2005 年第 10 期

任何应用程序的测试阶段都是棘手的。而让测试员和最终用户精确的报告出了什么错（任何比一个“**Hello World**”复杂一点的应用程序都会出错）更是棘手。这个月，**Andy Kramek** 和 **Marcia Akins** 讨论了如何最好的帮助他们的测试人员们通过以一种使得问题的识别、指定优先级、结束变得尽可能无痛的方式帮助开发团队。

调用你自己的打印作业

作者：**Lisa Slater Nicholls**

译者：**CY**

VFP 可以处理单个 REPORT FORM 命令，或者是一组带 NOPAGEEJECT 的 REPORT FORM 命令，以此来作为一个打印作业，但是有时你在打印过程中想发送另外的打印指令。Lisa Slater Nicholls 将为你展示如果利用 VFP9 的报表增强性能来解决这个常见的问题，或是其他与此类似的，在你直接控制下的打印作业。

在先前关于生成 **PDF** 的文章里（**"PDF Power to the People," June 2005**），我说过我的定制类 **Reportlistener** 可以使用"特别魔法"以保存和恢复打印机设置。在 **PDFListener** 类的两个 **PROTECTED** 方法里，**LoadPrinterInfo()** 和 **UnloadPrinterInfo()**，我利用 **VFP9** 的增强性能来调用 **SYS(1037,2)** 和 **SYS(1037,3)**，并且我建议你小心调用这些方法以相应的"推入"和"弹出"的顺序，以保存和恢复 **VFP** 的打印机环境设置。

也会有其他情况使得它最好不要过多的操纵 **VFP** 打印机环境。取而代之的是，你可以选择控制你自己的整个打印作业，并发送报表的内容给这个打印作业。在本文里，我将使用 **VFP9** 里新的 **SYS(1037)**调用，以及一些简单的 **Windows API** 调用来处理一个常见的要求：在打印报表时切换打印机纸盒、方向，或是其他打印特性。

在开始前，我向你保证我并不是 **C++**程序员。当我说"非常简单的 **Windows API** 调用"，我的意思就是"非常的简单"。为达到这个结果，你只需要认真的想想 **VFP** 能做什么，以及你想要在你的 **VFP** 应用程序里达到什么结果，而无需知道 **Windows** 是如何工作的。

如何给打印机发送你的想法

本文的示例类叫做 **PrintJobListener**。类似于 **PDFListener**，它派生于 **FFC** 里 **_ReportListener** 的中间类层，**OLEAwareReportListener**。参考其他项"**PrintJobListener** 的家谱明细"的其他信息和建议。

为实现它的工作，**PrintJobListener** 以 **ListenerType** 为 **3** 来运行所有报表，它告诉报表引擎存储所有的页图，就如同它是作为预览。类型 **3** 指定在运行后不自动显示活动，

因此你可以利用它来选择预览或是发送图像到不同的目标设备，比如图像文件。

PrintJobListener 允许你在报表运行时指定一系列的打印设置。在报表开始运行前，**PrintJobListener** 设置 **FRX** 和报表引擎，以你先前所指定的打印机指令信息，以尽可能的匹配你的打印作业请求。在运行后，当报表引擎准备好内容，**PrintJobListener** 创建一个打印作业并发送页面到打印机，为每一页按照需要插入你的打印机指令（见图 **1**）。

