            ENDWITH
        ENDIF
    ENDPROC

    PROCEDURE LOAD
        SET CLASSLIB TO (HOME(1) + "FFC\_gdiplus.vcx")
    ENDPROC

    PROCEDURE PAINT
        THIS.drawshapes()
    ENDPROC

    ENDDEFINE
```

这种办法里有什么错误？

看上去够简单了，是吗？好吧，这种办法有问题，而且我故意把前面这个例子写错了以强调其中的问题（有时候，从错误的例子里面能够学到更多的东西）。不明白我在说什么吗？请试着缩放一下示例表单、或者用鼠标右键拖着那个 **shape** 到处转一下。我向你保证，意外的结果会像酸痛的拇指一样让你别扭。

可能你会注意到的第一个问题是令人恼火的闪动。如果你确实非常欣赏频闪灯光的话，那么它也许不会让你厌倦（或者也许你是那种总是认为“这是一个功能，不是一个 **Bug**”的人），但对于我们中的大多数人（以及我们的用户）来说，这肯定是个麻烦。

这种闪动效果的根本原因，在于那些元素并不是在同一时刻被画到表单上去的。就在这个简单的示例中，表单的无效部分被重绘、整个界面都被清理干净、重绘长方形、重绘多边形、然后最后，万能的狐狸才被添加到表单上。

要注意的是，这个过程可以在一分钟内被重复几千次。我知道你们中的部分人正在想什么——只要去掉下面这行代码：

```
loGPGraphics.CLEAR(0xFFFFFFFF)
```

继续，我向你保证，这么做的结果会变得比闪光更糟（见图 2）。

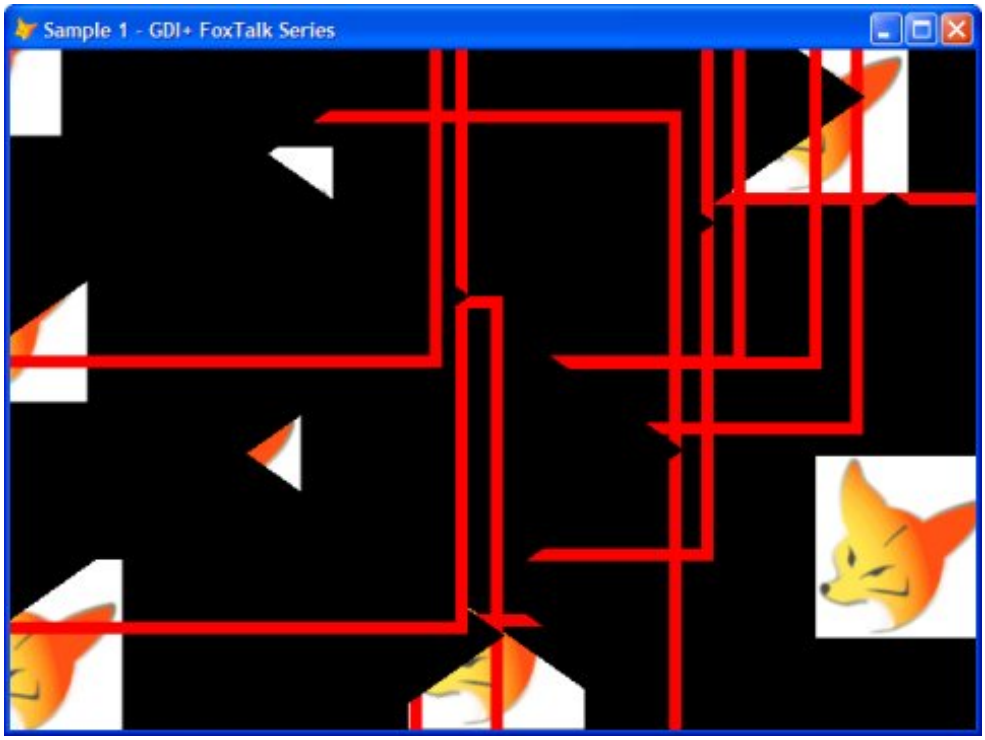

图 2、当你没有正确清理背景时会出现什么情况的示例

即使在回头重新加上这一行代码来清理背景之后，我们仍然有麻烦。这种办法的下一件出错的事情是：至少在这个例子里，难以做到精确。图像并没有像要求的那样在任何时候都被重绘了（见图 3）。

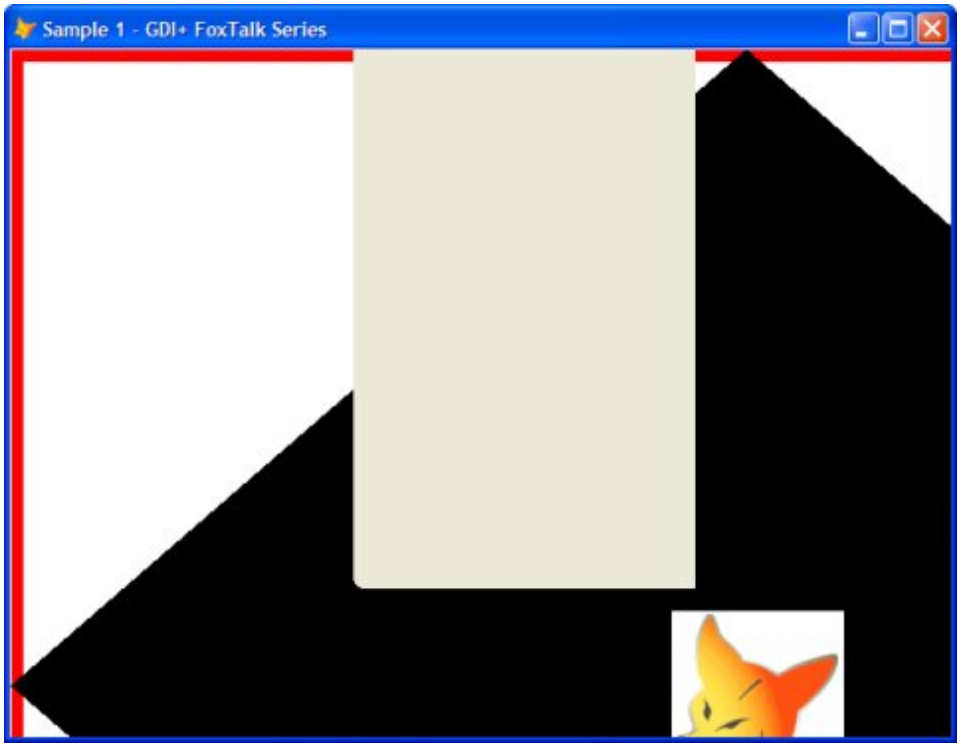

图 3、这是当我最大化了窗口、然后再还原窗口，
接着再把 Yahoo Messenger 调到窗口前面时出现的情况

Windows 只会重绘表单上已经无效的部分。有无数种原因会导致表单上的一部分无效、然后被重绘，但其中最重要的两个原因是：另一个窗口从该表单的上面经过、或者表单被缩放。

太糟糕了，如果你最大化、然后还原示例表单，你会注意到在还原的时候表单的 paint 事件甚至根本没有被触发，给你留下的是丢失了你想让 GDI+ 去绘图的那些图像的表单。

由于这些问题，我在 VFP 中除了报表以外很少使用 GDI+。谢天谢地，现在有一个解决办法了。

GDI+ 的专业用法

我将再试着做一遍同样的事情，但这次我将用专业的办法来做。运行下面的示例，并测试一下它的能力，然后我会解释我是怎么做的以及为什么这么做。

```
PUBLIC osample2
osample2=NEWOBJECT("sample2")
osample2.SHOW

DEFINE CLASS sample2 AS FORM
    TOP = 0
    LEFT = 0
    HEIGHT = 447
```

```
WIDTH = 633
DOCREATE = .T.
CAPTION = "Sample 2 - GDI+ FoxTalk Series"
surfacex = 0
surfacey = 0
originalxcoord = 0
originalycoord = 0
realhwnd = 0
*!* winprocaddress = 0
gpbitmap = NULL
gpcolor = NULL
gppen = NULL
gprect = NULL
gpgraphicsdb = NULL
gpsolidbrush = NULL
gpgraphics = NULL
gpimage = NULL
NAME = "Sample2"

PROCEDURE drawshapes
    #DEFINE IMAGEOFFSET 50
    WITH THIS
        IF .WIDTH > 0 AND .HEIGHT > 0
            .gpgraphicsdb.CLEAR(0xFFFFFFFF)
            .gpgraphicsdb.DrawRectangle(.gppen, .gprect)
            .gpgraphicsdb.FillPolygon(.gpsolidbrush,@aryPoints, 1, 1)
            .gpgraphicsdb.DrawImageAt(.gpimage, ;
                .WIDTH/2 - IMAGEOFFSET + .surfacex - 10, .HEIGHT/2 - ;
                IMAGEOFFSET + .surfacey - 10)
            .gpgraphics.CreateFromHWND(.realhwnd)
            IF .gpgraphics.gethandle() > 0
                .gpgraphics.DrawImageAt(.gpbitmap, 0, 0)
            ENDIF
        ENDIF
    ENDWITH
ENDPROC

PROCEDURE wineventhandler
    LPARAMETERS HWND, msg, wParam, LPARAM
    #DEFINE WM_ERASEBKGND 0x0014
    #DEFINE WM_PAINT 0x000F

    LOCAL lvHDC, lvPaintStruct
```

```
            DO CASE
                CASE msg = WM_PAINT
                    *!* DefWindowProc(HWND, msg, wParam, lParam)
                    lvPaintStruct = SPACE(56)
                    =BeginPaint(HWND, @lvPaintStruct) && Returns HDC
                    THIS.drawshapes()
                    =EndPaint(HWND, @lvPaintStruct) && Returns HDC
                CASE msg = WM_ERASEBKGND
                    RETURN 1
                OTHERWISE
                    *!* CallWindowProc(THIS.WinProcAddress, HWND, msg, wParam,
lParam)
            ENDCASE
        ENDPROC

        PROCEDURE creategpobjects
            #DEFINE UNITPIXEL 2
            WITH THIS
                .gpimage = CREATEOBJECT("gpImage")
                .gpimage.CreateFromFile(HOME(4) + "Gifs\morphfox.gif", .T.)
                .gpbitmap = CREATEOBJECT("gpBitmap")
                .gpcolor = CREATEOBJECT("gpColor", 255, 0, 0, 255) && INIT Params
(Red, Green, Blue, and Alpha)
                .gppen = CREATEOBJECT("gpPen", .gpcolor.ARGB, 8, UNITPIXEL) &&
INIT Params (ARGB Color, ;
                Pen Width, Unit of Measure)
                .gprect = CREATEOBJECT("gpRectangle", 0, 0, 0, 0)
                && INIT Params (X-Coordinate, Y-Coordinate, Width, Height)
                .gpgraphicsdb = CREATEOBJECT("gpGraphics")
                .gpcolor.Red = 0
                .gpsolidbrush = CREATEOBJECT("gpSolidBrush", .gpcolor)
                .gpgraphics = CREATEOBJECT("gpGraphics")
            ENDWITH
        ENDPROC

