2005 年第 11 期

继续上个月关于报告 **Bug** 的讨论，**Andy Kramek** 和 **Marcia Akins** 正在研究错误捕捉。关于这个主题已经有大量的文章了，而从 **8.0** 开始的 **TRY...CATCH** 极大的简化了程序员在这方面的工作。然而，在一个产品的环境下、在运行时的错误处理是一个完全不同的问题。

给 VFP9 报表增加富文本(RTF)格式

作者：Bo Durban

译者：CY

在 VFP 报表里包含 RTF 内容很长时间以来都是件难事。虽然利用通用字段和 OLEBound 控件以有限的办法可以实现它，但这个方法不能处理动态高度和跨页。通过利用 VFP9 定制的报表监听器和一个少见的对 Rich Edit 控件的 Windows 消息调用，Bo Durban 提出了在报表里包含 RTF 文本时所需要的东西。

能够在 **VFP** 报表里实现格式文本，已经在我的愿望表里很长时间了。我记得第一次我看到 **RTF** 控件是出现在 **VFRP** 表单里（它是在 **Solution** 示例的 **ActiveX** 部分里）。这是个非常令人激动的概念。你可以很容易的在 **VFP** 表单里创建和编辑 **RTF** 文本并存储在备注字段里。“我的客户会爱上它的”，我想。但是当我试图在报表里显示 **RTF** 时，所有这些兴奋都消失了。

我看到了其它实现 **RTF** 的可能性：我可以利用 **RTF** 控制的 **Print** 方法或是利用 **Word** 自动化服务。但是我不喜欢这些解决方案，因为我真的想在报表里合并格式文本，而并不只是利用它来打印。

我看到了微软的出版文章，“如何在 **VFP** 里应用 **RTF** 效果”。这个概念，非常灵巧，但相对于我要做的更效率低些。它在运行时动态的组装一个通用字段，并在报表的 **OLEBound** 控件里来显示 **RTF**。这比较慢，并且不支持动态高度和跨越多页。它在预览时也不能正确显示。我想要做的就是要与我所能显示备注字段里普通文本一样，能够在报表上显示格式文本。

然后，一直到 **VFP9** 所带来新的 **ReportListener** 类，我最终实现了我的愿望（有一点额外的工作，编写一个定制的监听器和一些其他代码）。参见图 **1**，就是可以容易在报表里包含显示格式文本的示例。

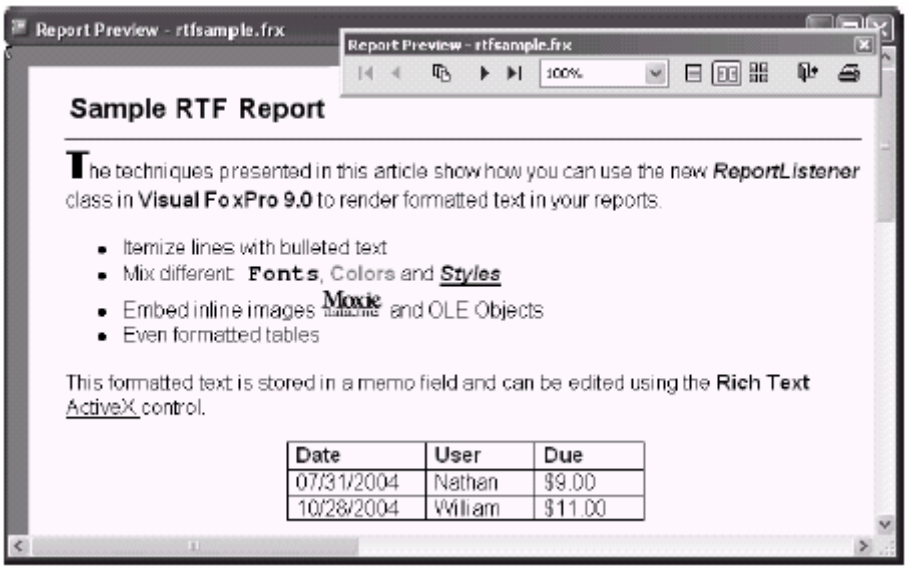

图 1：带格式文本支持的 VFP9 报表。

隐藏的消息

你可能知道 **VFP** 里所带的 **Rich Text ActiveX** 控件是 **RichEdit** 控件的封装，它是 **Windows Shell** 的一个控件。你不需要知道的是 **RichEdit** 控件通过 **SendMessage() API** 函数支持多种 **Windows** 消息。这包括 **EM_FORMATRANGE** 消息。通过发送 **EM_FORMATRANGE** 消息给 **RichEdit** 控件，你可以指示它在任何设备环境上重现 **RTF** 控件。这里是函数原型：

```
lResult = SendMessage( // returns LRESULT in lResult
    (HWND) hWndControl, // destination control handle
    (UINT) EM_FORMATRANGE, // message ID
    (WPARAM) wParam, // = (WPARAM) () wParam;
    (LPARAM) lParam // = (LPARAM) () lParam; );
```

hWndControl 参数是 **RichEdit** 控件的窗口句柄。这可以从 **Rich Text ActiveX** 控件的 **hWnd** 属性来获得。**EM_FORMATRANGE** 消息 **ID** 是定义为**(WM_USER+57)** 或 **0x400+57**。**wParam** 参数指示控件仅测量文本（**0**）或重现文本（非 **0**）。**lParam** 参数指向是 **FORMATRANGE** 结构的指针：

```
typedef struct _formatrange {
    HDC hdc;
    HDC hdcTarget;
    RECT rc;
    RECT rcPage;
    CHARRANGE chrg;
} FORMATRANGE;
```

通过利用结构，你可以调用 **RichEdit** 控件以在 **VFP9** 报表底层的 **HDC**（设备环境）

上重现，很容易从 ReportListener 的 GDIPlusGraphics 属性上派生出来。你可以指定你想要重现 RTF（从 ReportListener 事件发送给我们的信息）在页面上的矩形位置。在格式文本重现后，你可以查询矩形以获得 RTF 重现区域的尺寸。

你可以向 RichEdit 控件给出有效高度并且它可以返回给你它所需要的实际高度，这是有很有用的。你也可以利用 CHARRANGE 来指定重现的字符范围。这工作起来与 VFP 的 MLINE()函数的_MLINE 系统变量非常类似。如果 RTF 对于指定的矩形是太大的，而且你需要在下一页或下一列上继续，这将变得轻而易举。

好的，现在你知道了如何在报表上绘制 RTF。下一步你需要知道在哪里绘制。

这个概念是拉伸

我将利用一个标准的报表控件来作为报表的占位符，以指明我想要重现 RTF 的地方。然后我可以在我定制的报表监听器里替换那个占位符控件的 Render 事件，并让 RichEdit 控件在它的位置重现 RTF。然而，因 RTF 可以是任意高度的，我必须告诉 ReportListener 将要处理的格式文本有多高，并且这必须在 Render 事件触发前完成。

报表监听器的 AdjustObjectSize 事件似乎很完美。它为我给出了在这页上可以重现的最大有效高度，并且我可以告诉它实际上我需要的具体高度。我甚至可以强制这个元素连续到下一页上，通过设置它的高度为比最大有效高度更大的值。然后我可以挑选在下一页上的位置，并一直到 RTF 文本都被重现出来。

这个方法有两个问题：

首先，AdjustObjectSize 事件仅是在 Rectangle、Line 和 Image 控件上是有效的。我将利用 Rectangle 控件并以报表的运行时扩展来设置 RTF 表达式。我希望利用 Field 控件，因为在设计器里它已经有一个 Expression 属性。但是 Field 控件不能给我所需要的重现控制。

另一个问题是，Render 方法不会触发，如果对象高度被设置为比最大有效高度更大的值。Render 事件仅在对象可以在页里完全放入时才会被触发。为了克服这个问题，我们将在 AdjustObjectSize 事件里重现 RTF 文本，并且完全绕过 Render 事件（感谢 Lisa Slater Nicholls 的提示！）。

创建一个报表控件句柄类

为取代直接把重现代码放入到我的 ReportListener 子类，我决定创建一个定制控件句柄类（RTFHandler）。这个类的实例将在带 RTF 控件的报表运行的初期创建，并且控

件类句柄对象的 **HandleObjectSize** 和 **HandleRender** 方法将被调用，当每个控件的 **ReportListener** 事件发生时。

```
DEFINE CLASS RTFHandler AS Custom
    oForm = NULL
    oFR = NULL
    nTopMargin = 0
    nLeftMargin = 0

    PROCEDURE Init(oRL, oData, oNode)
        DECLARE Long GdipGetDC IN GDIPLUS ;
            Long graphics, Long @hdc
        DECLARE Long GdipReleaseDC IN GDIPLUS ;
            Long graphics, Long hdc
        DECLARE Long SendMessage IN WIN32API AS ;
            SendMessage_String ;
            Long hWnd, Integer Msg, ;
            Integer wParam, String @lParam

        ** The Top and Left positions in AdjustObjectSize
        ** do not include the offset for printer margins.
        ** Determine the printer margins for later
        This.SetPageMargins()
        IF PCOUNT() >= 3
            This.SetUp(oRL, oData, oNode)
        ENDIF
```

我利用了 **SetUp** 方法来初始化句柄，以与定制的 **RTF** 报表控件通信。我将需要一个 **Rich Text ActiveX** 控件的实例（包含一个非可见的表单对象），一个 **FORMATRANGE** 结构封装类的实例，和两个定制属性以跟踪当前文本重现的位置。

```
    PROCEDURE SetUp(oRL, oData, oNode)
        ** oRL – Reference to the ReportListener object
        ** oData – Reference to this controls FRX record
        ** oNode – Reference to this controls MetaData
        LOCAL cExpr
        This.oForm = CREATEOBJECT("Form")
        This.oForm.AddObject("oRTF","OLEControl", ;
            "RichText.RichTextCtrl")

        ** Create an instance of our FORMATRANGE
        ** wrapper class
        This.oFR = CREATEOBJECT("formatrange")

        ** We will need these custom properties to keep
        ** track of how much text has been rendered
```

```
    ** when we go to the next page
    ADDPROPERTY(oData, "nCharPos", 0)
    ADDPROPERTY(oData, "nCharLen", 0)
    cExpr = oNode.getAttribute("expr")
    IF NOT EMPTY(cExpr)
        oData.expr = cExpr
    ENDIF
```

当定制的 **RTF** 控件的 **AdjustObjectSize** 事件发生时，我将调用句柄的 **HandleObjectSize** 方法。这个方法将在报表上实际重现 **RTF** 文本，并且调整控件的高度以关联重现文本的高度。