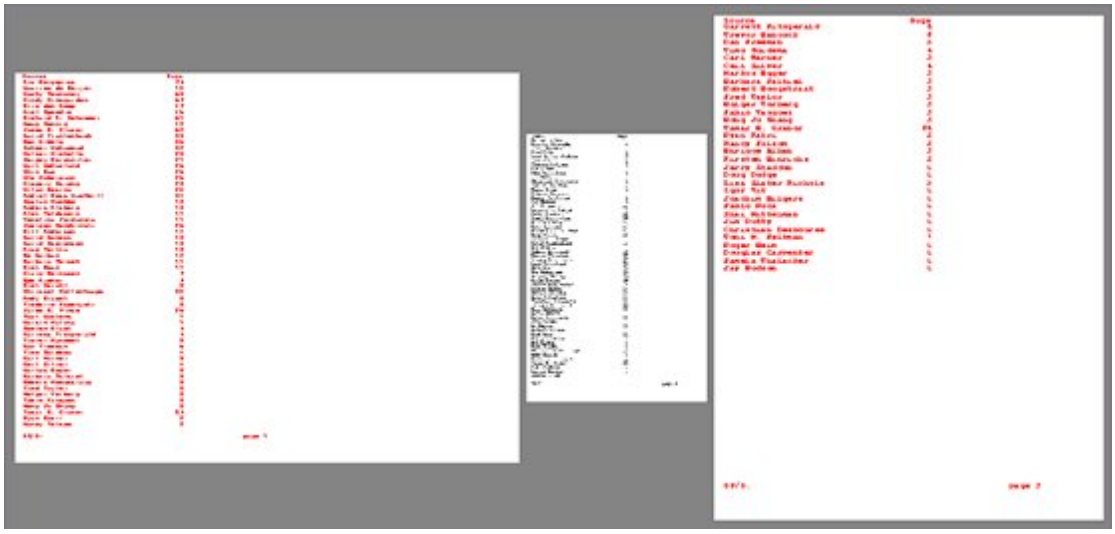


图 1：**PrintJobListener** 为你提供了在同一个打印作业里单页或多页的多组的打印机版面设置的良好细部控制。

那是个大图。它是怎么发生的？

提供指令给 PrintJobListener

为发送指令给任意类型的打印机，**PrintJobListener** 必须为你所需要的指定一个方法。这个类展示了一个公共属性，**PrinterSetupTable**，它用于存储多种打印机设置的 **DBF** 表名。你可以通过调用 **SYS(1037,1)**来填充这个表。

虽然这个表是标准的 **FRX** 格式，属性的默认值是“**PrintSetups.DBF**”，并且，如果你对你选择的表名不提供扩展名，**PrintJobListener** 将强制扩展名为 **DBF**。我已经这样做，以避免有人会认为这是个可运行的 **FRX**，因为它包含了非标准 **FRX** 版面记录。**SYS(1037)** 和打印作业的有效列是 **Expr**、**Tag** 和 **Tag2**，但是 **SYS(1037)**要求所有的标准 **FRX** 列在表内，并以 **EXCLUSIVE** 来存取表和运行。

你在这个表内指定不同的记录，通过在 **Name** 列加入对应记录的标识，这个列是 **SYS(1037)**所忽略的。**PrintJobListener** 利用备注字段 **Name**，而不是 **10** 字符的字段 **UniqueID**，因为这些标识是可变和可读取的。

当你准备运行报表时，你使用第二个公共属性，**FRXPrintSetups**，来保留集合对象。在集合里的每个项目的值是一组页的开始页码。在集合里每个项目的关键值将会是匹配于

你在你的表里 **Name** 字段的内容，不区分大小写。

PrintJobListener 严格按照集合顺序来处理集合里的每一个项目，因此按相应的顺序来使用集合的 **Add** 方法。例如，在 **PRINTJOBLISTENER.PRG** 里的测试代码如下：

```
LOCAL m.ox, m.oy

m.ox = CREATEOBJECT("PrintJobListener")
m.oy = CREATEOBJECT("Collection")
m.oy.Add(1,"FirstPage")
m.oy.Add(2,"SecondPage")
m.oy.Add(3,"OtherPages")
ox.FRXPrintSetups = m.oy
```

因为 **Name** 是一个备注字段，你可以包含每个打印设置的预期用途的更多信息，比如它所适合的报表或报表分组。对于你所加入的每一个打印机设置，**PrintJobListener** 应用相应的打印机指令，然后发送页面到打印机，一直到达到下一个设置的开始页。当它完成集合里的最终设置时，它处理所有的页一直到它达到这个报表运行的报表引擎所接收的 **OutputPageCount** 值。

可以确定，这是个处理页范围的简单想法。你可以提出你自己的存储和应用页范围指令的方法，只要你理解了“主要事件”：如何存储和应用实际的设置到打印作业。

确保正确的条件

正如你所期望的，**PrintJobListener** 的 **LoadReport** 事件代码设立了运行的条件：

```
PROCEDURE LoadReport()

IF NOT THIS.StripAndSaveFRXSetup(THIS.CommandClauses.FILE)
    THIS.DoMessage(;
        FAILED_COULD_NOT_REMOVE_FRXSETUP_LOC,;
        MB_ICONSTOP)
    RETURN .F.
ENDIF

IF THIS.FRXPrintSetups.Count = 0
    THIS.FRXPrintSetups.Add(1, "AllPages")
ENDIF
IF THIS.MultipleSetupsOnePrinter(THIS.FRXPrintSetups)
    THIS.Setup()
ELSE
    THIS.DoMessage( ;
        FAILED_COULD_NOT_USE_PRINTSETUPS_LOC,;
```

```
            MB_ICONSTOP)
        RETURN .F.
    ENDIF

    RETURN DODEFAULT()

    ENDPROC
```