    PROCEDURE resetgpobjects
        WITH THIS
            .gpbitmap.CREATE(.WIDTH, .HEIGHT)
            .gpgraphicsdb.CreateFromImage(.gpbitmap)
            .gprect.X = .surfacex + 5
            .gprect.Y = .surfacey + 5
            .gprect.X2 = .WIDTH - 5 + .surfacex
            .gprect.Y2 = .HEIGHT - 5 + .surfacey
            aryPoints(1,1) = .WIDTH/2 + .surfacex
```

```
            aryPoints(1,2) = .surfacey
            aryPoints(2,1) = .WIDTH + .surfacex - 1
            aryPoints(2,2) = .HEIGHT/2 + .surfacey
            aryPoints(3,1) = aryPoints(1,1)
            aryPoints(3,2) = .HEIGHT + .surfacey - 1
            aryPoints(4,1) = .surfacex
            aryPoints(4,2) = aryPoints(2,2)
        ENDWITH
    ENDPROC

    PROCEDURE LOAD
        PUBLIC ARRAY aryPoints(4,2)
        DECLARE LONG BeginPaint IN User32 LONG HWND, STRING @lpPaint
        DECLARE LONG EndPaint IN User32 LONG HWND, STRING @lpPaint
        DECLARE LONG GetWindowLong IN User32 LONG HWND, INTEGER nIndex &&
GetWindowLongPtr
        DECLARE LONG GetWindow IN Win32API LONG HWND, LONG uCmd
        *!* DECLARE LONG CallWindowProc IN User32 ;
            INTEGER lpPrevWndFunc, LONG HWND, ;
            INTEGER Msg ,INTEGER wParam, ;
            INTEGER lParam
        *!* DECLARE INTEGER DefWindowProc IN User32 ;
            LONG hWnd,LONG Msg,INTEGER wParam,;
            INTEGER lParam
        SET CLASSLIB TO (HOME(1) + "FFC\_gdiplus.vcx")
    ENDPROC

    PROCEDURE MOUSEDOWN
        LPARAMETERS nButton, nShift, nXCoord, nYCoord
        IF nButton = 2
            WITH THIS
                .MOUSEPOINTER= 15
                .originalxcoord = nXCoord
                .originalycoord = nYCoord
            ENDWITH
        ENDIF
    ENDPROC

    PROCEDURE MOUSEMOVE
        LPARAMETERS nButton, nShift, nXCoord, nYCoord
        IF nButton=2
            WITH THIS
                .surfacex = .surfacex + (nXCoord -  .originalxcoord)
                .surfacey = .surfacey + (nYCoord -  .originalycoord)
```

```
                .originalxcoord = nXCoord
                .originalycoord = nYCoord
                .resetgpobjects
                .drawshapes()
            ENDWITH
        ELSE
            THIS.MOUSEPOINTER= 0
        ENDIF
    ENDPROC

    PROCEDURE INIT
        #DEFINE GW_CHILD 5
        #DEFINE GWL_WNDPROC -4
        #DEFINE WM_ERASEBKGND 0x0014
        #DEFINE WM_PAINT 0x000F
        WITH THIS
            .realhwnd = IIF(.SHOWWINDOW = 2,
GetWindow(.HWND,GW_CHILD), .HWND)
            *!* can use SYS(2327, SYS(2325, SYS(2326,.hwnd)))
            *!* instead of GetWindow() in VFP9
            *!* .winprocaddress = ;
            GetWindowLong(.realhwnd, GWL_WNDPROC)
            =BINDEVENT(.realhwnd, WM_ERASEBKGND, THIS,'WinEventHandler')
            =BINDEVENT(.realhwnd, WM_PAINT, THIS, "WinEventHandler")
            .creategpobjects()
            .resetgpobjects()
        ENDWITH
    ENDPROC

    PROCEDURE RESIZE
        THIS.resetgpobjects
        THIS.drawshapes()
    ENDPROC

    PROCEDURE DESTROY
        UNBINDEVENTS(0)
        WITH THIS
            STORE NULL TO .gpbitmap, .gpcolor, .gppen, ;
            .gprect, .gpgraphicsdb, .gpsolidbrush,
            .gpgraphics, .gpimage
        ENDWITH
    ENDPROC

    ENDDEFINE
```

这里看上去内容有点多，是吗？我对许多地方做了明显的改动，它看起来应该比前面的第一个例子要有效的多了。现在，让我们深入探索一下这些代码，并看看幕后到底发生了什么。

对代码的深入研究

当在工作中使用 **GDI+** 的时候，要记住的第一件事情，是要给那些将会在表单上绘图的方法编写高效的代码。尽可能的把代码写得精干而简练。这是一个严格的规则，任何方法都不能例外。记住，这些方法可能会每秒被调用许多次，因此需要尽最大的努力去检查和优化你的代码。为了这个目的，我把那些声明语句放在了表单的 **Load** 事件中：

```
DECLARE LONG BeginPaint IN User32 LONG HWND, STRING @lpPaint
DECLARE LONG EndPaint IN User32 LONG HWND, STRING @lpPaint
DECLARE LONG GetWindowLong IN User32 LONG HWND, INTEGER nIndex && GetWindowLongPtr
DECLARE LONG GetWindow IN Win32API LONG HWND, LONG uCmd
```

我建立了一些表单属性来握有 **GDI+** 对象们：

```
gpbitmap = NULL
gpcolor = NULL
gppen = NULL
gprect = NULL
gpgraphicsdb = NULL
gpsolidbrush = NULL
gpgraphics = NULL
gpimage = NULL
```

并且，我写了两个新的表单方法：**CreateGPObjects** 和 **ResetGPObjects**。

是的，我在 **LOAD** 事件中还初始化了两个全局数组，因为我不能把它们建立为表单属性。数组必须通过引用被发送给 **thisform.gpgraphicsdb.FillPolygon()** 方法。唯一的另一种选择，是在 **DrawShapes** 方法中定义数组，但这样会破坏编写高效代码的严格规则。

顶层表单并非如我们所见

特别要注意的一点是，我给表单建立了一个 **RealHwnd** 属性以作为表单的 **hWnd**。这么做的原因很简单。在 **VFP** 中，顶层表单其实是两个窗口！其中的一个被用做另一个的框架（父窗口）。在一个顶层表单上的 **hWnd** 属性指向的是那个父窗口，而顶层表单中的整个客户端区域（灰色的部分）其实是另一个窗口——更准确的说，是一个顶层表单的子窗口。

因此，如果我正在处理的是一个顶层表单，那么我就将 **RealHwnd** 属性设置为子窗口的 **hWnd**，在一个没有可显示的客户端区域的窗口上绘图可没什么好处。

使用 **VFP 9** 的能力来挂钩到多个 **Windows** 消息（**Win Msg**）事件上

在 **Init** 事件中，我使用了 **VFP9** 的能力来挂钩到 **Win Msg** 事件上。这是微软的 **Visual FoxPro** 开发团队为 **VFP 9** 增加的最棒的增强之一。

在这个案例中，我对两个消息感兴趣：**WM_ERASEBKGND** 和 **WM_PAINT**。我将使用它们来使自己摆脱对表单的 **Paint** 事件的依赖，并获得一些对“通常由 **Windows** 处理的绘图事件的各个方面”的控制。

表单的 **Paint** 事件没法用（就像你前面看到的那样）。它只在表单的无效区域已经被重绘后触发，而你将需要的却是在此之前取得控制权。因此，我将让 **Windows** 将前面提到的消息发送给我的表单的 **WinEventHandler** 方法。

注意：在第二个示例中有一些代码被注释掉了。其中包括注释掉了 GetWindowLongPtr，它是一个最终将代替 API 函数 GetWindowLong 的函数（它有 32/64 位的含义）。至于被注释掉的 DefWindowProc 和 CallWindowsProc，有些时候需要在 WinEventHandler 方法中调用这些函数，一会儿我会谈到。

在 WinEventHandler 方法内部

在 **WinEventHandler** 方法中，我声明了重要的参数去处理 **Windows** 消息：

```
PROCEDURE wineventhandler
    LPARAMETERS HWND, msg, wParam, LPARAM
```

我还用一个 **56** 个空格的字符串伪造了一个 **PAINTSTRUCT**（绘图结构）（建立结构以供调用 **Win32 API** 时使用超出了这篇文章的讨论范围）：

```
lvPaintStruct = SPACE(56)
```

然后，我设立了一些简单的 **CASE** 语句来分别处理 **WM_ERASEBKGND** 和 **WM_PAINT** 消息：

```
    DO CASE
        CASE msg = WM_PAINT
            *!* DefWindowProc(HWND, msg, wParam, lParam)
            lvPaintStruct = SPACE(56)
            =BeginPaint(HWND, @lvPaintStruct) && Returns HDC
            THIS.drawshapes()
            =EndPaint(HWND, @lvPaintStruct) && Returns HDC
```

```
        CASE msg = WM_ERASEBKGND
            RETURN 1
        OTHERWISE
            *!* CallWindowProc(THIS.WinProcAddress, HWND, ;
                msg, wParam, lParam)
    ENDCASE
```

我通过返回 1 代替了 WM_ERASEBKGND 消息（也可以直接 Return 或者什么也不做，但我通常在其它的语言中都是返回 1 的，所以这里我也继续这么做）。这会阻止表单的背景被清除，因为我已经在表单的 DrawShapes 方法中实现了清除的任务了。

对于 WM_PAINT，我模仿了 Windows 通常会处理这个消息的做法，只是我使用了 DrawShapes 作为我的 Paint 方法。如果我没有在此前或此后任何一端调用 BeginPaint 和 EndPaint，那么 Windows 将不断的向我的窗口发送 WM_PAINT 消息，直到奶牛回家为之。

如果你没有把这一切处理正确，后果会很严重，你的应用程序最后会消耗掉几乎所有的 CPU 周期，因为它把一切都投入到这些漫游的消息中去了。

总是有另一种办法可选

公平的说，还有另一种办法。你可以通过调用 DefWindowProc 而不是 BeginPaint 和 EndPaint 来让 Windows 处理这件重要的事情。当处理这类消息的时候，DefWindowProc 告诉 Windows 就按它通常默认的办法完成任务。

你可以看到，当 Windows 接收一个 WM_PAINT 消息的时候，会发生一些通常会上演的默认处理过程。所以，通过调用 DefWindowProc，你就是在告诉 Windows 只管按它自己的办法去做无论什么事情。

这种办法在以下情况下会很有用：当你不确信自己完全弄清了你正在想要替代的“ Windows 对一个消息的默认处理”是怎么样的、或你是在或者改变其行为或当你更希望让 Windows 去为你照料好默认的过程时。

在这个案例里，我选择了 BeginPaint 和 EndPaint ，因为我知道它们需要去正确的处理 WM_PAINT 消息。它们还为我提供了对事件发生序列的更多控制，并且它们都会返回一个句柄给窗口的显示设备上下文（hDC）。当在工作中使用 GDI+ 的时候，后一个好处会提供更多的帮助，因为大量的 GDI+ 平台 API 函数会要求一个 hDC 作为参数。

继续传递消息是一个好主义

在前面的代码中，从某种意义上来说，我覆盖了 WM_APINT（我广义的使用这个术语）消息，并替换了 WM_ERASEBKGND 消息。但是，为了讨论的更深入一些，假设我需要根据这些消息做一些什么事情，然后再送它们上路以便 Windows 或者其它依赖这些消息的应用程序也能够处理它们。我就更愿意调用一下 CallWindowProc，并发送给它必要的参数。

我已经包含了（并且注释了）声明语句、实际的调用、以及一个“你需要或者想要使用 CallWindowProc 时”应该要用到的 WinProcAddress 属性。在这个特定的示例中，我没有使用 CallWindowProc，因为我不希望让 Windows 去处理 WM_PAINT 或者 WM_ERASEBKGND 消息。如果我发出了对 CallWindowProc 的调用，这个窗口将会在我已经渲染到表单上的图形上进行闪动和绘图。

作为一个通用的规则，无论如何消息都应该在你处理完它们后再被发送返回一次，因为可能有其它进程要求或需要去处理这些消息。只有当你确实有理由这么做、或者相当确定你自己在干什么的时候才可以例外，如前面的 CallWindowProc。

双缓冲的解决方案

你也许已经注意到了我在第二个例子中建立了一个名为 GPGraphicsDB 的新 GPGraphics 对象。