HandleObjectSize 利用了重试属性来确定是否这是 **ReportListener** 第一次要重现这个带区。如果是，它将设置 **Rich Text** 控件的 **TextRTF** 属性为格式文本，并复位字符位置。

```
PROCEDURE HandleObjectSize(oRL, oData, oProp)
    ** oRL – Reference to our ReportListener object
    ** oData – Reference to our controls FRX record
    ** oProp – Reference to oObjProperties from
    ** AdjustObjectSize
    ** If this is our first time for this record,
    ** we need to initialize the RTF text
    ** and reset our position pointers
    IF NOT oProp.reattempt
        This.oForm.oRTF.TextRTF = EVALUATE(oData.expr)
        oData.nCharPos = 0
        oData.nCharLen = LEN(This.oForm.oRTF.Text)
    ENDIF
```

利用定制的封装类，**HandleObjectSize** 随后组装了一个 **FORMATRANGE** 结构，通过来自于监听器的 **GDIPlusGraphics** 句柄 **HDC**，并设置最大的重现矩形的尺寸。这个报表单位（**960dpi**）必须转换为 **TWIPS** 或 **1/1440** 英寸。注意：在 **SP1** 以前，发送到 **AdjustObjectSize** 方法的座标并未包含页边空。这个下载文件里的代码已经作了计算。

```
#DEFINE CHRG_ALL -1

hDC = 0
GdipGetDC(oRL.GDIPlusGraphics, @hDC)

** Set both HDC attributes to our Graphics object
   This.oFR.SetHDC(hDC,hDC)

** Set our render rectangle to the maximum
** available height
```

```
** Make sure we include our page margins
WITH oProp
    nLeft = .left*1440/960
    nTop = .top*1440/960
    nRight = (.left + .width)*1440/960
    nBottom = (.top + .maxheightavailable)*1440/960
ENDWITH
This.oFR.SetRC(nLeft, nTop, nRight, nBottom)

** Set our Page size
This.oFR.SetRCPage(0,0, ;
    oRL.GetPageWidth()*1440/960, ;
    oRL.GetPageHeight()*1440/960)

** Set our character range.
This.oFR.SetCHRG(oData.nCharPos, CHRG_ALL)
```

接着，**HandleObjectSize** 发送一个 **EM_FORMATRANGE** 消息给 **Rich Text** 控件的窗口句柄。这将在报表底层重现格式文本。**SendMessage** 函数将返回最后重现的字符位置。这个方法保存了这个值以便在下一个开始位置可以引用，如果内容需要连续到下一页。

```
#DEFINE WM_USER 0x0400
#DEFINE EM_FORMATRANGE (WM_USER+57)
cFR = This.oFR.data
oData.nCharPos = SendMessage_String( ;
    This.oForm.oRTF.HWnd, ;
    EM_FORMATRANGE, ;
    -1, ;
    @cFR)
```

SendMessage 函数也将更新 **FORMATRANGE** 结构的“**rc**”属性为重现的实际尺寸。句柄对象可以利用这个信息来调整矩形的尺寸。如果所有的文本都被重现，矩形的高度被调整为重现的高度。如果只是部分文本被重现，矩形的高度就被调整为 **MaxHeightAvailable** 加 **1**，以强制带区连续到下一页。

```
This.oFR.data = cFR

** We want to grab the updated rectangle bottom.
** This is the physical position the RichText
** control stopped drawing.
nBottom = 0
This.oFR.GetRC(0,0,0,@nBottom)

** Did we render all of the RTF text?
IF oData.nCharPos < oData.nCharLen
    ** If not, force a new page by setting the object
```

```
        ** height to a value higher than the max available
        oProp.height = oProp.maxheightavailable+1
    ELSE
        ** If so, update the height to the position the
        ** RichText control finished drawing.
        oProp.height = (nBottom*960/1440) - ;
        (oProp.top+This.nTopMargin)

        ** The control requires that we clear its cache
        ** when finished rendering. Pass a NULL to lParam.
        SendMessage_String( ;
            This.oForm.oRTF.HWnd, ;
            EM_FORMATRANGE, ;
            -1, ;
            NULL)
    ENDIF

    ** Tell the report engine to read our updated values
    oProp.reload = .T.

    ** We must release the device context we created
    GdipReleaseDC(oRL.GDIPlusGraphics, hDC)
```

因为句柄对象已经重现了 **RTF** 文本，当从监听器的 **AdjustObjectSize** 事件里调用时，就不需要在报表控件的 **Render** 事件里做任何事。我在 **AdjustObjectSize** 事件里作重现的原因是，因为 **Render** 事件只会在矩形控件的每个带区触发一次，并且只在 **oObjProperties.Height** 属性小于**.MaxHeightAvailable** 时。句柄的 **HandleRender** 方法只是简单返回**.T.**，发信号给监听器以便给控件的 **Render** 事件发出一个 **NODEFAULT** 指令。

```
    PROCEDURE HandleRender(oRL, oData, nLeft, nTop, ;
        nWidth, nHeight, nObjectContinuationType, ;
        cContentsToBeRendered, GDIPlusImage)
        RETURN .T.
```

创建一个定制的ReportListener类

现在让我们来定义一个定制 **ReportListener** 类，它将替代矩形元素的缺省重现行为。

在 **BeforeReport** 事件里我装载整个 **FRX** 到一个集合对象里，以便在报表运行时我可以快速访问报表元素的 **FRX** 记录。我也可以加入定制属性到每个数据对象，以便让我可以在事件间保留信息。

注意到我在调用 **LoadObject** 方法前切换 **DataSession** 回到 **CurrentDataSession**。

如果我不这样做，在 **LoadObject** 方法里创建的任何对象将会在报表的 **DataSession** 范围里。

```
DEFINE CLASS CtrlListener AS ReportListener
    oFRX = NULL
    ListenerType = 1\

    PROCEDURE BeforeReport()
        LOCAL oData AS Object

        ** Load the entire FRX into a collection object
        ** so we can easily access the records later.
        This.oFRX = CREATEOBJECT("Collection")
        SET DATASESSION TO (This.FRXDataSession)
        SELECT frx
        SCAN
            SCATTER MEMO NAME oData
            ** Set the DataSession back to the current. If
            ** not, any objects created in LoadObject( )
            ** will be scoped to the FRX DataSession
            SET DATASESSION TO (This.CurrentDataSession)
            This.LoadObject(oData)
            This.oFRX.Add(oData)
            SET DATASESSION TO (This.FRXDataSession)
        ENDSCAN
        SET DATASESSION TO (This.CurrentDataSession)
        RETURN
    ENDPROC
```

LoadObject 方法用于读取存储在报表元素里的 **XML MemberData Extensions**。它可以让我们知道是否报表元素请求一个特别的句柄，因此代码可以加入它的一个实例到数据对象里。

为了这篇文章的目的，我只是想做一个非常简单的查找 **MemberData XML**，以确定每个元素的句柄。我主要想要展示利用 **MemberData** 和句柄对象可以实现定制控件的概念。对于 **VFP9**，利用 **MemberData** 以动态扩展 **ReportListener** 是首选的方法。更多信息，参见 **VFP9** 帮助文件的“**Report XML MemberData Extensions**”。

```
PROCEDURE LoadObject(oData)
    LOCAL oXML AS MSXML2.DomDocument
    LOCAL oNode, cClass

    ** Add a custom handler property
    ** to hold our handler object
    ADDPROPERTY(oData,"oHandler",NULL)
```

```
    ** Read our custom metadata to see if we need
    ** to create a custom handler for this object
    IF NOT EMPTY(oData.style)
        oXML = CREATEOBJECT("MSXML.DomDocument")
        IF oXML.loadXML(oData.style)
            oNode = oXML.selectSingleNode("*/rptctrl")
            IF NOT ISNULL(oNode)
                ** <rptctrl class="RTFHandler"
                ** expr="rtfsample.rtf"/>
                cClass = LOWER(oNode.getAttribute("class"))
                oData.oHandler = CREATEOBJECT(cClass, ;
                    This, oData)
            ENDIF
        ENDIF
    ENDIF
    oXML = NULL
ENDPROC
```

加入代码到 **AdjustObjectSize** 事件里，以调用 **HandleObjectSize** 方法，如果报表元素有关联的句柄。

```
PROCEDURE AdjustObjectSize(nFRXRecno, oObjProperties)
    LOCAL oData AS Object
    oData = This.oFRX(nFRXRecno)

    ** See if there is a custom handler for this
    ** object and let it handle the size adjustment
    IF VARTYPE(oData.oHandler) = "O"
        oData.oHandler.HandleObjectSize(This, oData, ;
            oObjProperties)
    ENDIF
ENDPROC
```

也可以加入代码到监听器的 **Render** 事件里以调用句柄对象的 **HandleRender** 方法，如果这个报表元素有关联的句柄。如果 **HandleRender** 方法返回为真，它可以假定为这个报表元素的重现已经被处理，并且已经发出 **NODEFAULT** 指令。

```
PROCEDURE Render(nFRXRecno, nLeft, nTop, nWidth, ;
    nHeight, nObjectContinuationType, ;
    cContentsToBeRendered, GDIPlusImage)
    LOCAL oData AS Object
    oData = This.oFRX(nFRXRecno)

    ** Check to see if there is a custom handler for
    ** this object and let it handle the Render.
```

```
        ** If it returns .T. bypass the default behavior
        IF VARTYPE(oData.oHandler) = "O"
            IF oData.oHandler.HandleRender(This, oData, ;
                nLeft, nTop, nWidth, nHeight, ;
                nObjectContinuationType, ;
                cContentsToBeRendered, GDIPlusImage)
                NODEFAULT
            ENDIF
        ENDIF
    ENDPROC
```

最后，加入代码到监听器的 **AfterReport** 事件里，以便清理在 **AfterReport** 事件里创建的报表数据对象集合。如果集合没有被手动清理干净，对象指针将会继续打开着。这可能会导致 **FRX** 的 **DataSession** 将永远不会关闭，并且也可能会让 **FRX** 表继续打开着。

```
    PROCEDURE AfterReport()
        ** Clear out all of the objects or the FRX
        ** DataSession will not go out of scope
        This.oFRX.Remove(-1)
        This.oFRX = NULL
    ENDPROC
```

加入 RTF 控件到报表里

加入一个矩形控件到报表里你想要 **RTF** 控件出现的地方。记住高度并不重要。**ReportListener** 将会拉伸矩形以适应文本。

接着，设置矩形的 **MemberData XML**，以便 **ReportListener** 监听器知道这个控件的重现将会被 **RTF** 句柄所处理。想象中，它将被一个定制报表设计器屏幕所处理。但是对于这篇文章，你将不得不勇敢面对它，并且手工编辑 **XML**。为此，打开矩形属性窗口，并选择“**Other**”标签。在“**Run-time extensions**”部分点击“**Edit Settings...**”按钮。然后，在“**Run-time extensions**”屏幕点击“**Edit XML...**”按钮（见图 **2**）。