首先，类检查了你所处理的 **FRX** 或 **LBX** 文件的打印机设置指令。如果有，就把它移去并作保存以便后面恢复。如果报表或标签是只读的并且包含打印机环境数据，或者 **PrintJobListener** 因任何原因无法从 **FRX** 或 **LBL** 移去打印机环境细节，它将不会继续运行报表。

在报表运行的过程中，如果你喜欢，你可以创建一个可读写的报表副本，并把你的副本与运行的报表交换报表名（在 **VFP** 文档的 **LoadReport** 标题里有一个示例展示如何实现它）。对于 **PrintJobListener**，我确定如果你在只读的表里嵌入这样的指令，就如你所希望的那样，那么，报表或标签就可能不太适合于打印作业处理。

StripAndSaveFRXSetup(tFRXFile)，这个方法处理在你的报表或标签里的任何打印机环境值，保存打印机环境数据到保护成员属性。当你研究这个方法时，注意 **StripAndSaveFRXSetup** 将以它是否确实保存设置信息来表明一个成功的结果。如果没有设置信息被保存，那么当报表是只读或可读写时是没有问题的。如果报表包含设置信息，并且是只读的，那么它将返回一个失败的结果（见图 **2**）。

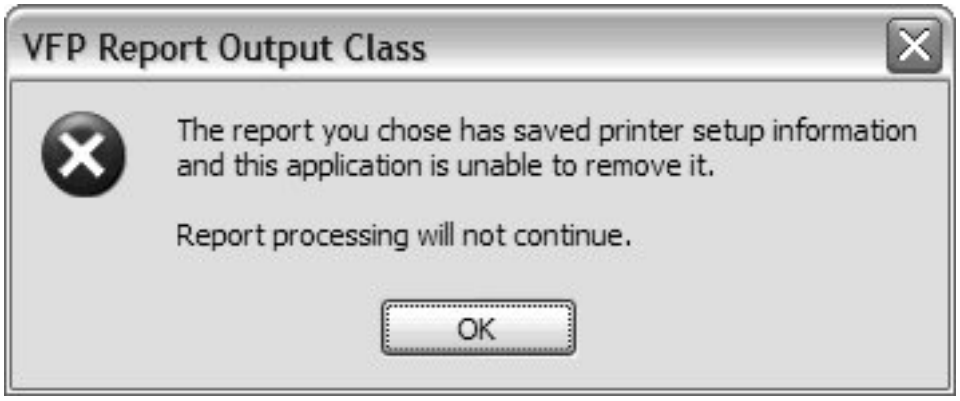


图 2：如果报表包含打印机设置信息并且是只读的，报表将不能继续运行。

你或许会说，这个公共方法是被设计为在报表运行之外时使用，就象是在 **LoadReport** 事件里。它有一个直接的组合方法，**RestoreSavedFRXSetup(tFRXFile)**。你可以利用这两个方法做试验来传递打印机设置信息，在 **FRX** 之间，或在 **FRX** 与打印机设置存储表之间，如果你愿意。

接着，**PrintJobListener** 的 **LoadReport** 代码检查它的 **FRXPrintSetups** 集合属性的相应内容。如果你没有提供任何指令，它将加入一个单一的项目到集合内以处理你的报

表所有页。

作为最后的预备步骤，它调用了 **MultipleSetupsOnePrinter** 方法来确认其打印机设置表是否可用，并且你的指令是否在表内。如果表不存在，**MultipleSetupsOnePrinter** 将能够创建表。实现这个任务的辅助方法所使用的代码，我推荐给任何类似的需求；这里并不需要维护一个“哑”报表来作为模型，并且对于任何普通 **FRX** 记录都是轻松的结果。

```
m.lcSafety = SET("SAFETY")
SET SAFETY OFF
CREATE CURSOR (THIS.FRXTempAlias) (onefield l)
CREATE REPORT (THIS.PrinterSetupTable) ;
    FROM DBF(THIS.FRXTempAlias)
USE IN (THIS.FRXTempAlias)
USE (THIS.PrinterSetupTable) ALIAS (THIS.FRXPrintSourceAlias) EXCLUSIVE
ZAP
IF m.lcSafety = "ON"
    SET SAFETY ON
ENDIF
```

PrintJobListener 在它的处理过程中这个时刻试图独占打印机设置表。如果这个表先前已经打开，将无法独占使用，因此它只能共享打开设置表。

如果表可用，**MultipleSetupsOnePrinter** 将试图在 **Name** 字段里查找你指定的设置名。如果表不可用，或者如果它已经独占打开，这个方法将为你或你的用户提示一个机会以在运行时加入新的设置（见图 **3**）。