```
PROCEDURE creategpobjects
    #DEFINE UNITPIXEL 2
    WITH THIS
        * 一些前面的代码
        .gpgraphicsdb = CREATEOBJECT("gpGraphics")
        * 一些后面的代码
    ENDWITH
ENDPROC
```

这个 DB 是为了“双倍的缓冲”而存在的，并且是用来减少“你在前面第一个示例中看到的令人烦恼的闪动问题”的重要技巧之一。这里的概念相当简单。我不是把每一样东西直接绘制到表单上，而是先把它们绘制到内存中的一个位图上（某些情况下可以把这看作是一个“脱离屏幕的位图”），最后再一次将全部结果组成的位图绘制到表单上。因此，GPGraphicsDB 向 GPBitmap 绘图，只有当全部绘图结束以后，GPBitmap 才移交给 GPGraphics 去渲染到屏幕上。

这种技术可以用在你使用 GDI+ 的任何语言中，所以 VFP 也不例外。事实上，VFP 正是以这种技术来渲染表单的，我们每天都会用到。这就是为什么 VFP 的基础控件没有任何 hWnds。它们并非真正的窗口，而只是被绘制在一个脱离屏幕的位图上的图形，然后被通过位块传输（BIT BLOCK Transferred——读做“bit-blit”）给屏幕。非常整洁，嗯？

明显得违反规则

在表单的 DrawShape 方法中还有一行代码也许会引起你的关注，所以，在结束这篇文章以前我花一点时间去谈谈它：

```
.gpgraphics.CreateFromHWND(.realhwnd)
```

你也许会认为我这么做违反了我提出的第一个严格规则，你也许是对的。它出现在 DrawShape 方法中，是因为这是一个变通的办法。由于某些不明的原因，如果我把这行代码换成放在 CreateGPObjects 方法中，它将生成一个面积等于表单大小的裁剪区域，并且拒绝在该区域之外绘图，即使表单已经被缩放了。

我试验了各种办法试图去加大这个裁剪区域的大小、而又不需要调用 gpGraphics 类的 Create 方法，包括 GdipSetClipRectI——但都没有效果。所以，目前这是我唯一的办法了。如果你有一个解决这个问题的好办法，请发 Email 给我（或者把它贴到这个世界上看得到的什么地方）。

总结

好吧，这次就先到这里了。我希望你喜欢这第一篇文章，并发现到在我们已经克服了一些障碍之后 GDI+ 的潜力。以后的文章将在这些基本的信条之上展开。你可以期待着看到一篇关于使用 GDI+ 去打开、操作、和保存图像（包括多页的 TIFF/GIF 以及存储在表中的二进制图像数据）、一篇关于将 GDI+ 用于数据表现（包括图表[Chart]/图形[graph]以及实时数据）的文章。

我甚至将放出一些封装了这个功能的类，这样一来，在 VFP 中使用 GDI+ 的工作将会变得比以往更轻松。请继续关注、并坚持订阅当前的 FoxTalk 2.0，因为我保证，我们还仅仅开头而已！

附件：508BOYD.ZIP

给智能感知的 Favorites

作者：Doug Hennig

译者：fbilo

在这篇文章中，Doug Hennig 将带着你参观智能感知中 Favorities ——一个扩展智能感知的新工具，它让你可以指定一个类的哪些成员被显示在智能感知中，以及轻松的编写 ToolTips 脚本或者属性值或方法参数的列表。

智能感知无疑是 Visual FoxPro 曾经增加的最佳功能。它为 VFP 开发人员提供的巨大的生产效率的提升，比任何时候增加的任何东西提供的都多。不过，关于智能感知，有一件一直困扰我的事情，就是当在一个类上使用它的时候，它总是显示那个类的所有成员，而不是只显示我希望看到的那一个。

例如，图 1 展示的是智能感知为 ConnectionMgr 类显示的内容。注意，虽然我们只对少数几个自定义属性和方法感兴趣，智能感知还是把所有东西都显示出来了。这就需要花更多的精力去正确的选择你需要的成员，尤其是如果你不熟悉这个类的时候。

直到 VFP 9 以前，属性窗口也有类似的问题。在这个版本中，微软增加了一个 Favorites 页，它只显示你定义为属于该页的那些成员（一种办法是：在属性窗口中用右键单击某个成员，然后选择“Add to Favorites”）。所以现在，就不需要再去遍历成打的成员来找到你想找的那个了，你可以从你已经指定显示到 Favorites 页上的少量成员中去挑选。

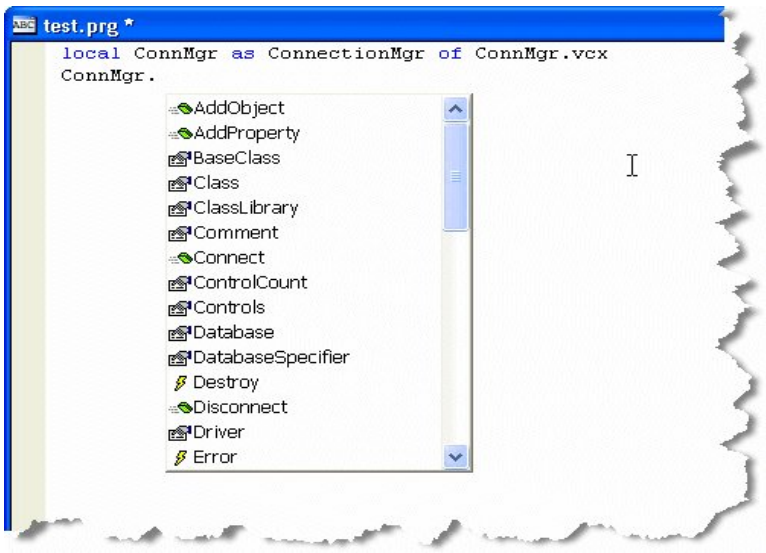

图 1、尽管智能感知让你可以从一个列表中选取一个成员名称，

但它显示的数据项比我们所需要的多多了

我所希望看到的，就是给智能感知一个像 **Favorites** 页那样的东西。所以，我决定自己做一个。结果就是 **Favorites for IntelliSense(给智能感知的 Favorites, 简称 FFI)**。

Favorites for IntelliSense

在探讨其内幕之前，让我们先来看一下 **FFI** 的表现。我们需要做的第一件事情，是在 **FFI** 中注册一个类。打开在 **ConnMgr.VCX** 中的 **ConnectionMgr** 类（附带在下载文件中），然后在命令窗口中输入 **DO FORM FFIBUILDER**。你将看到图 2 中的对话框。

译者注：实际使用中可能会出现一个“无法打开 Favorites for IntelliSense's table”的错误，这是因为当前目录并不在 FFI 工具目录下。建议大家都把下载文件中的以下文件拷贝到 VFP 主目录下：

FFI.VCX
FFI.VCT
FFIBuilder.SCX
FFIBuilder.SCT
FFI.DBF
FFI.CDX
FFI.FPT

然后修改一下 FFI.VCX 中 ffibuildform 和 ffifoxcode 类的 cffitalbe 属性，将这个属性修改为 “=Home()+ffi.dbf”。

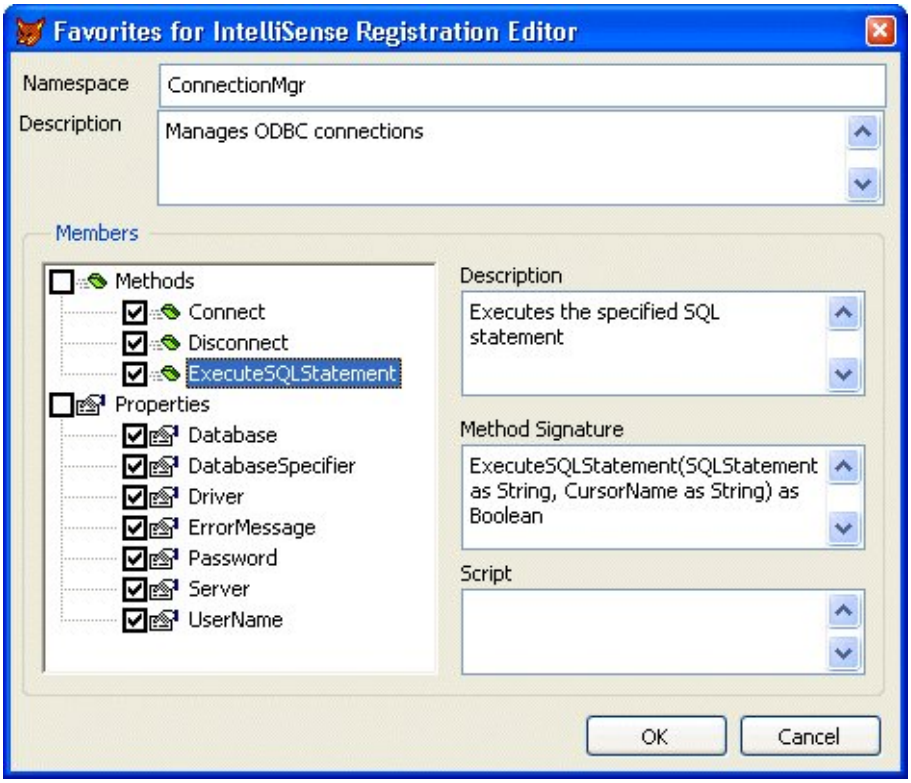

图 2、Favorites for IntelliSense 生成器让你可以为选定的类指定哪些东西要显示在智能感知中

这个对话框允许你指定该类的“名称空间(namespace)”。名称空间是当你输入 LOCAL SomeVarialbe AS 的时候将会显示在智能感知列表中的名称。默认就是这个类的名字，但你也可以根据需要指定别的什么东西。例如，对于一个名为 cApplication 的类，你也许想用 Application 来作为它的名称空间。

如果这个类被用作另一个类的成员，那么，这个对话框中的 description（说明，描述）就会被用作该类在智能感知列表中的 ToolTip（例如，cApplication 类示例中的 User 属性包含 cUser 类的一个实例，于是 cUser 类的 description 就被用作 User 属性的 ToolTip）。它默认为该类的说明，就是在“类”菜单中用类信息（Class Info）菜单项打开的对话框中的 description、或者当类在项目管理器窗口中被选中时从右键菜单中选择“编辑说明”所修改的内容。

对话框里面的 TreeView 会显示这个类的全局自定义属性和方法；你如果想要让类把所有的属性/方法都显示出来的话，可以修改在 FFI.VCX 类库里 FFIBuilderForm 类的 LoadTree 方法中的 AMEMBERS() 语句。在名称前的复选框说明该成员是否被包含在智能感知中；默认的情况下，所有的自定义成员都被包含进去了。

对话框里的 description （说明、描述）用作在智能感知列表中的成员的 ToolTips，它默认为你在建立这个成员时为它输入的说明。

对话框里的 Method signature （方法特征）部分，是在你为该方法输入一个括号或者逗号的时候作为 ToolTips 显示在智能感知中的。这个 ToolTip 告诉你什么参数可以被传递给该方法。这个特征默认为该方法的名称和在方法中的任何 LPARAMETERS 语句的内容，但你也可以根据需要去编辑它以显示你想要的任何东西，包括返回值的数据类型。

对话框里的 Script （脚本）编辑框让你可以输入代码，这些代码是在你从智能感知中选择了这个成员并输入了一个括号、一个参数列表中的逗号、或者一个等于号（为了赋值给一个属性）时被执行的。稍候我们会看到一个示例。

在这个对话框中做些你希望的改动，然后单击 OK。这个表单将为该类及其每个注册了的成员向一个 FFI 表中添加相应的记录。它还向你的智能感知表中添加了两条记录：一条用于这个类，另一条用于这个类的智能感知脚本。稍候我们也会看到关于这个的更多细节。

现在让我们来看看给智能感知的 Favorites 是怎么工作的，建立一个 PRG，并输入 LOCAL ConnMrg AS。当你在 AS 后面按下空格键的时候，你将看到如图 3 中那样被智能感知列出的类型（Type）。当你选择了 ConnectionMgr 并按下回车键的时候，你将看到下面的代码被插入到 PRG 中（其中的 Path 将会是 VCX 文件所在的路径）：

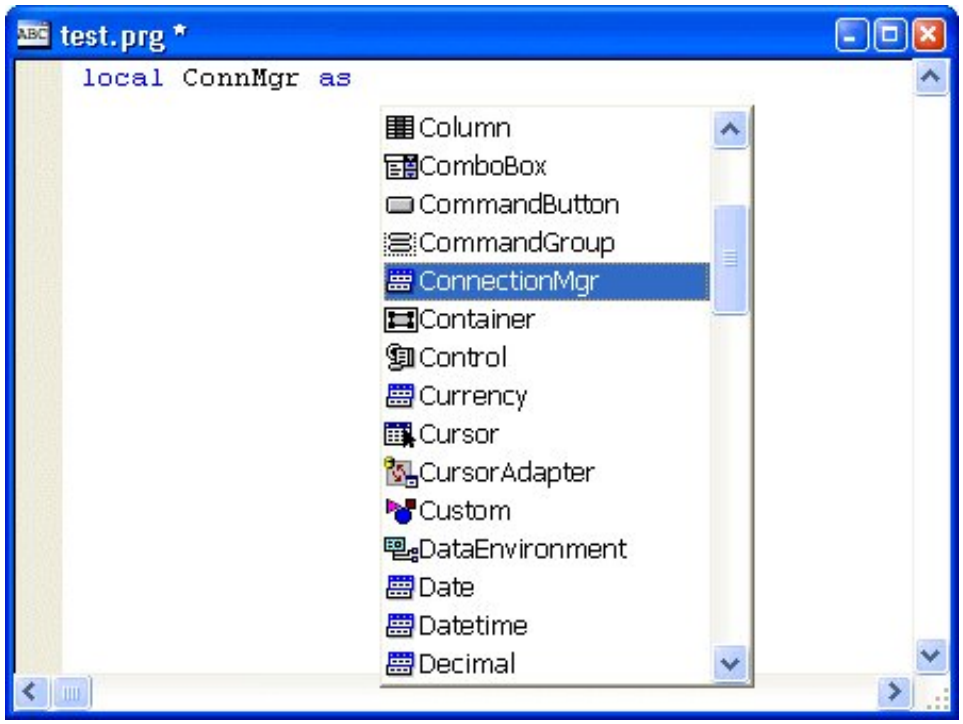

图 3、任何用给智能感知的 Favorites 注册了的类都将被作为一个类型显示在智能感知中