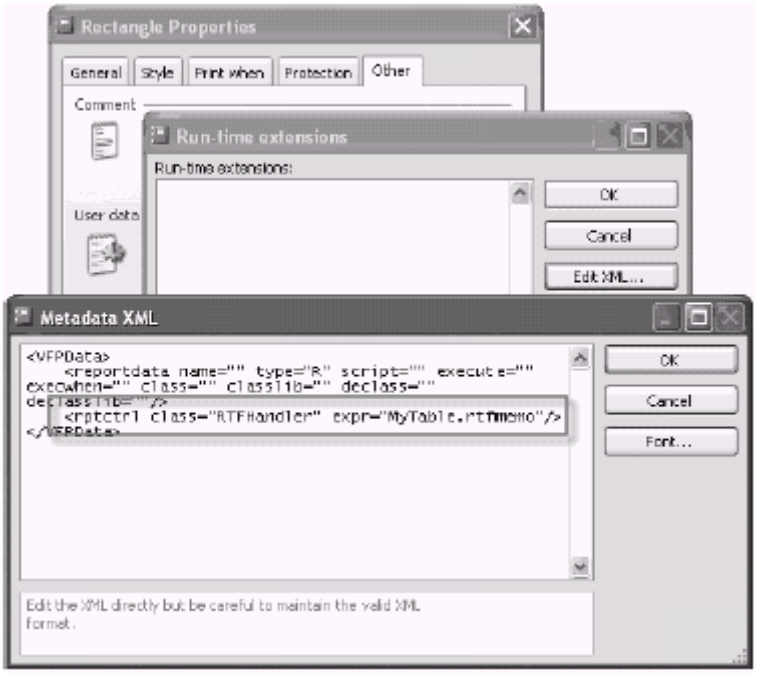

图 2：在 Report Designer 的“Run-time extensions”屏幕指定 RTFHandler 类。

你将可以见到已经有些 XML 代码在编辑框里。你需要加入一个“rptctrl”节点，以告诉定制的 ReportListener 这个控件有定制句柄。在</VFPData>关闭标志前加入下列的文本，那里“MyTable.rtfmemo”是一个计算 RTF 文本的有效表达式。

```
<rptctrl class="RTFHandler" expr="MyTable.rtfmemo">
```

在所有打开的对话框里选择 OK 并保存报表。

为了运行报表，只需发出 REPORT FORM 命令，并在 OBJECT 子句里指定该定制的 ReportListener 对象。

```
REPORT FORM "MyRTFReport" PREVIEW ;
OBJECT CREATEOBJECT("CtrlListener")
```

总结

现在我可以在VFP报表里重现格式文本，以我所想要的方法。这个代码只是稍稍有点复杂，但是实现起来非常简单。Rich Text控件甚至可以支持RTF规范的巨型子集，它包含黑点、图像、表、OLE嵌入对象等等。我在这里所展示的这个示例使用了一个备注字段来存储RTF文本，但是你也可以从文件里装载RTF，甚至在运行时创建RTF。对于RTF代码，参见“Rich Text Format (RTF) Specification, version 1.6”，在http://msdn.microsoft.com/library的MSDN库。

运行时的智能感知

作者：Doug Hennig

译者：CY

VFP9 提供了对运行时智能感知的支持。本月，Doug Hennig 探查了它的用途，讨论如何实现它，并且扩展了智能感知收藏夹以支持它。

正如我在**FoxTalk 2.0 2005.08**的文章，“智能感知的收藏夹”里所说，智能感知是最早加入**VFP**的最好特性。它对**VFP**开发者提供了更高效率的促进。然而，一直到**VFP9.0**之前，它都仅限于开发环境。从**VFP9.0**开始，智能感知就支持在运行环境下工作。在我们查看如何工作前，让我们先来讨论其原因。

近来，我已经把钩子放入我的应用程序以允许其被定制。比如，**Stonefield** 查询，用户可以在许多地方指定代码：当程序启动或关闭时，在数据字典被装载后（因此你可以动态切换它，如果你需要），在用户登录前，在要查询的数据被检索后但在报表被运行前，在报表运行后，等等。这样做的理由是灵活，我不可能提出每个用户所要求的每个配置变化，因此我让它自行完成。明显的，这需要 **VFP** 语言的知识，但是它并不繁琐，对于开发者、**IT** 维护人员、高级用户，仅仅是简单的改变。

在 **VFP8** 版本里，用户可以输入必要的代码到在相应的代码编辑对话里提供的编辑框，但是并没有智能感知的帮助。在 **VFP9** 版本里，并不是仅有 **VFP** 的命令和函数才有智能感知，在 **Stonefield** 查询对象模式下也有智能感知。图 **1** 展示 **VFP9** 版本的一个示例，它看起来象在输入代码。注意到在这个示例里语法着色和成员列表智能感知。

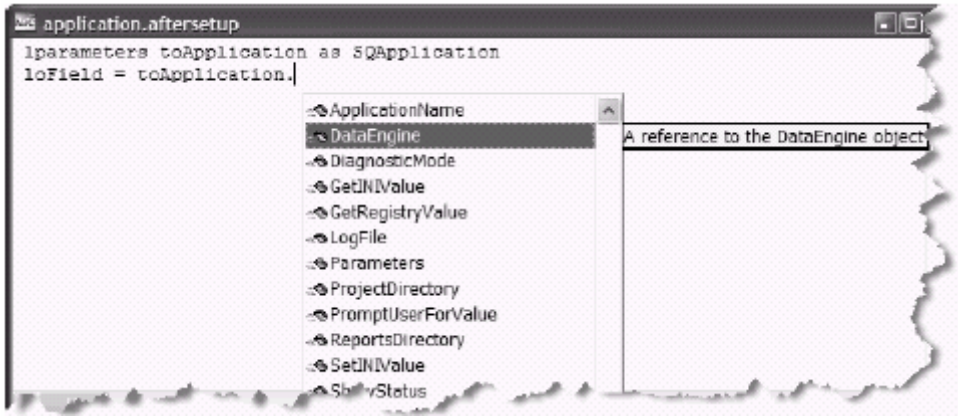

图 1：运行时的智能感知是很有用的，如果你让用户来编导你的程序。

在 **2004** 年 **DevEssentials** 会议讲话里，**Toni Feltman** 介绍了其它几种，不用太多

技巧的，用于运行时的智能感知，比如提供某些类似于微软 Office 的自动校正特性，和业务信息的智能感知，比如医学诊断。

智能感知如何工作

智能感知是通过由_CODESENSE 系统变量所指定的程序和利用_FOXCODE 系统变量所指定的表来运行的。默认的，在开发环境里，_CODESENSE 指向在 VFP 程序目录里的 FOXCODE.APP，而_FOXCODE 指向 HOME(7)所指定目录的 FOXCODE.DBF。这些系统变量在运行时环境默认为空。在运行时提供智能感知所需要做的一件事就是在你的代码里设置这些变量为相应的值，并为你的用户提供智能感知的应用程序和表。

虽然我可以随着我的应用程序一起发布 FOXCODE.APP，我实际上想要一个定制的智能感知应用程序以支持智能感知收藏夹（FFI），正如我在我的八月份文章里所讨论的，因此我想要如何创建它。在查看了 VFP 程序目录 Tools\XSource\VFPSource\FoxCode 目录里 FOXCODE.APP 程序的源代码后（从 Tools\Xsource 的 XSOURCE.ZIP 里解压缩出来），我注意到 FOXCODE.PRG 是 FOXCODE.APP 的主程序。我的最先想法是简单的把 FOXCODE.PRG 包含到我的应用程序里并设置_CODESENSE 为“FOXCODE.PRG”。不幸的是，智能感知并没有如我所做的运行起来，或许是因为 VFP 无法在运行的 EXE 文件里找到指定的程序。因此，我决定创建一个定制的 FOXCODE.EXE 来代替。

为了创建这个 EXE，我开始从 Tools\XSource\VFPSource\FoxCode 里复制 FOXCODE .PRG 和 FOXCODE.H（因为我需要对 PRG 作些改变，我复制它并对副本作修改，而不是仅仅把程序加入到我的项目里）。接着，我注释了在程序代码里的 DO FORM 调用和 FOXCODE.PRG 里的 GetInterface 方法，否则这些调用将会导致项目管理器加入许多我们在运行时所不需要的文件到项目里。

然后我创建了一个名为 FOXCODE.PJX 的项目，并加入定制的 FOXCODE.PRG、FFI.VCX（FFI 类库，详见我的八月份文章），包含我的应用程序对象 FFI 记录的 FFI.DBF 表，一个仅简单包含 RESOURCE=OFF 的 CONFIG.FPW 文件（避免用户意外运行 FOXCODE.EXE 时创建 FOXUSER.DBF 文件），和 Tools\XSource\VFPSource\FoxCode 目录下的 FOXCODE2.DBF 文件（这个是 FOXCODE.PRG 用到的）。最后我从这个项目建立了 FOXCODE.EXE。

我所尝试的另外两件事也没有成功：

我在 FOXCODE.PJX 里包含了 FFI 用于成员列表的图像文件，PROPTY.BMP 和 METHOD.BMP，但是成员列表里没有图像。我发现这些图像文件要么不得不包含在我的主应用程序项目里或者作为独立文件提供。

我试图在 **FOXCODE.PJX** 里并/或者 **SAMPLE.PJX** 里包含智能感知表、**FOXCODE.DBF**，智能感知给出了一个错误，智能感知表没有找到。因此，你必须把 **FOXCODE.DBF** 和 **FPT** 作为独立文件发布。你可以更名它们为其它的，只需简单的改变 **_FOXCODE** 系统变量以指定新的名。同样，你不需要提供你在开发环境里所使用的智能感知表的副本；你可以复制它并编辑你觉得适合的内容，比如移除某些在运行时环境下不会被感知到的命令或函数的智能感知记录，或者在备注字段里加入业务相关的记录。

现在我有一个定制的智能感知应用程序和表，所有我所要做的就是告诉我的应用程序，让它把 **_CODESENSE** 和 **_FOXCODE** 设置为相应的值。这里的代码来自于 **STARTUP.PRG**，**SAMPLE.PJX** 的主程序。注意到它保存和恢复了 **_CODESENSE** 和 **_FOXCODE**，如果我是运行在开发环境，因此在测试时我不会弄乱平常的智能感知设置。

```
* Set up IntelliSense.
local lcCodeSense, ;
    lcFoxCode

if version(2) = 2
    lcCodeSense = _codesense
    lcFoxCode = _foxcode
endif version(2) = 2

_codesense = 'FoxCode.EXE'
_foxcode = 'FoxCode.DBF'

* Do the normal application stuff here (this is a
* demo, so we'll just run the Script Editor form).
_screen.Caption = 'Sample Application'
do form ScriptEditor

* If we're in development mode, restore the
* IntelliSense settings before exiting.
if version(2) = 2
    _codesense = lcCodeSense
    _foxcode = lcFoxCode
endif version(2) = 2
```

更新智能感知收藏夹

我不得不对 **FFI.VCX** 作些调整以处理运行时的智能感知：

我把 **FFIBuilderForm** 从 **FFI.VCX** 移动到一个新的名为 **FFIBUILDER.VCX** 的类库，因为我们不需要在运行时 **EXE** 的可视类开销。