```
local ConnMgr as ConnectionMgr
ConnMgr = newobject('Connectionmgr', 'Path\connmgr.vcx')
```

现在，象图 4 那样输入 **ConnMgr** 并跟上一个句号，智能感知将只显示这个类的已经注册了的成员（这就是“智能感知的 **Favorites**”了）。如果你选择了一个象 **ExecuteSQLStatement** 那样的方法，当你输入左括号的时候，你将会看到作为智能感知的 **ToolTips** 出现的该方法的特征，它使得你可以更清楚的发现应该传递给该方法什么样的参数。

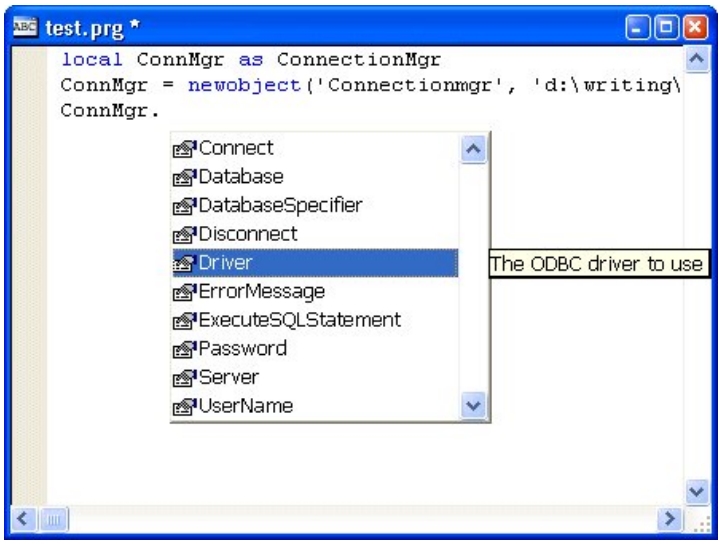

图 4、智能感知只为指定的类显示注册了的成员。把这个图跟图 1 比较一下

它是如何工作的

隐藏在 FFI 背后的秘密依赖于两件事情：智能感知是如何处理在智能感知表中被定义为“类型（Type）”的东西的、和智能感知的脚本。类型通常被用于数据类型（例如整形或者字符型）和基础类（例如复选框和表单）。不过，你可以把其它东西定义为类型，或者通过手工添加一条 Type 字段为 "T" 的记录、或者通过使用工具菜单中的智能感知管理器。

其它类别的类型记录通常是自定义类或者 COM 对象，让你可以在智能感知中用上它们。这也是我们将使用类型记录的目的所在，不过我们还将通过使用一个脚本和一个自定义智能感知处理类来自己定制让智能感知怎么工作。

如果你使用 FFIBUILDER 表单注册了一个类以后再去看看你的智能感知表（USE (_FOXCODE) AGAIN 和 BROWSE），你将看到在表的末尾有两条新记录。一条是类的类型记录，其中除了你在生成器表单中定义的名称空间、和一个让智能感知应该为这个类使用的脚本的名称（它在 CMD 字段中指定为一个由中括号包起来的名称）以外，并没有包含多少信息。另一条记录是一条脚本记录，它的 TYPE 字段的值为“S”而 ABBREV 字段中则包含着在 T 记录的 CMD 字段中所指定的名称。该名称的格式是 FFI_ 再加上一个 SYS(2015) 产生的值。由于这个字段太短，所以没有足够的空间让我们可以使用一些象“Namespace_Script”这类更有意义的名称。

这条脚本记录的备注字段 DATA 中有着以下代码（在这段代码中，Path1 会被替换成 FFI.VCX 所在的路径，而 Path2 则会被替换成要注册的类所在类库的路径）：

```
lparameters toFoxCode
local loFoxCodeLoader, luReturn
if file(_codesense)
    set procedure to (_codesense) additive
    loFoxCodeLoader = createobject('FoxCodeLoader')
    luReturn = loFoxCodeLoader.Start(toFoxCode)
    loFoxCodeLoader = .NULL.
    if atc(_codesense, set('PROCEDURE')) > 0
        release procedure (_codesense)
    endif atc(_codesense, set('PROCEDURE')) > 0
else
    luReturn = ''
endif file(_codesense)
return luReturn

define class FoxCodeLoader as FoxCodeScript
```

```
    cProxyClass = 'FFIFoxCode'
    cProxyClasslib = 'Path1\ffi.vcx'

    procedure Main
        local loFoxCode, luReturn
        loFoxCode = newobject(This.cProxyClass, This.cProxyClasslib)
        if vartype(loFoxCode) = 'O'
            luReturn = loFoxCode.Main(This.oFoxCode, ;
                'ConnectionMgr', 'Connectionmgr', ;
                'Path2\connmgr.vcx')
        else
            luReturn = ''
        endif vartype(loFoxCode) = 'O'
        return luReturn
    endproc
enddefine
```

这段代码定义了一个“定义在由 _CODESENSE 系统变量指定的智能感知应用程序中的 FoxCodeScript 类”的子类。这个子类覆盖了 Main 方法（该方法由智能感知来调用）来建立位于 FFI.VCX 类库中的 FFIFoxCode 类的实例，并调用它的 Main 方法。这个调用会传递一个对智能感知数据对象（它包含这关于用户输入了什么、以及其它智能感知设置的信息）的引用、用户正在操作的名称空间、以及该名称空间的类和类库。

作为这个脚本的结果，在所有针对这个名称空间的智能感知任务中都会调用 FFIFoxCode.Main 方法，例如，当你输入 LOCAL SomeVariable AS 的时候从显示出来的智能感知列表中选择这个名称空间、或者你在一个“包含了由该类实例化成的变量名称”的语句中输入了某个“触发器”字符——例如一个句号、一个左括号、或者一个等号——的时候。

FFIFoxCode

FFIFoxCode 类为 FFI 完成所有的自定义智能感知工作，现在我们来看一下这个类的细节。它的 INIT 方法做了两件事情：

- 在系统部件中打开调试（不这么做的话，你就不能轻松的调试代码中的问题）。
- 通过调用 OpenFFITable 方法来打开 FFI.DBF（包含一个类及其部件注册信息的表）。如果这个表不能被打开，那么 INIT 会显示一个错误消息，并返回 .F.，如此一来则不会建立这个类的实例。

```
* 打开调试.
sys(2030, 1)
```

```
* 打开 FFI (Favorites for IntelliSense) 表
local llReturn
llReturn = This.OpenFFITable()
if not llReturn
    messagebox('Could not open the Favorites for ' + ;
        'IntelliSense table.', 64, ;
        'IntelliSense Handler')
endif not llReturn
return llReturn
```

从智能感知脚本中调用的 **Main** 方法为 **FFI** 处理所有的智能感知任务。如我们前面所见，这个脚本会把一个 **FoxCode** 对象、类的名称空间、以及这个类和它的类库传递给 **Main**。如果在 **FoxCode** 对象的 **MenuItem** 属性中找到了这个名称空间，那么触发智能感知的就一定是在一个 **LOCAL SomeVariable AS** 语句中，因此 **Main** 就调用 **HandleLOCAL** 方法来对此进行处理。此外，**Main** 还会判定是哪个字符触发了智能感知，并调用 **GetFFIMember** 方法来判定你所输入的是类的哪一个成员（也可以是这个类本身），并返回一个从 **FFI** 表的相应记录中 **SCATTER Name** 来的对象。

如果触发字符是一个句号，我们需要去显示一个该类的注册了的成员的列表，所以 **Main** 就调用 **DisplayMembers** 来完成这个工作。如果触发字符是一个左括号、一个等号、或者一个逗号，并且在 **FFI** 记录的备注字段 **SCRIPT** 中有代码，那么这块代码就被运行。这段代码可以是智能感知应该为一个属性值显示的枚举值的列表、或者一个方法的参数（稍候我们会举这么一个例子）。

最后，如果 **FFI** 记录的备注字段 **TIP** 被填充过了，**Main** 将把它用作智能感知的 **ToolTip**。它通常被用于显示一个方法的特征（例如：**Login(UserName as String, Password as String) as Boolean”**）。

```
lparameters toFoxCode, ;
    tcNameSpace, ;
    tcClass, ;
    tcLibrary
local lcReturn, ;
    lcTrigger, ;
    loData

with toFoxCode

* 如果是在 LOCAL 语句中，就通过返回我们想要插入的文本来处理它

lcReturn = ''
```

```
    if atc(tcNameSpace, .MenuItem) > 0
        lcReturn = This.HandleLOCAL(toFoxCode, tcNameSpace, tcClass, tcLibrary)

    * 取得触发智能感知的字符，
    * 并弄清楚用户输入的是哪个成员
    else
        lcTrigger = right(.FullLine, 1)
        loData = This.GetFFIMember(.UserTyped, tcClass)
        do case

        * 如果没法在 FFI 表中找到这个成员，就什么也不做
        case vartype(loData) <> 'O'

        * 如果是由一个 "."触发智能感知的，
        * 就显示一个成员列表
        case lcTrigger = '.'
            This.DisplayMembers(toFoxCode, loData)

        * 如果是由一个 "(" (开始输入一个方法的参数列表)、
        * 一个 "=" (用于一个属性)，或者一个 ","（开始输入一个新的参数）
        * 触发的智能感知，并且我们有一个脚本，就运行这个脚本
        case inlist(lcTrigger, '=', '(', ',') and not empty(loData.Script)
            lcReturn = execscript(loData.Script, toFoxCode, loData)

        * 如果是由一个 "(" 或者 ","触发的智能感知，
        * 则为该方法显示 Tooltip (通常是该方法的特征)。
        case inlist(lcTrigger, '(', ',') and not empty(loData.Tip)
            .ValueTip = loData.Tip
            .ValueType = 'T'
        endcase
    endif atc(tcNameSpace, .MenuItem) > 0
    endwith
    return lcReturn
```

这里我不准备讲 **HandleLOCAL** 方法的代码了，有兴趣的话可以自己去看。**Main** 在你输入 **LOCAL SomeVariable AS** 的时候 调用这个方法，并从类型列表中选择一个注册了的名称空间。它所做的一切就是为这个类生成一个 **NEWOBJECT()**语句，这样你就不用再输了。这个方法中唯一困难的问题在于为 **NEWOBJECT()**使用的大小写格式（如果只考虑我自己的话，由于我总是为 **FoxPro** 关键词使用小写，也许就写成“**newobject**”了，但考虑到“别的程序员也许对大小写有自己的看法”总是好事情）。这是通过在智能感知表中查找一条 **TYPE="F"**（用于函数）和 **ABBREV="NEWO"(NEWOBJECT** 的缩写**)**的记录、并使用该记录中 **CASE** 字段的值来解决的。

从 **Main** 中被调用的 **GetFFIMember** 向 **FFI** 表中搜索你输入的类或者成员。它使用 **FoxCode** 对象的 **UserTyped** 属性（被作为第一个参数传递），这个属性中包含着你输入的属于该名称空间的文本。例如，当你输入“**llStatus=ConnMgr.ExecuteSQLStatement("**”的时候，**UserTyped** 属性中包含着的就是“**ConnMgr.ExecuteSQLStatement**”。

GetFFIMember 通过在 **FFI** 表中查找指定类的记录来开始。如果在 **UserTyped** 属性中包含有一个句号，那么它就会去查找相应的成员记录。如果它找到了正确的记录，它就会从那条记录返回一个 **SCATTER NAME** 来的对象。

```
lparameters tcUserTyped, ;
    tcClass
local loReturn, ;
    lcUserTyped, ;
    llFound, ;
    lnPos, ;
    lcMember, ;
    lnSelect

* 取得用户所输入的东西。如果它是以一个左括号结尾的，
* 就去掉它

loReturn = .NULL.
lcUserTyped = alltrim(tcUserTyped)
if right(lcUserTyped, 1) = '('
    lcUserTyped = substr(lcUserTyped, ;
    len(lcUserTyped) - 1)
endif right(lcUserTyped, 1) = '('

* 找到该类在 FFI 表中的记录
* 如果在输入的文本中有一个句话，
* 就试着去找到该成员的记录

if seek(upper(padr(tcClass, len(__FFI.CLASS))), ;
    '__FFI', 'CLASS')
    llFound = .T.
    lnPos = at('.', lcUserTyped)
    if lnPos > 0
        lcMember = alltrim(__FFI.MEMBER) + ;
        substr(lcUserTyped, lnPos)
        llFound = seek(upper(padr(lcMember, ;
        len(__FFI.MEMBER))), '__FFI', 'MEMBER')
    endif lnPos > 0
```

```
    * 如果我们找到了期望的记录，
    * 就为它建立一个 SCATTER NAME 的对象

      if llFound
          lnSelect = select()
          select __FFI
          scatter memo name loReturn
          select (lnSelect)
      endif llFound
    endif seek(upper(padr(tcClass ...
    return loReturn
```

当你在命令行中输入一个句号的时候，**DisplayMembers** 从 **Main** 中被调用，以告诉智能感知去显示一个该类的注册了的成员列表。**DisplayMember** 调用 **GetMembers** 去返回一个指定类的成员的集合（这里就不讨论这个方法的代码了）。然后它会以那些成员的 **names** 和 **descriptions** 去填充 **FoxCode** 对象的 **Items** 数组，并将这个对象的 **ValueType** 属性设置为“**L**”，这样做就表示让智能感知去显示一个其内容就是 **Items** 数组的列表框。

这段代码表现出了智能感知设计上的一个轻微缺陷：**FoxCode** 对象只有一个 **Icon** 属性，其中包含着将显示在列表框中的图像文件的名称。但事实上，我们需要的是在 **Items** 数组中再多加一个列，因为在这种情况下，我们需要为属性和方法显示不同的图像。不幸的是，我们只有一个图像可以供所有的数据项显示：

```
    lparameters toFoxCode, toData
    local loMembers, lnI, loMember

    with toFoxCode

      * 取得一个当前类的成员的集合

      loMembers = This.GetMembers(alltrim(toData.Member))

      if loMembers.Count > 0

          * 将每个成员添加到 FoxCode 对象的 Items 数组
          dimension .Items[loMembers.Count, 2]
          for lnI = 1 to loMembers.Count
              loMember = loMembers.Item(lnI)
              .Items[lnI, 1] = loMember.Name
              .Items[lnI, 2] = loMember.Description
              if loMember.Type = 'P'
                  .Icon = home() + 'ffc\graphics\propty.bmp'
              else
```