我修改了 **FFIFoxCode.DisplayMembers** 以在运行时环境下不使用图像文件的路径，因为它们被编译进了 **EXE**。

这是个为运行时智能感知的简化修改。不同于为每个命名空间在智能感知表的不同脚本记录请求，我创建了单个通用的 **HandleFFI** 脚本记录并使得所有的命名空间记录（其 **TYPE = "T"**，并且 **ABBREV** 设置我们所需要的智能感知对象的命名空间）利用 **HandleFFI** 作为它的脚本记录。现在这个 **HandleFFI** 代码可以在 **FOXCODE.EXE** 的 **FFI.VCX** 里找到，并且不再需要传递硬编码的类名或类库给 **FFIFoxCode.Main**。这里是更新脚本代码：

```
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 = 'FFI.vcx'
    cProxyEXE = 'Path\FoxCode.EXE'

    procedure Main
        local loFoxCode, ;
            luReturn
        loFoxCode = newobject(This.cProxyClass, ;
            This.cProxyClasslib, This.cProxyEXE)
        if vartype(loFoxCode) = 'O'
            luReturn = loFoxCode.Main(This.oFoxCode)
        else
            luReturn = ''
        endif vartype(loFoxCode) = 'O'
        return luReturn
    endproc
```

```
enddefine
```

正如前面的修改结果，现在 **FFIFoxCode.Main** 只接受 **toFoxCode** 作为参数；先前的版本也接受命名空间、类和所要的类库。因为我们可以从 **toFoxCode** 对象的 **Data** 成员得到命名空间，并且从 **FFI.DBF** 的相应记录里得到类和类库，就不需要传那些参数。这里是这个方法的代码：

```
lparameters toFoxCode
local lcNameSpace, ;
    loData, ;
    lcReturn, ;
    lcTrigger

with toFoxCode
    * Get the namespace and an object from the FFI table
    * for that namespace.
    lcNameSpace = .Data
    loData = This.GetFFIMember(.UserTyped, ;
        lcNameSpace)
    lcReturn = ''

    * If we're on the LOCAL statement, handle that by
    * returning text we want inserted.
    if atc(lcNameSpace, .MenuItem) > 0
        lcReturn = This.HandleLOCAL(toFoxCode, ;
            lcNameSpace, trim(loData.Class), ;
            trim(loData.Library))
    else
        * Rest of the code unchanged
```

我对 **FFIBuilderForm** 所做的其他两个修改要感谢读者 **Michael Hawksworth**。首先，我调整了 **LoadTree** 方法以加入处理创建者表单上运行对象的任何成员数据，它也挑选了任何全局成员数据。第二个修改是在 **GetObjectReference**，它现在可以处理基于表单的对象。

在运行时使用智能感知

现在我们已经把它建立起来了，那么我们在运行时如何使用它？就如设计时，有两个地方的智能感知在运行时是支持的：在代码窗口（**PRG**）和备注字段。如果在编辑框里它可以支持将会是很完美的，因为我们要处理的是控件而不对象，但是并没有这么幸运。注意到让智能感知能够对备注字段工作是很有机智的：你需要确保折行是关闭的并且语法着色是打开的，这意味着提供一个定制的 **FOXUSER** 资源文件来代替。正因为如此，我的优

先选择是使用一个 **PRG** 文件。即使用户在备注字段里最后输入结束，你仍然可以复制备注字段内容来文件，利用 **PRG** 窗口来修改那个文件，然后再把文件内容写回到备注字段。这就是示例程序所做的。

这个在 **ScriptEditor.SCX** 里的 **Edit Code** 按钮的 **Click** 方法，示例程序的主表单，有下列代码：

```
local lcPath, ;
    loForm

* Write the current contents of CODE to a file.
lcPath = addbs(sys(2023)) + trim(NAME)
strtofile(CODE, lcPath)

* Create a form with the desired characteristics for
* the PRG window.
loForm = createobject('Form')
with loForm
    .Caption = trim(NAME)
    .Width = _screen.Width - 50
    .Height = _screen.Height - 50
    .FontName = 'Courier New'
    .FontSize = 10
Endwith

* Edit the code in a PRG window, then put the results
* back into CODE.
modify command (lcPath) window (loForm.Name)
replace CODE with filetostr(lcPath)
erase (lcPath)
```

这个代码把记录的当前内容到写到用户临时文件目录下的一个 **PRG** 文件里，创建一个带有我们想要的特性的编辑窗口表单，编辑这个文件，然后并且把 **PRG** 文件内容写回表并删除 **PRG** 文件。图 **2** 展示了智能感知是如何工作的，当你运行示例程序，选择一个脚本，并且点击按钮。

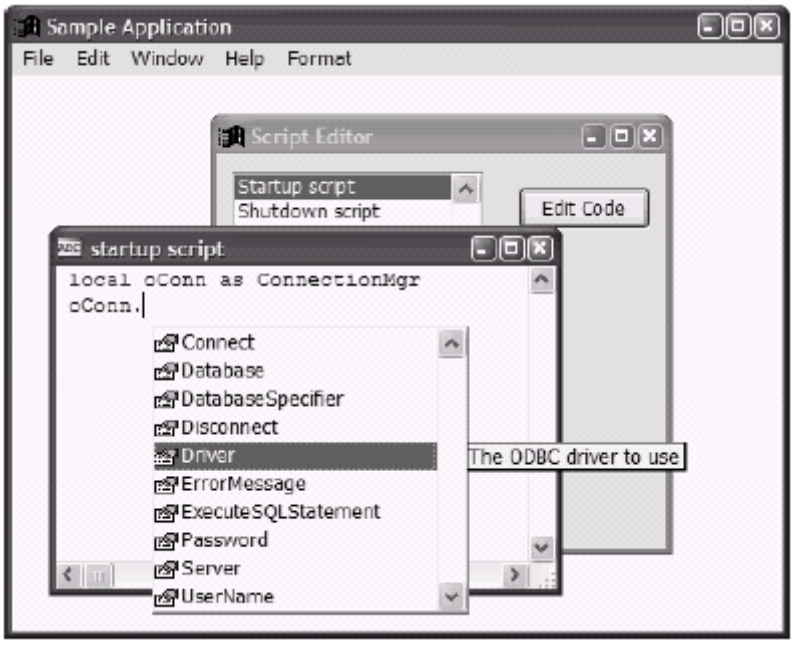

图 2：智能感知在运行时正如我们在设计时一样运行。

注意到其他特殊东西，如字体、窗口尺寸和布局，你在 **PRG** 所出现的窗口上没有太多的控制。不受欢迎的动作是在代码窗口外点击，将会导致它被自动关闭（除非你使用了 **NOWAIT**，它将引入其他复杂问题）。同样，如果你的程序是在顶层表单，也有其它问题。如果你不在 **MODIFY COMMAND** 语句里使用 **IN WINDOW** 子句（并且指定你的顶层表单名），代码窗口将不会出现。使用 **IN WINDOW** 将使得代码窗口作为顶层表单的子窗口，因此它不能移出或大过表单。因此我们将不得不面对控制的不足，我很担心。

在测试时我也陷入智能感知的两个缺陷：

当你输入 **LOCAL SomeVariable AS** 时，正常情况下出现的类型列表并没有在运行时出现。

如果你有一个对象是另一个对象的成员（比如在这个月示例里 **cApplication** 对象的 **User** 成员，它包含一个指向 **cUser** 对象的指针），为此成员出现的智能感知却是父类对象的。比如，如果你输入 **LOCAL loUser as Application.User**（**cApplication** 为是以 **Application** 命名空间来注册的）。当你输入“**loUser**”和一个点号，智能感知列表显示的是 **cApplication** 类，而不是 **cUser** 类。

总结

智能感知是那些我们 **VFP** 开发者一旦开始使用后将不可失去的东西。现在你可以在你的应用程序里给你的用户同样的益处。明显地，这个技术并不是适合于每一个人，但是如果它适合于你的应用程序，我保证你的用户将会因此爱上你。

在 VFP 9 表单上的 GDI+：更多文本效果

作者：Craig Boyd

译者：fbilo

这是由 Craig Boyd 撰写的关于 GDI+ 及其在 VFP 中应用系列文章的第四篇，同时也是上个月话题（在 VFP 表单上的 GDI+ 文本效果）的延续。本月，Craig 使用文本轮廓（图形路径）和大量的笔刷，并还将绘制 Unicode 文本。

在我上个月的文章中，我演示了如何使用 **GDI+** 来把文本放到 **Visual FoxPro** 表单上，其中包括如何实现几个有趣的文本效果。这个月，我将通过对“一些使用轮廓和画笔来建立令人震惊的文本效果示例”的解释，来结束这个讨论。

注意：由于这篇文章是对上个月话题的继续，因此我们将从下载文件中的示例 **2** 来开始。在本月下载文件中的项目包含两篇文章的全部示例代码。

更多的文本效果：示例 2

在这个示例中呈现的文本效果演示的是如何使用一个图形通道（**drawing path**）。一个图形通道是一系列的绘图要素（线段、矩形、弧形等等）。为了便于理解，你可以把一个图形通道看作是一个轮廓。图形通道是在 **GDI+** 中的一个强大工具，就像图 **1** 中演示的文本效果。

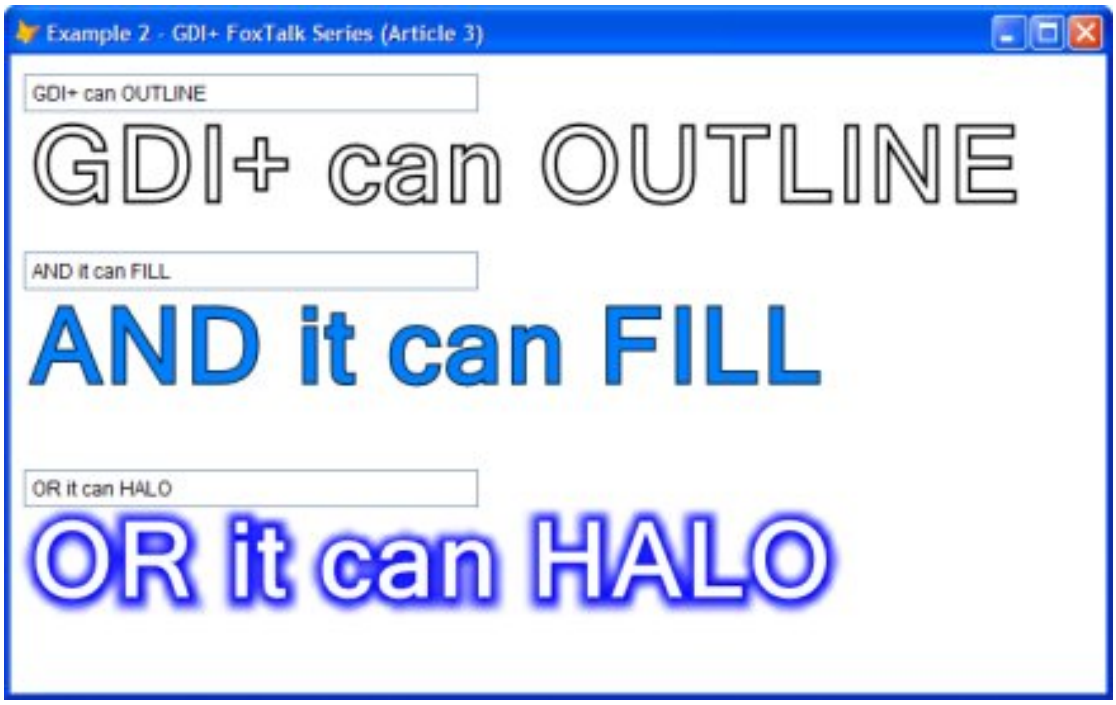

图 1、图形通道可以被用来生成轮廓、并填充它们。当与别的 GDI+ 功能结合使用时，可以生成象光晕这样的高级文本效果。

文本轮廓技术

文本轮廓是通过“使用一个图形通道、被它添加一个字符串、然后让 GDI+ 去绘制这个通道”来实现的。让我们来看看用于这个效果的代码：

```
*!* 要描绘出轮廓的字符串
lcString = ALLTRIM(thisform.text1.Value)

*!* 获得将被用于绘制这个字符串的字体系列
lnFontFamily = 0
GdipGetGenericFontFamilySansSerif(@lnFontFamily)

*!* 如果你想要用一个特别的字体，
*!* 可以使用象下面这样的代码
*!* lnFontCollection = 0
*!* lcFontName = STRCONV("Verdana",5)
*!* GdipCreateFontFamilyFromName(lcFontName, ;
*!* lnFontCollection, @lnFontFamily)

*!* 建立一个将被轮廓的通道
lnPath = 0
GdipCreatePath(0, @lnPath)
```

```
*!* 建立一个矩形来告诉 GDI+
*!* 我们想把字符串添加到通道的何处
logpRect = CREATEOBJECT("gpRectangle", 1, 30, ;
    this.Parent.width, this.Parent.height)

*!* 将字符串添加到通道上去
GdipAddPathStringI(lnPath, STRCONV(lcString,5), ;
    LEN(lcString), lnFontFamily, FontStyleBold, 64, ;
    logpRect.GDIRect, 0)

*!* 建立一个画笔，该画笔将被用于绘制该通道（轮廓）;
*!* 然后指定轮廓的颜色和宽度
lnRGB = RGB(255,0,0)
lnAlpha = 255 && 不透明 - 若为 0，则将为透明
logpColor = CREATEOBJECT("gpcolor", MOD(lnRGB, 256),;
    MOD(BITRSHIFT(lnRGB, 8), 256), ;
    MOD(BITRSHIFT(lnRGB, 16), 256), ;
    lnAlpha)
logpPen = CREATEOBJECT("gppen", logpColor.argb, ;
    2, UnitPixel)

*!* 把通道（轮廓）画到脱屏位图上，该位图将会被绘制到表单上
GdipDrawPath(this.gpgraphicsdb.gethandle(), ;
    logpPen.GetHandle(), lnPath)
```

在决定了将要绘制轮廓的字符串以后，我通过使用 **GDI+** 平台 **API** 函数 **GdipGetGenericFontFamily()** 来取得了一个对 **Sans Serif** 字体系列的句柄。这么做就决定了在字符串被添加到图形通道上去的时候，将被用于该字符串的字体的类型（如果希望使用一个指定的字体而不是一个字体系列，你可以使用我在注释中提供了的代码）。

为什么要使用一个字体系列而不是一个专门的字体？有时候，某个用户的系统上也许并没有一个特定的字体，一个字体系列是许多类似字体的一个集合，**GDI+** 将试着换用来自同一字体系列中的一个类似字体。这就能最大限度的保证了即使用户没有你在设计时用于绘制内容的字体，你绘制出来的文本仍然能保持你所期望的外观和效果。

下一步，我使用 **GDI+** 平台 **API** 函数 **GdipCreatePath()**。**lnPath** 是一个输出参数，当该函数执行成功时，**lnPath** 中将包含着一个有效的图形通道句柄，而我使用该句柄作为发送给 **GDI+** 平台 **API** 函数 **GdipAddPathStringI()** 的第一个参数。这个函数将会把“我已经为之定义了一个特定字体系列、样式和大小”的字符串添加给图形通道对象。当这个函数成功的执行完毕之后，**lnPath** 就代表这个文本字符串的轮廓。

最后，我定义了一个 **gpPen** 对象，该对象定义了这个轮廓的颜色（在本例中是蓝色）

和宽度（2 pixels），然后让 GDI+ 使用 GDI+ 平台 API 函数 GdipDrawPath() 去绘制这个通道。

文本轮廓的填充技术

继续谈文本轮廓，我可以用一个对 GDI+ 平台 API 函数 GdipFillPath() 的调用来对轮廓通道进行填充：

```
*!* 在脱屏位图上填充通道（轮廓）
GdipFillPath(this.gpgraphicsdb.gethandle(),;
    logpSolidBrush.GetHandle(), lnPath)
```

在这段代码中要注意的另一件事情，是 GDI+ 平台 API 函数 GdipResetPath() 的使用。由于我是在再次使用前面文本轮廓示例中的同一个通道，因此，在把这个字符串添加到这个通道之前，我必须先重置（Reset）这个通道，因为如果不重置就直接把字符串添加到该通道上的话，得到的结果将是同时包含着前后两个字符串的一个通道。

光晕技术

在这里示例中最复杂的文本效果当然是在图一底部显示的光晕了。这个效果是通过使用一个位图、一个单独的图形对象、一个矩阵来建立的，并且它还利用了一个抗锯齿的边缘效果。

```
LOCAL logpBitmap, logpHaloGraphic, ;
    logpHaloPen, logpBlackBrush, ;
    logpRectDest, logpRectSrc, lnMatrix

*!* 要做出光晕效果的字符串
lcString = ALLTRIM(thisform.text3.Value)

*!* 建立一个位图，我们将在其上画光晕效果
*!* 我们将它设定为表单的 1/5 大小
logpBitmap = CREATEOBJECT("gpBitmap",thisform.width/5,;
    thisform.height/5)

*!* 因为我们要再次使用前面的通道，所以对通道进行重置
GdipResetPath(lnPath)

*!* 建立一个长方形对象，
*!* 以告诉 GDI+ 我们将把这个光晕效果画在位图上的什么地方
logpRect.create(1, 270, this.Parent.width, this.Parent.height)
```

```
*!* 将字符串添加到通道上
GdipAddPathStringI(lnPath, ;
    STRCONV(lcString,5), LEN(lcString),;
    lnFontFamily, FontStyleBold, 64,;
    logpRect.GDIRect, 0)

*!* 建立一个图形对象，以便我们用于在其上画我们建立的缩略图
logpHaloGraphic = CREATEOBJECT("gpGraphics")
logpHaloGraphic.CreateFromImage(logpBitmap)

*!* 保证在绘制图像和文本用上了抗锯齿功能，
*!* 当位图被放大到全尺寸时，
*!* 正是抗锯齿生成了光晕效果
GdipSetSmoothingMode(logpHaloGraphic.gethandle(),SmoothingModeAntiAlias)
GdipSetTextRenderingHint(logpHaloGraphic.gethandle(),;
    TextRenderingHintAntiAliasGridFit)
GdipSetInterpolationMode(logpHaloGraphic.gethandle(),;
    InterpolationModeHighQualityBicubic)

*!* 建立一个矩阵，并改变图形对象的大小
*!* 以便 64 点的文本能以缩略图的形式被绘制到位图上
*!* (.2, 0, 0, .2, 0, 0) = 1/5 size
lnMatrix = 0
GdipCreateMatrix2(.2, 0, 0, .2, 0, 0, @lnMatrix) && 使它缩小
GdipSetWorldTransform(logpHaloGraphic.gethandle(),lnMatrix)

*!* 我们想让光晕的颜色为 Alien green
*!* 因此我们为轮廓建立一个画笔对象
lnRGB = RGB(0,255,0)
lnAlpha = 225
logpHaloColor = CREATEOBJECT("gpcolor", MOD(lnRGB, 256),;
    MOD(BITRSHIFT(lnRGB, 8), 256), ;
    MOD(BITRSHIFT(lnRGB, 16), 256), ;
    lnAlpha)
loHaloPen = CREATEOBJECT("gppen", logpHaloColor.argb,2, UnitPixel)

*!* 轮廓将以同样的绿色来填充，
*!* 因此我们建立一个画笔来填充轮廓
logpHaloBrush = CREATEOBJECT("gpsolidbrush", logpHaloColor.argb)

*!* 画出绿色的光晕，并填充它
GdipDrawPath(logpHaloGraphic.gethandle(), loHaloPen.GetHandle(), lnPath)
GdipFillPath(logpHaloGraphic.gethandle(), logpHaloBrush.GetHandle(), lnPath)
```

```
GdipDeleteMatrix(lnMatrix) && release matrix

*!* 使用两个长方形和 DrawImagePortionScaled 方法，
*!* 我们把 1/5 大小的位图再还原为它原始的大小，
*!* ...由于抗锯齿设置的缘故，导致文本（通道）被画成了糟糕的蓝色
*!* 这就是光晕效果产生的原因
logpRectDest = CREATEOBJECT("gpRectangle",;
    0, 0, thisform.width, thisform.height)
logpRectSrc = CREATEOBJECT("gpRectangle",;
    0, 0, thisform.width/5, thisform.height/5)
THIS.gpgraphicsdb.DrawImagePortionScaled(logpBitmap,;
    logpRectDest, logpRectSrc, UNITPIXEL, .F.)

*!* 现在使用一个画笔以黑色填充文本轮廓来结束这个效果
logpBlackBrush = CREATEOBJECT("gpsolidbrush",0xFF000000)
GdipFillPath(this.gpgraphicsdb.gethandle(),;
    logpBlackBrush.GetHandle(), lnPath)
```

在这段代码的第一部分中，我建立了一个小的 **bitmap(logpBitmap)**，后面我将会把通道绘制在这个位图上面。前面我已经讲过了如何建立一个通道、并向通道添加一个字符串，那么现在就把注意力放到我建立 **logpHaloGraphics** 对象的那行代码上吧。如我在以前的几篇文章中所述，要在一个位图上绘制图形（在本例中是要绘制一个通道），我需要一个 **graphics** 对象：**logpHaloGraphics**。

前面我建议了让抗锯齿在实现光晕效果中出一份力。我已经把 **logpHaloGraphics** 的平滑模式和文本绘制方式设置为了抗锯齿。我还把插值模式设置为了高质量双立方来提高绘制的质量。