```
                    .Icon = home() + 'ffc\graphics\method.bmp'
                endif loMember.Type = 'P'
            next loMember
            * 将 FoxCode 对象的 ValueType 属性设置为 "L",
            * 表示显示一个包含着定义在 Items 数组中元素的列表框
            .ValueType = 'L'
        endif loMembers.Count > 0
    endwith
```

一些高级用法

你可以对 **FFI** 做一些有趣的事情，以使得它更容易被使用。其中一件，是那些类可以包含一个对象的层次，并且通过在 **FFI** 中将这些类注册为一个类的成员，你可以获得在这些成员上的清楚的智能感知。例如，**cApplication** 类在运行时把一个 **cUser** 类的实例建立到它自己的 **User** 属性中去。

通常，你无法在 **User** 属性上获得智能感知，因为这个属性在设计时并没有包含着一个对象。不过，如果你打开 **cUser** 类，并指定“**Application.User**”作为它的名称空间，那么，你就获得了在 **Application** 名称空间中的 **User** 成员的智能感知，如图 **5** 所见：

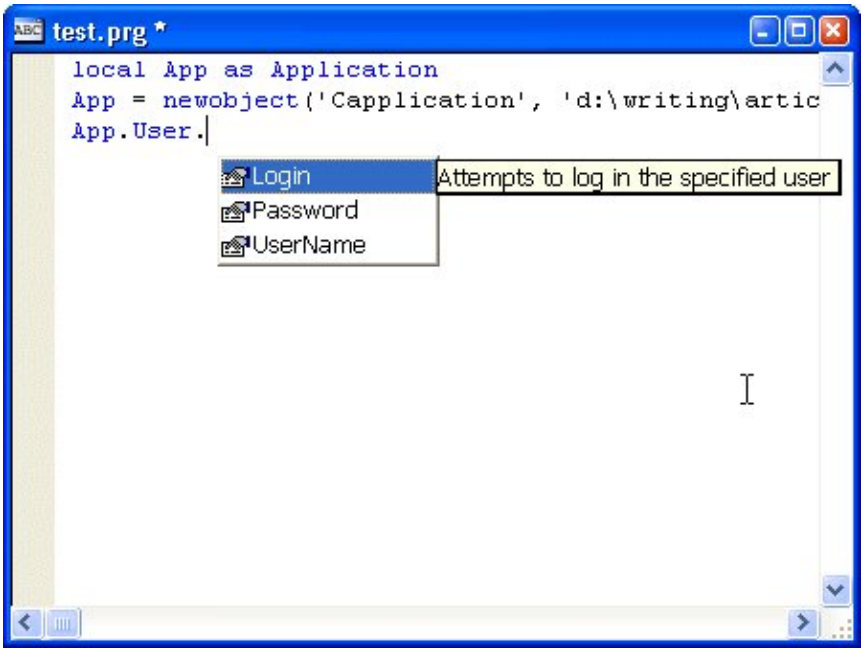

图 5、你甚至可以获得对在运行时才被实例化的对象的智能感知，只要它们被注册在主名称空间的一个子名称空间中。

另一件有用的事情，是通过在 **FFI** 生成器表单的 **Script**（脚本）编辑框中输入代码来指定让智能感知如何处理一个成员。例如，**ConnectionMgr** 的 **DatabaseSpecifier** 属性指定数据库应该如何被指定在一个 **ODBC** 连接字符串中。对于某些数据库，例如 **Access** 和 **dBase**，你得使用“**dbp=database path**”这样的格式，而对于另一些数据

库，则需要指定“database=database name”。

不用指望程序员知道为 DatabaseSpecifier 正确的选择“database”还是“dbp”（并且不会把它们输错），你可以用与显示一个指定的成员列表类似的方式告诉智能感知去显示一个有效值的列表：通过填充 FoxCode 对象的 Items 数组、并将它出 ValueType 属性设置为“L”。下面的 DatabaseSpecifier 的脚本就是这么做的。

注意，所有的脚本都会接收到两个参数：一个对 FoxCode 对象引用、和一个来自 FFI 记录的 SCATTER NAME 对象。还必须注意的是，如果包含在列 1 中的值是一个字符串，则这个值中还必须包含着引号，否则它们就不会被正确的插入到代码中去。最后，注意这个脚本的返回值被发送给智能感知，所以，除非你想让这个值被插入到代码中，否则的话就应该返回一个空字符串、并将 ValueType 设置为除了“V”以外的什么东西。

```
lparameters toFoxCode, toData
dimension toFoxCode.Items[2, 2]
toFoxCode.Items[1, 1] = '"database"'
toFoxCode.Items[1, 2] = 'Used for most databases'
toFoxCode.Items[2, 1] = '"dbq"'
toFoxCode.Items[2, 2] = 'Used for Access and dBase'
toFoxCode.ValueType = 'L'
return ''
```

这里是一个脚本，用于为一个方法的某些参数显示一个上下文敏感的 ToolTip 、以及为其中的一个参数显示一个有效值列表：

```
lparameters toFoxCode, toData
local lcMember, ;
    lnPos, ;
    lcParameters, ;
    lnParameters
lcMember = alltrim(substr(toData.Member, rat('.', toData.Member) + 1))
lnPos = atc(lcMember, toFoxCode.FullLine)
lcParameters = substr(toFoxCode.FullLine, lnPos + len(lcMember))
lnParameters = occurs(',', lcParameters) + 1
do case
    case lnParameters = 1
        toFoxCode.ValueTip = 'First parameter tooltip'
        toFoxCode.ValueType = 'T'
    case lnParameters = 2
        toFoxCode.ValueTip = 'Second parameter tooltip'
        toFoxCode.ValueType = 'T'
    case lnParameters = 3
        dimension toFoxCode.Items[2, 2]
        toFoxCode.Items[1, 1] = '1'
        toFoxCode.Items[1, 2] = 'Description for value 1'
```

```
            toFoxCode.Items[2, 1] = '2'
            toFoxCode.Items[2, 2] = 'Description for value 2'
            toFoxCode.ValueType = 'L'
        endcase
```

总结

通过定制智能感知工作的方式，现在我们让 **VFP** 中生产率最佳的工具变得更强大了。使用 **FFI**，可以让你对常用类（或者甚至包括那些不常用的类、不熟悉的类）的工作变得更加得轻松。智能感知只显示那些你想要看到的成员，可以在那些运行时才被实例化的成员上提供智能感知、可以为枚举属性或者方法的参数提供有效值的列表。

特别感谢微软 **Visual FoxPro** 的程序管理员 **Randy Brown**，在他的指点下我才弄清楚了如何让智能感知只显示你想要的成员。

附件：**508HENNIG.ZIP**

在 VFP 应用程序里发送 SMTP 消息，第二部分

原著：Anatoliy Mogylevets

翻译：CY

上月，Anatoliy Mogylevets 展示了如何在 VFP 里创建 SMTP 电子邮件信息。本月，第二部分，他将详细解释如何连接到 SMTP 服务器以发送你所创建的信息，并提供一个 SendEmail 类库，可以让你用于你自己的应用程序里。

如果你读过我在 **FoxTalk 2.0** 六月的文章，你将会知道如何创建 **SMTP** 电子邮件信息，加入附件，指定格式，等等。现在让我们来看看通过与 **SMTP** 服务器通讯来发送信息。

现在，我将假装 **SMTP** 服务器是个黑盒子，已经连接到我的 **VFP** 应用程序。它将接受命令和数据，并作出响应。同样，我也假装有一个含三个成员的虚拟类，如表 **1**：

表 1：一个简单的电子邮件发送类的三个成员

方法	描述
SendCommand(cCommand As String)	发送命令到 SMTP 服务器
SendData(cData As String)	发送数据字符串到 SMTP 服务器
GetResponse() As String	从 SMTP 服务器接收响应

我发送给服务器的第一个命令是问候：

```
THIS.SendCommand("HELO " + GETENV("COMPUTERNAME"))
```

现在我将等待响应：

```
cResponse = THIS.GetResponse()
```

通常，邮件服务器会立即作出响应：

```
250 smtp5.emailserver.net
```

返回的字符串包含 **SMTP** 响应代码和响应的描述或一些其他如 **SMTP** 服务器名的信息。**SMTP** 代码 **250** 表示“**OK**，你可以继续。”我将继续发送另一个命令：

```
THIS.SendCommand("MAIL FROM: sender@somedomain.com")
cResponse = THIS.GetResponse()
```

一旦我接收到响应是以“**250**”打头，我将发送下一个命令：

```
THIS.SendCommand("RCPT TO: <somebody@somewhere.com>")
cResponse = THIS.GetResponse()
```

如果我想发送的一个信息要交付给多个接收人，我将发送“**RCPT TO**”命令的邮件地址，无论它是 **To:**或是 **Cc:**或是 **Bcc:**。下一个命令是“**DATA**”。它指明我将发送一个信息：

```
THIS.SendCommand("DATA")
cResponse = THIS.GetResponse()
```

这时服务器响应为不同的代码：

```
354 Ok Send data ending with <CRLF>.<CRLF>
```

这个响应意味着服务器已经准备好接收信息。同样，它提醒我以一个被两个 **CRLF**（回车换行）包围的点号来结束数据传送，这是所有服务器的标准。因为如此，我将不得不确保我的信息里不包含这个控制符。如果有，我将对它作改变。

```
TEXT TO cData NOSHOW
From: <sender@somedomain.com>
To: <receiver@somedomain.com>
Cc: <another_receiver@somedomain.com>
Subject: Test message
Date: Sun, 29 May 2005 21:50:44 -0400
MIME-Version: 1.0
Content-Type: text/plain;
charset="Windows-1252"
Content-Transfer-Encoding: 7bit

This is a test message. Please do not respond.
ENDTEXT

THIS.SendData(m.cData)
cEndOfTransfer = Chr(13) + Chr(10) + "." + Chr(13) + Chr(10)
THIS.SendCommand(m.cEndOfTransfer)
cResponse = THIS.GetResponse()
```

信息里“**From:**”、“**To:**”和“**Cc:**”头并不真正影响它所要分发的去向。“**RCPT TO**”命令定义了所有一切。这是垃圾邮件常用的老诡计：他们假装或是忽略了这些头。这就是为什么你有时会接收到“未知的接收人”的信息。正确的做法是匹配每一个“**RCPT TO**”命令于“**To:**”和“**Cc:**”头。当然，不要伪装“**From:**”头。某些 **SMTP** 服务器或许不会认为这是个玩笑，你将会遇到麻烦。

当数据传送结束时，我将接收到另一个“**250**”响应。这包含有由服务器为我的信息所分配的一个唯一 **ID** 号。当不再发送信息时，我发送给服务器的最后一个命令是“**QUIT**”：

```
THIS.SendCommand("QUIT")
cResponse = THIS.GetResponse()
```

其他 SMTP 命令将不在本文中讲述了。

The Winsock

现在该是承认 SMTP 服务器不是真正的黑盒子的时候了。为了与 SMTP 服务器建立连接，等同于 HTTP、FTP 和 NNTP 服务器，VFP 应用程序创建了套接字。实际上，这是另一个级别的抽象，于是我们并不完全抵制黑盒子的观点，但还是让我们继续吧。

可以认为套接字只是一个远程服务器的入口。这个入口有两个参数：服务器的 IP 地址和端口号。IP 地址，是被点号分割的四个数字的字符串，表示着在互联网上的一台计算机。如果你有，它将是你的邮局街道地址。

端口号就象是墙上你要发送信息的一个小窗口。在每个服务器上大约有 65535 窗口。其中许多是关闭或隐藏的。在某些打开的窗口里，雇员根本不对你说话，或者是纠缠不清的，“服务器不可用”。别冒犯，一旦来到#25 窗口，他们正等着你。

在大多数服务器上，端口 25 是保留给 SMTP 连接的。为开始与隐藏在计数器后面的人说话，要产生一个 ID。第一个 ID 所检测的是你的计算机的 IP 地址。SMTP 通常只接受某个范围的 IP 地址，而并不是任意的 IP 地址。在另一方面，互联网提供商并不喜欢从世界每个人转发电子邮件。某些服务器要求更多的 ID，这个过程叫 SMTP 认证，但它不在本文讨论中。

一旦你从服务器获得认可，它将不再只是套接字，你获得了套接字连接。这是个通道，它已经准备好在你的计算机和 SMTP 服务器间来回传送数据。Windows 操作系统提供了一个编程接口，可以让编程人员访问类似 SMTP 的网络服务。它就叫 Winsock(Windows 套接字)。所有的事都被封装在 ws2_32.dll 里，默认安装于每一个 Windows 机器里。你可以在你的计算机的 System32 目录里找到这个文件。

应用程序经过 Winsock 接口就可以与任何 SMTP 服务器进行对话，而不管它的操作系统，只要它支持同样的协议。在 VFP 里，你可以通过申明和调用它的函数来访问 Winsock。同样，你可以利用 MS Winsock 控件，把你不得不要作的复杂编程抛出你的视线。

利用 Winsock API 来编程与 SMTP 服务器连接并交换数据

为实现这个任务，我开发了 TCP_socket 类。这个类创建了套接字：