下一步是这个矩阵。我给通道添加的字符串是用的 **64** 点的一个字体大小，但我其实想要在这个位图上把它绘制的很小（**1/5** 大小）。为了这个目的，我给矩阵发送了 **.2**、**0**、**0** 和 **.2** 作为前面的四个参数来缩放 **graphics** 对象。这么做就是告诉 **GDI+**：任何我绘制到这个 **graphics** 对象的设备现场（**device context**）上的东西都应该以 **20%** 的宽度和 **20%** 的高度来绘制。

如果你已经看过了我在九月份文章（在 **VFP 9** 表单上的 **GDI+**：操作图像）中的示例，你也许会想看一下示例 **14**。把那个示例中的第一个和第四个微调控件（**spinner**）设置为 **.2**，你会看到其中的图像就被绘制为它原始大小的 **20%**了。

当已经在小位图上画出了通道、并填充了它之后，我删除了这个矩阵，并通过使用 **gdidoublebuffer** 对象的成员对象 **gpGraphicsdb** 的 **DrawImagePortionScaled** 方法，将这个位图绘制到脱屏位图上。该方法允许我定义一个目标矩形和一个来源矩形，这两个矩形分别用于接收缩小了的位图和还原为原始尺寸的位图（记住，**64** 点的文本已经被

绘制为它原始尺寸的 **20**%大小）。

这么做实现了什么目的？当抗锯齿文本的尺寸放大时，该通道在一次平滑其边缘及其周围的像素的尝试中被重新取样。因此而导致通道的边缘变得极为模糊而失真，而这正是我在建立光晕效果时想要的东西。想了解关于抗锯齿方面更彻底的解释以及它是如何工作的，请参考 **http://en.wikipedia.org/wiki/Anti-alias**。

前面代码中的最后一个步骤，是以黑色（或是在表单背景上的无论什么颜色）去重新填充定义了的图形通道。这次，我使用了 **gpGraphicsdb** 对象，这个对象没有被矩阵缩放过，所以这个通道（文本轮廓）被百分之百的填充了，给人的印象就好像是其中心被从光晕文本字符中踢了出来。

象在第一个示例中一样，我还在示例二表单上提供了多个文本框，以便你可以用一些自己的不同文本去尝试一下。你还可以试着修改一些代码（例如位图、矩阵的大小，以及画笔的宽度和颜色）来看看你还能建立什么别的效果。

介绍底纹笔刷：示例 3

在这个示例中，我将演示如何用一个底纹笔刷来实现一些非常令人惊讶的效果。有 **53** 种可供选择的底纹样式，而且在这个示例中我还提供了一个方便的微调控件以便你试用一下这些样式（见图 **2**）。

图 2、底纹笔刷可以被用于实现 53 种不同的文本填充样式

尽管在这个示例中的大部分代码都已经被解释过了，我还是想要指出其中对 **GDI+** 平台 **API** 函数 **GdipCreateHatchBrush** 的调用：

```
GdipCreateHatchBrush(INT(THISFORM.spnStyle.VALUE),;
    THISFORM.textforecolor, THISFORM.textbackcolor, ;
    @lnBrush)
```

这个函数接收一个底纹笔刷样式作为它的第一个参数（在下载的附件中的包含文件 **vfpgdiplustext.h** 中有一个完整的底纹笔刷样式常量的列表）、一个 **ARGB** 前景色作为它的第二个参数、一个 **ARGB** 背景色作为它的第三个参数、以及一个数值型输出参数作为它的第四个参数。当这个函数调用成功后，**lnBrush** 将代表一个有效的底纹笔刷句柄，可以被用于为 **GDI+** 平台 **API** 函数 **GdipDrawString()** 定义笔刷。

在这个 **GDI+** 系列文章中，我已经演示过了如何使用实心的笔刷、梯度线条笔刷，而现在轮到底纹笔刷了，但还有其它一些非常有用的笔刷还没讲过。下一个例子讲的就是其中的一种：纹理笔刷。

介绍纹理笔刷：示例 4

纹理笔刷非常强大，它让你可以使用已有的图像（或者你刚刚建立的图像）去以 **GDI+**

绘制对象（见图 3）。“只有天空才能限制它”正是对这个笔刷的一个非常准确的描述。当你运行示例 4 的时候，你将被提示去选择一个图形文件。我已经使用了 Windows 文件夹作为默认的目录，因为在这个文件夹里通常会多少有几个纹理图形文件，不过你可以选择任何图形文件。想要用一副你的家庭的照片来绘制文本吗？只要在运行示例 4 的时候选择这幅图像就行了。

图 3、纹理笔刷让你可以使用你硬盘上已有的一个图像文件去绘制图形（设置是文本）

象示例 3 一样，在这个例子中所涉及的大部分代码都已经讲解过了。这里是建立纹理笔刷以供 GdipDrawString() 使用的那行代码：

```
GdipCreateTexture(THISFORM.gpImage1.GetHandle(),;
    INT(THISFORM.spnWrapMode.VALUE), @lnBrush)
```

为了让纹理笔刷能够访问一个图像文件，你将需要为该文件建立一个 gpImage 对象，并将它的句柄作为第一个参数发送给 GDI+ 平台 API 函数 GdipCreateTexture()。

第二个参数是封装模式（wrap mode）。该参数影响纹理笔刷如何绘制并且/或覆盖这个图像。有五种封装模式：Tile（覆盖）、Tile Flip X(沿 X 轴翻转覆盖)、Tile FlipY(沿 Y 轴翻转覆盖)、Tile Flip XY（沿 XY 轴翻转覆盖）、还有 Clamp(强加)。我已经在这个示例表单上添加了一个微调框以便你尝试各种不同的封装模式，在 vfpgdiplusetext.h 包含文件中你也会找到一个封装模式内容的权威性列表。

发送给 **GdipCreateTexture()** 的最后一个参数是一个输出参数，一旦这个函数执行成功，则该参数中将包含着对纹理笔刷的句柄。然后就只剩下用刚建立的纹理笔刷绘制内容的事情了。

GDI+ 做 Unicode ： 示例 5

示例 **5** 虽然没有展示一个文本效果，但演示了一种在一个表单上绘制 **Unicode** 字符串的途径。即使在 **Visual FoxPro** 最早的版本中也已经提供了有限的 **Unicode** 支持。然而，通过使用有着强大的 **Unicode** 支持的 **GDI+**，我们可以克服这些限制中的部分（见图 **4**）：

图 4、GDI+允许 Visual FoxPro 开发人员去绘制 Unicode 文本

让我们先看一下涉及的代码，然后我将解释做了什么以及为什么这么做：

```
LOCAL logpSolidBrush, logpStringFormat, logpRect, ;
    lcChinese, lcHebrew, lcRussian, lcThai, ;
    logpFont, lcRectF

*!* 建立一些 Unicode 字符串
*!* 我使用了一个网站来为这个示例进行转换
*!* 下面是该网站的链接：
*!* weber.ucsd.edu/~dkjordan/resources/unicodemaker.html

*!* 简体中文 Unicode 字符串
lcChinese = STRCONV("Chinese: ", 5) ;
    + BINTOC(0x4EAC,"SR2") + BINTOC(0x4EC5,"SR2");
    +BINTOC(0x5C3D,"SR2") + BINTOC(0x5F84,"SR2");
```

```
    +BINTOC(0x60CA,"SR2") + BINTOC(0x740E,"SR2")

*!* Unicode 希伯来字符串
lcHebrew = STRCONV(CHR(13) + CHR(10) + "Hebrew: ", 5);
    +BINTOC(0x05D3,"SR2") + BINTOC(0x05D1,"SR2");
    +BINTOC(0x05DC,"SR2") + BINTOC(0x05D1,"SR2");
    +BINTOC(0x05EA,"SR2") + BINTOC(0x05D7,"SR2")

*!* Unicode 俄语字符串
lcRussian = STRCONV(CHR(13) + CHR(10) + "Russian: ",5);
    +BINTOC(0x0439,"SR2") + BINTOC(0x043E,"SR2");
    +BINTOC(0x0442,"SR2") + BINTOC(0x044F,"SR2");
    +BINTOC(0x0437,"SR2") + BINTOC(0x0432,"SR2")

*!* Unicode 泰语字符串
lcThai = STRCONV(CHR(13) + CHR(10) + "Thai: ", 5) ;
    + BINTOC(0x0E28,"SR2") + BINTOC(0x0E17,"SR2") ;
    + BINTOC(0x0E40,"SR2") + BINTOC(0x0E30,"SR2") ;
    + BINTOC(0x0E23,"SR2") + BINTOC(0x0E1B,"SR2")

*!* 为了绘制上面的字符串，要选择一个 Unicode 字体
*!* 并且还要选择一个格式
logpFont = CREATEOBJECT("gpfont", "Arial Unicode MS", ;
    24, FontStyleBold, UnitPoint) && SimSun
logpStringFormat = CREATEOBJECT("gpstringformat", 0)

*!* 建立一个矩形，用来指定字符串将被绘制于的位置
logpRect = CREATEOBJECT("gpRectangle", 25, 25, ;
    THISFORM.WIDTH, THISFORM.HEIGHT)
lcRectF = logpRect.GdipRectF

*!* 为字符串选择一个笔刷和颜色- 在本例中，我选择的是黑色
*!* 0xFF000000 是黑色-不透明的 ARGB
logpSolidBrush = CREATEOBJECT("gpsolidbrush", 0xFF000000)

*!* 绘制 Unicode 字符串
*!* 注意长度被除了 2，
*!* 这是因为每个 Unicode 字符要占用两个字节的空间
GdipDrawString(THIS.gpgraphicsdb.GetHandle(), ;
    lcChinese + lcHebrew + lcRussian + lcThai, ;
    LEN(lcChinese + lcHebrew + lcRussian + ;
    lcThai)/2, logpFont.GetHandle(), ;
    lcRectF, logpStringFormat.GetHandle(), ;
    logpSolidBrush.GetHandle())
```

```
*!* 清理
RELEASE logpSolidBrush, logpStringFormat, logpRect
STORE .NULL. TO logpSolidBrush, logpStringFormat, logpRect
```

要完成这个示例有大量的途径可以选择，但我选择了使用字符的十六进制等价替代品和 **Visual FoxPro** 的 **BinToC()** 函数来建立 **Unicode** 字符串。第一个任务，是去找到一些字符串的十六进制等价替代品。我决定使用一个拥有在线转换器的网站：**http://weber.ucsd.edu/~dkjordan/resources/unicodemaker.html.**

当我有了这些十六进制值以后，我简单的把它们每一个转换成二进制的字符表示，然后这些二进制字符表示就可以被连接起来生成所需的 **Unicode** 字符串。在前面示例中我使用 **GdipDrawString()**函数的地方，你会看到我使用了 **Visual FoxPro** 的 **StrConv()** 函数，当把单字节字符串作为第二个参数传递给该函数的时候，将它们转换成它们的 **Unicode** 等价替代品。

不过，在本例中，这些字符串已经被转换过了。我简单的把这些字符串连接起来去生成期望被绘制到表单上去的文本。应该指出的是：你必须使用一个 **Unicode** 字体以正确的绘制文本。我选择了“**Arial Unicode MS**”，但其它的 **Unicode** 字体也是可以的，并且某些字体被明确的连接起来以翻译一个特定方言的字符集（译者注：由于译者水平所限，这段话实在不好翻，附上原文，请大家指正：**and certain fonts are specifically geared toward rendering a particular dialect's characters**），例如 **SimSun**，它会生成非常高质量的简体中文字符。

在这个示例中，我已经提供了中文、希伯来文、俄文、以及泰文的例子，但我鼓励你去找网上更多的 **Unicode** 字符，并且使用象 **ucsd.edu** 这样的网站提供的转换器服务、将它们转换为十六进制值以进行测试。此外，你也许想要改变这个例子中的字符串格式，以绘制垂直的字符串或者去掉封装。在 **vfpgdiplustext.h** 文件中有一个完整的字符串格式常量列表。例如，你可以通过简单的修改这一行用于建立 **gpStringFormat** 对象的代码来将文本改成垂直的：

```
logpStringFormat = CREATEOBJECT("gpstringformat", 2)
```

你也许会奇怪，在没有任何一行“**#INCLUDE vfpgdiplustext.h**”这样的代码的情况下，我是如何访问在 **vfpgdiplustext.h** 包含文件中的常量的。这是因为在 **Visual FoxPro** 中有一个功能允许你为每个表单定义一个包含文件。要看看这个功能，请打开任意一个示例表单，然后单击“**Froms**”系统菜单下的“**Include File**”菜单项。这会打开一个该表单的包含文件对话框。当在该对话框中指定了一个包含文件的时候，表单中的每一个方法就都能够访问该包含文件中的常量了。

目前就说这么多了

我希望这两篇关于 **GDI+** 文本的文章已经让你对能在一个 **VFP** 表单上可以对文本做些什么有了更深的理解。此外，记住在 **Visual FoxPro 9** 的报表引擎也提供了对 **GDI+** 绘图层的访问，因此任何我在这篇以及之前的几篇文章中演示国的技术或者代码也都可以用在报表上。**GDI+**几乎可以绘制任何东西，而且对 **_gdiplus.vcx** 类库和 **GDI+** 平台 **API** 的良好理解是在任何 **Visual FoxPro** 程序员技巧集中的一个基本元素。

在我下一篇文章中，我将提供大量的用于对形状、弧形、线条等等对象进行工作的技术。

下载：**511BOYD.ZIP**

看哪！这是个陷阱...

作者：**Andy Kramek & Marcia Akins**

译者：**fbilo**

继续上个月关于报告 Bug 的讨论，Andy Kramek 和 Marcia Akins 正在研究错误捕捉。关于这个主题已经有大量的文章了，而从 8.0 开始的 TRY...CATCH 极大的简化了程序员在这方面的工作。然而，在一个产品的环境下、在运行时的错误处理是一个完全不同的问题。

玛西亚：上个月你曾提到，当应用程序中发生一个运行时错误时，你的错误处理器会给你发送一封 **Email**。详细情况是怎么样的？

安　迪：它捕捉、并且处理任何任何导致 **VFP** 触发一个错误的事情。不管这是否是一个代码错误、一个由于数据损坏或者系统崩溃导致的错误、或甚至是一个由于在代码中使用了 **ERROR** 命令而生成的自定义错误。

玛西亚：你说的“处理”一个错误是什么意思？

安　迪：问得好！程序员们总是谈论着“错误处理器”，可实际上我们真正能够处理的错误却只有很少几个。

玛西亚：嗯？

安　迪：想一下这个问题：假设在你的应用程序运行的时候发生了 **12** 号错误（变量没有找到）。在运行时你为了恢复正常有什么可以做的吗？

玛西亚：什么都干不了，我想。你所能做的只有关闭这个应用程序，然后再打开试试，尽管我听说所谓神经错乱的定义就是不停的尝试着做同样的事情一遍又一遍、然后期望会得到不同的结果。

安　迪：那么第 **9** 号错误（数据类型不匹配）或者第 **1234** 号错误（下标超出了定义了的范围）呢？

玛西亚：好吧，我明白你的意思了。你没法真正处理这些类型的错误。

安　迪：事实上，由 **VFP** 报告的所有错误中，只有很少几个是你可以重试（指在错误对话框中按 **Retry** 按钮、或者在代码中执行一个 **Retry** 命令）的。例如，错误号 **1589**(表或者行缓冲要求 **SET MULTILOCKS** 是被设置为 **ON** 了的)就是一个

——你可以简单执行 **SET MULTILOCKS ON** 然后执行一个 **RETRY**。

玛西亚：当然，这样的情况还有 **125** 号错误（打印机没有准备好）——我们可以简单的告诉用户去检查打印机！在这种情况下就不需要发送一封 **Email**、或者把错误记录到日志中去了。

安　迪：是的，而且还有一个错误是我们可以 **Retry**、但是又显然会想要记录到日志中的。它就是 **1545** 号错误（表别名为**"name"**的缓冲中包含有未提交的改动）。我们简单的放弃改动、然后 **Retry**。

玛西亚：那是个当你试图刷新一个含有未决改动（**pending changes**）的视图时会发生的错误。可你难道不应该在放弃改动前先问一下用户吗？

安　迪：不是这种情况。由于用户的直接操作（比如视图退出一个编辑窗口）而出现的这类情况不能算是问题，这类问题是你能够轻松处理的。我谈的是代码中出现的问题。

玛西亚：哦，我知道了。我曾碰到过这种错误，当时我试图在保存了改动以后立即去刷新一个参数化视图。很明显，有某些不属于用户输入的东西弄脏了缓冲，所以在这种情况下我就直接执行一个无条件的 **TABLEREVERT()**。

安　迪：正是如此。但作为一个程序员来说，我当然希望知道曾经发生过这种事情、以便我能找出为什么会发生。

玛西亚：除了这里很少的几个错误以外，我们能够做什么呢？

安　迪：你自己已经说过了。我们必须捕捉到错误、好好的告诉用户、记录下错误的细节、然后尽可能安全的关闭应用程序。

玛西亚：你说的“好好告诉用户”是什么意思？

安　迪：好吧，你不会真的想让用户看到经典的 **VFP** “取消、挂起、忽略”对话框或者一个象“找不到变量**'LCNAME'**”（这是你能从 **12** 号错误中所能得到的全部信息）这样含糊的消息。如果有象图 **1** 这样的东西的话会好得多。

图 1、用户友好的错误捕捉

玛西亚：是的，我更喜欢这个。那么它是怎么完成所有的工作的呢？还是让我们来看看代码吧！

安　迪：错误处理器位于一个 **PRG** 文件中，被定义为一个基于 **Session** 基类的 **PRG** 类中，当它被调用的时候会建立它的实例。我们象下面这样启动这个错误处理器：

```
ON ERROR DO errorhandler
```

玛西亚：你采用一个 **Session** 类是为了让它不会干扰到别的那些表，对吗？但这样的话，又该怎么让它自己进入到正确的数据工作期中呢？

安　迪：当它被调用的时候，建立它的实例的程序会传递给它当前的数据工作期 **id**，就象这样：

```
oHandler = CREATEOBJECT( "generrhandler", SET( "Datasession" ))
```

玛西亚：那么是由调用它的程序来负责建立和释放处理器对象的实例喽？

安　迪：是的。在这个处理器类自身中的所有代码是从 **Init()** 开始运行的。首先，它检查环境中是否真正有一个要处理的错误，然后，它保存当前的 **ON ERROR** 设置（这当然就是错误处理器自己了）、并禁用 **VFP** 的错误处理器。

玛西亚：你的意思是执行一个 **ON ERROR *** 命令，从而把这个错误处理器注释掉？

安　迪：是这样。然后它打开它自己的错误日志，该日志放在一个自由表里。

玛西亚：那么如果不存在这么一个表、或者由于路径错误找不到这个表的话该怎么办？尽管我们现在已经禁止了可能会发生的任何错误，但我们的错误处理器现在自己也不能正常工作了，而且我们没法知道关于这些错误的信息。

安　迪：这不是问题。如果这个错误处理器找不到表，它就简单的自己新建一个。打开了日志以后，它会切换到错误所发生于的数据工作期，并调用 **HandleError()** 方法，该方法会完成所有的工作。

玛西亚：我明白了。你是在尽可能多的取得错误发生时系统状态的信息。这里的窍门是完成这个任务而又不对状态产生影响。哦，我现在理解你为什么把这个错误处理器用到的变量声明为 **PRIVATE** 而不是 **LOCAL** 了。这么做的话，只有那些真正拥有值的变量才会被建立起来（如果你把它们声明成了 **LOCAL**，那么它们立刻会被分配以一个值 **.F.**并出现在内存的堆中）。

安　迪：这也是为什么在 **HandleError()** 中的第一个操作是去取得内存中的内容的原因……这是状态中最不稳定的部分了。幸运的是，它也非常的简单……我只是建立了一个临时文件名，并将它保存到一个属性中，然后使用代码去格式化文本并输出它。

```
SET CONSOLE OFF
```

```
SET ALTERNATE TO ( This.cFileName )
SET ALTERNATE ON
*** 拷贝内存
? DIVLINE
? "*** Contents of memory ***"
? DIVLINE
LIST MEMORY LIKE *
```

玛西亚：但为什么你用上了 **LIKE *** 子句？

安　迪：哦，我对 **VFP** 自己的所有系统变量——就是以下划线开头的那些——不感兴趣，象这些：

```
_ALIGNMENT Pub C "LEFT "
_ASCIICOLS Pub N 80 ( 80.00000000)
_ASCIIROWS Pub N 63 ( 63.00000000)
_ASSIST Pub C ""
```

玛西亚：**_Assist**？天知道它为什么还留在这里。它只在 **Foxbase+** 的时代才有用，已经无效了几十年了。

安　迪：我猜它的存在是为了向后的兼容性，不过所有的系统变量我们都不需要。下一件事情是去弄清楚我们该如何去处理错误——这是在 **AssessError()** 方法中完成的。

玛西亚：我看到在这个方法里面你把所有的错误都按逻辑进行了分组，以便你能向用户显示一个恰当的错误消息。例如，我看到你把 **56** 号错误单独放在一个它自己的类别中，并当发生这个错误的时候显示下面这条消息：

```
The Application has run low on disk space. Try increasing the space on the hard disk by removing unused files and restart the application. Contact your IT support person if you are unsure of what to do next.
（应用程序的运行缺少磁盘空间。请通过删除无用的文件来增加硬盘上的空间，并重启这个应用程序。如果你不确定下一步该怎么做，请联系你的 IT 支持人员。）
```