```
THIS.hSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
```

然后它打开对远程服务器的连接：

```
PROCEDURE ConnectTo(ipaddress As String, nPort As Numeric) As Boolean
#INCLUDE winsock.h
```

```
LOCAL nHost, cBuffer
nHost = inet_addr(m.ipaddress)

cBuffer = THIS.num2word(AF_INET) +;
    THIS.num2word(htons(m.nPort)) +;
    THIS.num2dword(m.nHost) +;
    Repli(Chr(0),8)

THIS.errorcode = ws_connect(THIS.hSocket, @cBuffer, Len(cBuffer))

IF THIS.errorcode = 0 && success
    THIS.ipaddress=m.ipaddress
    THIS.port=m.nPort
    THIS._connected=.T.
ELSE
    THIS._connected=.F.
ENDIF
RETURN THIS._connected
```

ConnectTo 方法接收两个参数:远程服务器的 **IP** 地址和远程服务器上的端口号，邮件服务器通常保留端口 **25** 给 **SMTP** 连接。类的另一个核心方法是 **Send**。它通过套接字连接来发送数据字符串到服务器。字符串可以是命令或是邮件信息。

```
PROCEDURE Send(cBuffer As String)
PARAMETERS cBuffer
#INCLUDE winsock.h

IF THIS.connected()
    LOCAL nResult
    nResult = send(THIS.hSocket, @cBuffer, Len(cBuffer), 0)
    RETURN (nResult <> SOCKET_ERROR)
ELSE
    RETURN .F.
ENDIF
```

Recv 方法从邮件服务器接收响应。

```
PROCEDURE Recv() As String
LPARAMETERS nBufsize
IF NOT THIS.connected()
    RETURN ""
ENDIF

IF VARTYPE(nBufsize) <> "N"
    nBufsize = 0x4000
```

```
ENDIF
LOCAL cRecv, nRecv, nFlags
cRecv = REPLICATE(Chr(0), m.nBufsize)
nFlags = 0

nRecv = recv(THIS.hSocket, @cRecv, m.nBufsize, m.nFlags)
RETURN IIF(nRecv<=0, "", LEFT(cRecv, nRecv))
```

Recv 是被封装在另一个方法里，它包含有一系列的 **API** 调用：**WSACreateEvent**，**WSAEventSelect**，**WSAWaitForMultipleEvents**，和 **WSACloseEvent**。他们提供了所谓的模块化模式，当一上应用程序控制了套接字并把它自己放入等待状态。这个状态将一直保持，一直到服务器的响应到来或是产生超时。这里就是它如何的编程：

```
hEventRead = WSACreateEvent()

= WSAEventSelect(THIS. hsocket, hEventRead, BITOR(FD_READ, FD_CLOSE))

nWait = WSAWaitForMultipleEvents(1, @hEventRead, 0, m.nTimeout, 0)

= WSACloseEvent(hEventRead)
```

SendEmail 类库

本文的下载里包含有数个 **VFP** 类。利用它们可以创建无格式和多节信息，可以利用 **SMTP** 服务器和 **Hotmail** 服务器来发送。这些类和描述如表 **2** 所示。

表 2：包含在本文下载文件 SendEmail 类库里的电子邮件类

类	描述
Email_address	邮件地址的实现。用于在 MSG 类作为发送人和接收人（集合）。
Email_header	邮件信息头，放置于头或信息体内。用于 MSG_BASE 类作为头（集合）。
Mailgate	基本邮件网关类。实现到邮件服务器的连接。不要实例化这个类。
Mailgate_hotmail	实现到 Hotmail 和 MSN 服务器的连接。
Mailgate_smtp	Winsock 邮件网关的实现。
Mailgate_stub	实现一个虚拟邮件网关。不要发送邮件，仅供测试。
Msg_base	基类。实现邮件信息的头和节
Msg	邮件信息。
Msg_part	邮件的节，如无格式文本的节或 HTML 节或附件。
Tcp_socket	TCP 套接字的实现。

下列示例展示了如何利用这个类库来发送带附件的无格式文本信息：

```
SET CLASSLIB TO SendEmail ADDITIVE
#DEFINE CRLF CHR(13)+CHR(10)
#DEFINE DCRLF CRLF+CRLF

LOCAL msg As msg, gate As mailgate
```

```
* 建立一封邮件
msg = CREATEOBJECT("msg")
WITH msg
    .subject = "Test message"
    .setsender("user@mydomain.com", "User")
    .addrecipient("admin@mydomain.com", "Admin")
    .addpart("This is a test message.", "text/plain", "7bit")
    .addattachment("c:\data\somedata.zip")
ENDWITH

* 使用一个有效的 smtp 服务器来建立一个邮件网关
gate = CREATEOBJECT("mailgate_smtp", "smtp8.mydomain.com")

IF NOT gate.isconnected()
    = MESSAGEBOX("Error: " +;
        TRANSFORM(gate.errorcode) +;
        ". " + gate.errormsg + " ",;
        48, "SendEmail class library")
    RETURN
ENDIF

IF NOT gate.sendmessage(m.msg)
    = MESSAGEBOX("Error: " +;
        TRANSFORM(gate.errorcode) +;
        ". " + gate.errormsg + " ",;
        48, "SendEmail class library", 5000)
ELSE
    = MESSAGEBOX("Email sent successfully! ",;
        64, "SendEmail class library", 5000)
ENDIF
gate=Null
```

附件：**508ANATOLIY.ZIP**

漂亮的查找方法

原著：Andy Kramek 和 Marcia Akins

翻译：CY

在运行时向组合框加入值是很普通的要求。然而，当列表项的源为表时管理它却是个难事，因为组合框只允许数据被加入到单列里。当然，解决的办法是，使用弹出表单来处理编辑的要求，但把它全部集成到一个普通的类来做这些事。本月，Andy Kramek 和 Marcia Akins 将紧紧围绕这个令人头痛的小问题进行处理并带来一些现成的普通类。

Andy：这有更好的办法来做这事。我正在做一个可广泛应用于表查找的程序，其实，是一个带有 21 个分别独立查找的表单。每个都需要一个组合框都可以让用户在运行时对其增加、甚至编辑。它将会是大量的工作。

Marcia：听起来你需要一个类来处理它。首先我们需要知道你所查找的数据是如何存储的。你给这些值分别使用单独的表吗？

Andy：当然不！任何人陷入泥潭都是无能为力的。我有一对表（名为 **Lookup_Hdr** 和 **Lookup_Dtl**），它对实际值作了定义并按种类分组。图 1 展示了结构。

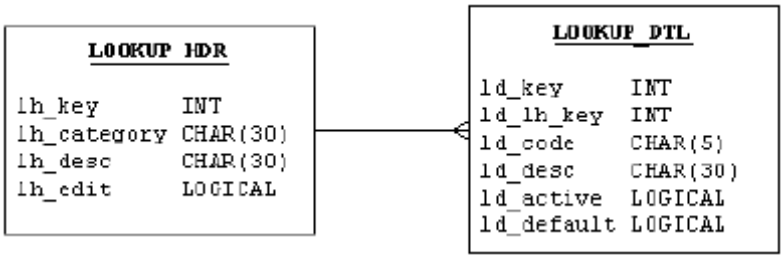

图 1：Lookup_Hdr 和 Lookup_Dtl。

Marcia：这当然会更容易些。因为我们使用通用代码来找回所要查找的数据，并增加或更新这个实际值。我们应该知道任何的组合框将要被处理的类型。

Andy：但是现在我们可以允许加入一个新值吗？你可以在组合框里输入一个值，并且查找的明细表可能会超过一列。我猜想主键和外键就足够了，但是在明细表里有两个代码和描述列，加上标记！

Marcia：这是组合框的基本问题。仅当组合框的数据只包含一列时，在运行时才可以加入

新项目。这发生在组合框是发布为确定的值，通过使用 **AddItem()**或是在代码里创建数组。一旦你有一个棘手的表，加入新值的用处就会受到限制。更糟糕的是，尽管组合框可以让你加入新元素，却没有提供方法来编辑已存在的条目（比如，校正一个拼写错误的条目）。

Andy：这就是你所说的—我们无法让用户增加或编辑？

Marcia：我什么时候说过？我想，组合框类可以很漂亮的弹出一个小小的数据输入表单，来处理在查找表中增加，或编辑内容。我猜想，创建新的类目是通过应用程序的维护过程来处理的，通过组合框来做这事是没有意义的。

Andy：对的，只是我需要用户可以编辑那些值—那些类目已经在其他地方处理了。因此我该怎么来弹出表单？是一个右键菜单？

Marcia：我们可以的，但这并不是组合框的通常行为。它或许对你的用户是更明显的，如何我们把组合框放入一个带有增加和编辑按钮的容器（如图 **2** 所示）。显然它是可以做到的。

图 2：查找组合框容器

Andy：我还不太有把握。我是看得很明白，但是还记得我的问题吗？我不得已在一个表单里有 **21** 个这样的东西。我将会把它变小以塞得满满的，而且所有的这些活动按钮肯定会引起问题。我想我不愿意在已经拥挤的表单上加入任何的其他控件，除非没有其他办法来实现。

Marcia：好吧。我们当然可以以你想要的办法来实现。那你喜欢做什么？从组合框的右键弹出上下文敏感菜单，并带有增加，编辑和删除查找内容的选项？

Andy：绝对不作删除！如果我们允许最终用户在运行时删除内容，将会有太多的数据完整性问题要处理。我打算把删除选项保留在查找维护表单里。他们可以标记一个项目为“非活动”—但那只是编辑表单的部分。

Marcia：因此第一件事就是创建一个 **Style=2**—下拉列表类型的新组合框类。由于我们要使用一个弹出表单，我们不需要用户直接输入到组合框。利用下拉列表可以简化其他事情。

Andy：比如哪些？

Marcia：好了，下拉列表的 **Valid()**仅当在用户在真正做了选择时才引发。但是，如果类型为 **0**，下拉组合框，**Valid()**在控件失去焦点时就会引发—甚至当用户只是用 **Tab**

键跳过控件时。这在某些情况下会产生问题，但是我们却是不需要担心的。

Andy：我喜欢这样。少担心是好的。因此我们要下拉列表。那现在呢？

Marcia：首先，让我们来考虑如何组装列表。我们打算在组合框上来使用一个数组属性。

Andy：为什么使用数组来代替游标？

Marcia：它这样更容易处理禁止项目。然后，如果用户查看的数据包含指向当前已经关闭的选项，他们将可以看到，即使是不被使用的。

Andy：我还没想过这些。我只是简单的猜想，我们将选择一个值标记为活动。你是对的，那样在查看旧数据时会不太好。

Marcia：我们所必须要知道的是我们要显示的是哪种查找类目，并且我们已经加入属性来保存它。然后就只是写一个简单的 **SQL** 查询来获取相应的数据：