哼嗯，这可会是一个相当危险的消息。多年前，我有一个客户就碰到了磁盘空间不足的问题，所以他删除了一些磁盘上最大的文件。不幸的是，这些最大的文件恰好是运行那个系统的文件，所以该系统就毫不奇怪的不再运行了。

安　迪：但你可以把这个消息弄成你喜欢的任何东西。这里重要的是：这个消息应该与错误相关、并且对用户来说不危险。例如，我已经把第 **1711**、**1726**、**2027**、还有 **2028** 号错误都分在了一个我称之为 **API** 错误的逻辑组中。当 **1711** 号错误发生的时候，下面这个消息会被显示给用户：

```
Windows has reported an error from one of its system Files. This has nothing to do with the application, and if the problem persists after restarting your PC, you should seek
```

```
advice from your IT Support Person.
```

而不是：

```
    API library revision mismatch. Rebuild library.
```

玛西亚：那太好了!此外，既然错误处理器把这个错误记入了日志并向你发送了一封 **Email**，你就有了真正的错误号和消息并能采取相应的行动去修正它了。

安　迪：**AssessError()**还会设置返回值，它就定义了要采取的下一个行动。如我前面所述，只有极少几个错误是你真正可以在运行时恢复的，所以默认的行动就是将错误记入日志并退出。

玛西亚：我能够理解。我猜当你碰到 **125** 号错误（打印机没有准备好）或者其它特殊错误的时候，你会把操作设置为 **“Retry”**、把记录错误日志标志设置为**“false”**。

安　迪：没错。该方法象下面这样返回一个参数对象给 **HandleError()**：

```
    loPObj = CREATEOBJECT( 'empty' )
    WITH loPObj
        AddProperty( loPObj, 'cMessage', lcStr )
        AddProperty( loPObj, 'cAction', lcAction )
        AddProperty( loPObj, 'lLogErr', llLogError )
        AddProperty( loPObj, 'lQuitApp', llQuitApp )
    ENDWITH
    RETURN loPObj
```

玛西亚：我看要做的下一件事情是去获得系统状态。从本质上来说，这跟拷贝内存变量是一样的，除了我们使用的是 **LIST STATUS** 命令而不是 **LIST MEMORY** 以外。

安　迪：是的，但这一点特别重要，因为这是我们取得可靠的部分信息的惟一位置——比如哪些 **DLL** 被加载了、关于索引和关联（**Relation**）的信息、以及所有的 **SET** 命令设置。这些信息对于准确的判定错误发生在何种条件下是至关重要的。

玛西亚：我没有意见。下一步你好像是在收集所有被打开了的表的信息——但是难道你没有从就在刚才所做的拷贝状态那一步中得到它吗？

安　迪：没有。那个状态列出了所有已经打开的表，但这个方法会从每个已打开表的当前记录中返回每个字段的值。它还列出了缓冲模式和每个字段的改动状态。任何带有未决改动的字段都会以一个前置的星号被标记出来，象这样：

```
    ---------------------------------------------------
    *** Open Table Information ***
    ---------------------------------------------------
    TABLE: XXXPASS (Buffering: Optimistic Row)
    Record #2
    C DEPARTMENT ..... P
```

```
* C NAME ..... DAVE
```

玛西亚：这的确很方便，并且我特别喜欢这里的改动状态。说不清有多少次我从用户那里听到“但我什么都没动”这种话了。你还取得了什么别的东西？

安　迪：最后一块得到的是我们用 **ASTACKINFO()**取得的堆栈信息。

玛西亚：这给了我们程序运行链以及导致错误发生的代码行号和真正的代码，是吗？

安　迪：差不多。你总是能得到程序名称和行号，但只有当源代码可用的时候你才能得到代码（就像 **MESSAGE(1)**）。当我们得到了这些信息以后，就可以写日志了。

玛西亚：就是向表中插入一条记录罢了，是吗？

安　迪：是的，不过还要加上一个头（**header**）以便让这一条数据项更有意义。这里是最后的日志数据项的一个例子（当然，省略了细节）：

```
---------------------------------------------------
Application Error occurred at 08/28/2005 12:21:21 PM
At Tightline Computers Inc
(Email: andykr@tightlinecomputers.com)
(Phone: (330) 785 9078)
To the user logged in as ACS-SERVER # Administrator
---------------------------------------------------
The error occurred in: frmsystem.validatecontrols
The VFP Error Number was: 12
And the VFP Error Text was: Variable is not found.
---------------------------------------------------
The User Message displayed was: The Application has
reported an internal error identified as Error #: 12
---------------------------------------------------
*** Contents of memory ***
---------------------------------------------------
<detail here>
---------------------------------------------------
*** System Status ***
---------------------------------------------------
<detail here>
---------------------------------------------------
*** Open Table Information ***
---------------------------------------------------
<detail here>
---------------------------------------------------
*** Calling Stack ***
---------------------------------------------------
<detail here>
```

玛西亚：这已经相当全面了。还有什么剩下的吗？

安　迪：除非我们将要 **Retry**，否则就必须尽可能温和的关闭应用程序。这是在**CloseApp()**方法中完成的，该方法遍历所有数据工作期，回滚所有打开的事务、放弃所有未决的改动、然后关闭每一个表。最后，它关闭所有打开的表单，并执行一个 **CLEAR EVENTS**。

玛西亚：我的应用程序对象已经有了一个完成这些任务的方法了。难道我真的还需要再加这么一个吗？

安　迪：当然不。这正式为什么这段代码上放在一个方法中的原因——你可以从这里调用你喜欢的任何东西。最后一件事情，是向用户询问是否要发送一封关于该错误的 **Email**。

玛西亚：那么你又是怎么实现的呢？是用自动化吗？

安　迪：不。请记住，我们是在一个错误处理器中。这就意味着系统正处于一种不可靠的状态下。所以我更倾向于调用 **Windows API** 并使用 **ShellExecute()**函数来发送 **Email**，因为在这个时候它的危险会更少一些。

玛西亚：挺合逻辑的。用这种办法你就可以直接使用在 **Windows** 中注册为 **e-mail** 客户端的任何东西，它所需要的只是下面这一小块“从 **#DEFINES** 中读入信息到程序中”的代码：

```
*** 首先是 TO 和 Copy To 地址
lcMail = "mailto:" + DEF_EMAILTO
IF NOT EMPTY( DEF_CCMAILTO )
    lcMail = lcMail + ;
    IIF( AT( '?', lcMail ) = 0, "?", "&" ) ;
    + "CC=" + DEF_CCMAILTO
ENDIF

*** 现在把这些值正确的连接起来
lcMail = lcMail + ;
    IIF( AT( '?', lcMail ) = 0, "?", "&" ) + ;
    "Subject=" + DEF_EMSUBJECT
lcMail = lcMail + ;
    IIF( AT( '?', lcMail ) = 0, "?", "&" ) + ;
    "Body=" + tcDetails

*****************************************
*** 检查 API 库是否已经启动，若没有
*** 则打开它
*****************************************
```

```
lnRes = ADLLS( laJunk )
IF lnRes = 0 OR ;
   NOT ( ASCAN( laJunk, 'Send', 1, -1, 1, 15 ) > 0)
   *** 没有可用的函数
   DECLARE INTEGER ShellExecute IN shell32 ;
   INTEGER lnhwnd, STRING lcOperation, ;
   STRING lcFile, STRING lcParameters, ;
   STRING lcDirectory, INTEGER lnShowCmd
ENDIF

*** 发送邮件
lnRes = ShellExecute( 0, "open", lcMail, "", "", 3 )
```

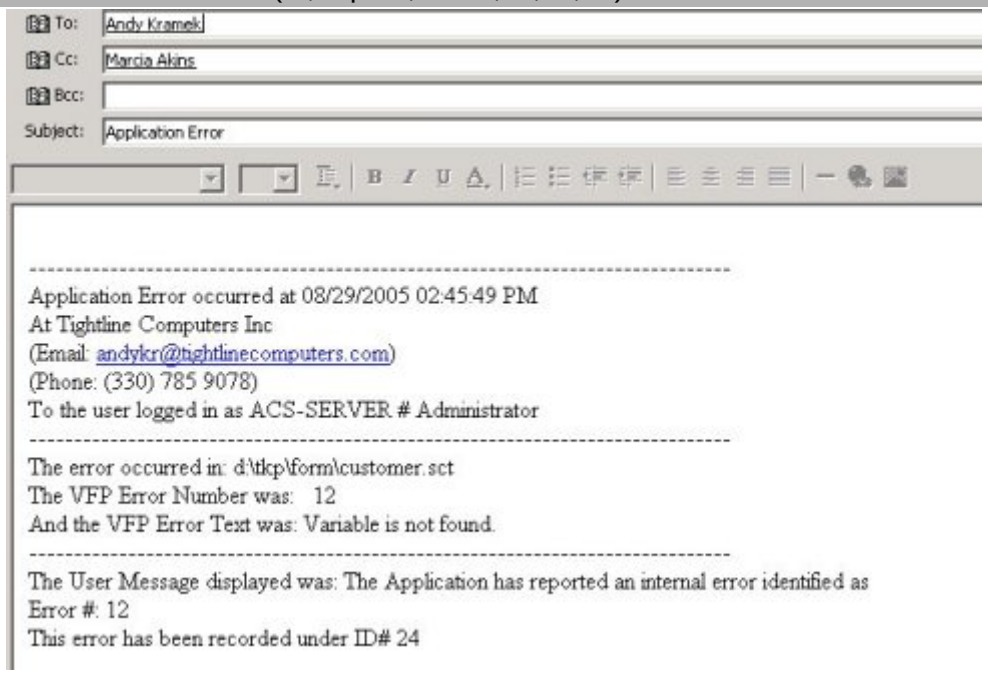

图 2、由错误捕捉生成的 Email 邮件

安　迪：我们释放这个错误处理器对象、并在外部程序中执行相应的操作（退出、取消、或重试）。这就是这里我们需要的所有东西：

```
RELEASE oHandler
DO CASE
   CASE LOWER( lcToDo ) = 'quit'
      *** 清理并立即退出
      CANCEL
      CLEAR EVENTS
      CLEAR ALL
      RELEASE ALL EXTENDED
      QUIT
   CASE LOWER( lcToDo ) = 'cancel'
      *** 清理并回到 VFP
      CANCEL
```

```
            CLEAR EVENTS
            CLEAR ALL
            RELEASE ALL EXTENDED
            RETURN .F.
        OTHERWISE
            *** 我们将要 Retry
            RETRY
    ENDCASE
```

玛西亚：嗯，相当全面了，而且我喜欢这个可以在用户碰到错误的时候收到 **Email** 的主义。现在当任何错误发生时，我可以在用户打电话找我之前先打电话给用户了。图 **2** 显示了一个由错误捕捉发送的一封 **Email** 的例子。

安　迪：老规矩，代码包含在这篇文章附带的下载文件中。

下载：**511KITBOX.ZIP**