```
SELECT LUD.ld_key, LUD.ld_code, LUD.ld_desc, ;
       LUD.ld_default, LUD.ld_active, LUD.ld_lh_key ;
  FROM lookup_dtl LUD, lookup_hdr LUH ;
  WHERE LUD.ld_lh_key = LUH.lh_key AND LUH.lh_category = This.cCategory ;
  INTO ARRAY This.aItems
```

Andy：我还不太有把握。通过在类里包含象这样的一个查询，我们把组合框与一个表绑定在一起，是吗？

Marcia：是的。这就是整个思路。组合框被用来以最少的设置来获取正确的数据。为减少设置它需要知道将使用哪个表。

Andy： 我明白了。我们可以创建一个参数视图吗（带有标准化的结构和完整的字段名）？然后那个类就根据它自己的类目属性来并设置那个视图参数，并查询视图以代替原先的表。现在，如果我们也加入视图名来作为第二个属性来使用视图，我们就可以用这个组合框类来使用任何的数据集作为它的数据源，假定视图存在有所需的列名。然后在类里查询时可以这样：

```
SELECT lu_desc, lu_code, lu_active, lu_default, lu_pk, lu_cat_fk ;
  FROM ( lcViewName ) INTO ARRAY This.aItems
```

难道不够通用吗？

Marcia：是的，我猜想它是可行的。最后，那个视图可以定义可更新列，就象我们先前直接使用表那样，它就会变得更通用了。我喜欢这样。

Andy：好的。因此我们有两个参数，但代码在什么时候来建立列表？在组合框的 **Init()**吗？

Marcia：那样可以运行，但因为我们一直这样说，“代码在方法里，而不是在事件里”。我更喜欢创建一个自定义的名为“**PopulateCombo**”的方法，并让它从 **Setup()**方

法里被调用（它自己是从 **Init()**里被调用）。那样，无论我们何时需要查询组合框（比如，在用户加入或编辑了值后），我们只需要调用 **PopulateCombo()**方法。那样比调用 **Init()**和 **Setup()**更容易明白！这里是代码：

```
LOCAL vp_category, lcViewName

*** 从属性中取得视图的名称
lcViewName = This.cViewName

*** 从类别属性中取得视图参数
vp_category = This.cCategory

*** 确保该视图被打开
IF NOT USED( lcViewName )
    USE ( lcViewName ) AGAIN NODATA IN 0
ENDIF

*** 刷新视图以取得指定类别的数据项
REQUERY( lcViewName )

*** 将这些数据项转换到该类的内部数组中去
SELECT lu_desc, lu_code, lu_active, lu_default, lu_pk ;
    FROM ( lcViewName ) INTO ARRAY This.aItems

*** 扫描这个数组，并禁用那些不活动的数据项
FOR lnItem = 1 TO ALEN( This.aItems, 1 )
    IF NOT This.aItems[lnItem,3]
        This.aItems[lnItem, 1] = '\ '+ This.aItems[lnItem,1]
    ENDIF
ENDFOR

*** 最后，重建列表
This.Requery()
```

Andy：那是非常简单易懂的。

Marcia：是的，使用视图确实使得类更通用些。当然，我也加入代码到组合框的 **Setup()**方法里以检查视图名和类目属性是否有确定值。 **SetDefaultValue()**里有更详细的代码来处理设置组合框的默认值，如果是还没有定义的，就默认为列表里的第一个。

Andy：但是我没有在类见到 **SetDefaultValue()**被调用。它是怎么工作的？

Marcia：这所有的处理都是利用 **BindEvent()**（关于它如何实现的更多细节，参见 **2004.02** “它不是我的错，它是我的默认”）。

Andy：最后一个问题是在我们要编辑前。我如何告诉组合框在何处存储值？在某些情况下，我需要存储代码（比如州），同时在另一方面我需要存储查找明细表的外键。

Marcia：仅需要设置组合框的 **BoundColumn** 到数组里适当的列。在默认情况下它为 **5**（它是查找明细表的主关键字）并且把 **BoundTo** 设置为**.T.**以处理那样的假设。但是如果你需要代码，仅需要设置 **BoundColumn=2**。

Andy：但不会因为 **BoundTo** 设置为**.T.**导致出错？

Marcia：不，它是另一个方法。如果你想要从组合框返回一个实际的数值，你必须设置 **BoundTo** 为**.T.**。但是如果值是字符串，无论 **BoundTo** 如何设置，它都不会有不同。图 **3** 展示了查找组合框用于一个包含有查找州的表单。

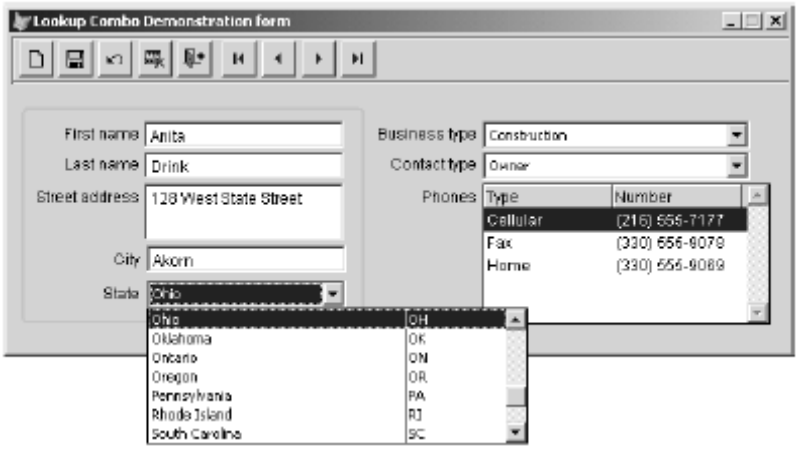

图 3：示例表单展示了通用的组合框应用。

Andy：我明白你使用了在电话号码表格里同样的类。这很好，真的。因此我们知道它可以同样工作于表格内。那编辑弹出菜单和右键菜单怎样？

Marcia：右健菜单是很简单的。我们所要做的只是加入一个名为 **ShowMenu()**方法到类里，并从 **RightClick()**里调用它。基本上，我们全部所做的是弹出一个带有两个选项的小菜单，“增加”和“编辑”，并且返回用户选择的选项号（例如，**1** 或 **2**），如图 **4** 所示。

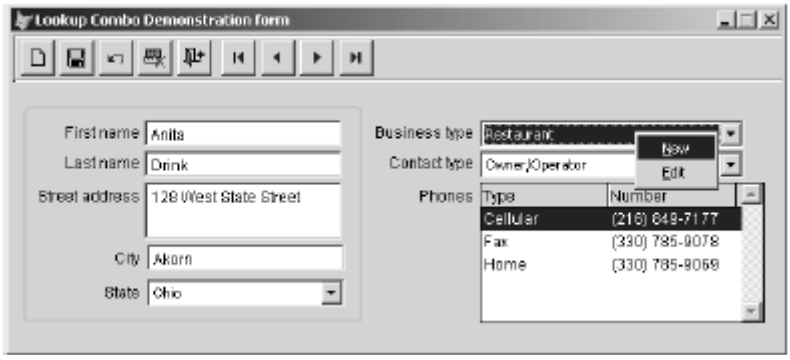

图 4：用户动作菜单

Andy：好看，但是弹出菜单如何获取数据？

Marcia：那是有点复杂。在 **ShowMenu()**方法里，我们创建了一个参数对象，它传递传递当前选择项目的关键值，和要求的动作（比如，“增加”或“编辑”），类目的名，和视图的名。它允许表单查看组合框所使用的数据，并显示相应的记录以编辑（或为空，如果动作为“增加”）。表单名为 **LookUpMaint**。

Andy：我明白表单在运行时使用了自定义的属性来保存参数对象所传递的值。有件事，我注意到偿已经定义了一个 **cDatabase** 属性，并且明确的设置了它的值，而不是在参数对象里来传递它。为什么？

Marcia：好的，我假设我们可以从组合框的视图里找出自己的数据库名。我不想麻烦。大多数情况下你只使用一个数据库，对吗？因此在设计时直接设定了它并不会产生问题。

Andy：在我的情形里是好的，但是如果有人想要处理多个数据库，那么它可以简单的加入必要的代码到 **ShowMenu()**方法里。最后，返回自己的数据库：

```
CURSORGETPROP("database")
```

Marcia：好的，如果你确实需要，你自己可以改它。

Andy：总之，由于它是个可更新的视图，在用户完成后表单所要做的只是 **TableUpdate()** 来提交数据，并返回给组合框，仅仅要 **PopulateCombo()**来更新变更的列表。这样做后你如何知道选择的是什么？

Marcia：弹出表单返回被增加或编辑的记录的主关键值。在 **PopulateCombo()**里我只是简单的利用 **AScan()**来在数组里查找与主关键值第五列相匹配的行。然后我设置组合框的 **ListIndex**，并且它会自动排序，不管绑定的列是字符还是整数。

Andy：这很酷！现在我对带有 **21** 查找的这个表单所要做的就是对这 **21** 个组合框实例设置它的 **ControlSoure** 和 **Category** 属性（并更改 **BoundColumn**，如果我需要绑定代码以代替主关键字）？

Marcia：是的。因为默认的视图名就是我们所使用的，你可以保留着。编辑屏幕是如图 **5** 插图，所有的源代码，都包含在下载文件里。

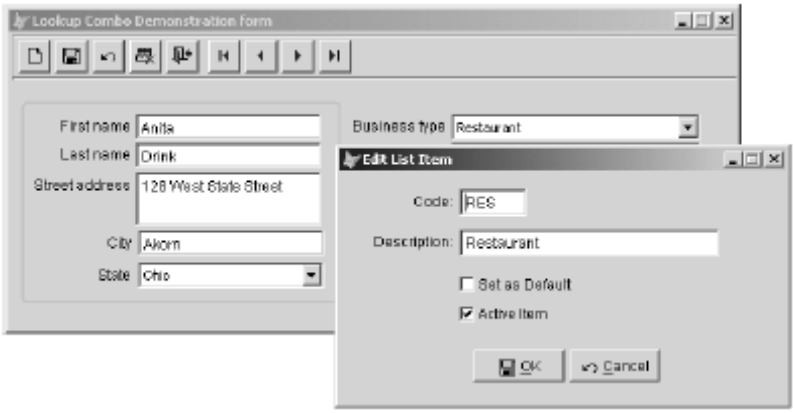

图 5：弹出编辑屏幕。

Andy：谢谢！这正可以用来处理我（和我们读者）先前的查找维护。

附件：**508KITBOX.ZIP**