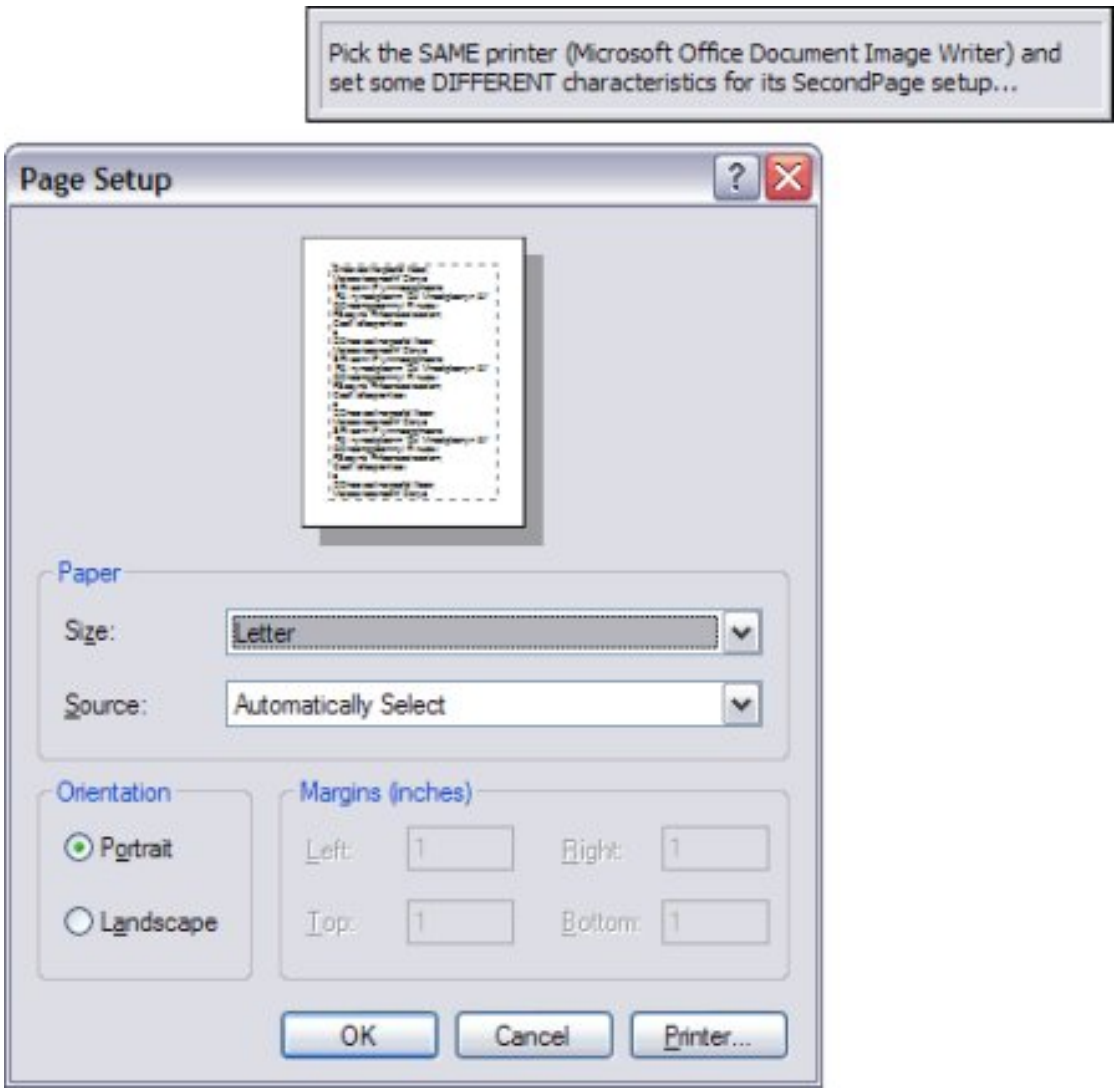


图 3：你可以在运行时为同一个打印机加入新的设置。

下面节选展示了 **SYS(1037,1)**如何实现这个次要的奇迹。这个代码在一个循环里处理了所有集合成员，在它处理了你的集合里的第一个项目后。处理第一个项目的代码类似如下：

```
m.lcSetup = toSetupList.GetKey(m.liIndex)
GO TOP
INSERT BLANK BEFORE
GO TOP && probably not required, doesn't hurt
THIS.DoStatus(PICK_PRINTERSETUP2_LOC)
SYS(1037,1)
DO WHILE .T.
    IF THIS.GetPrinterSetupDeviceInfo(Expr) == m.lcPrinter
        REPLACE Name WITH m.lcSetup
        EXIT
    ELSE
        BLANK FIELDS Expr, Tag, Tag2
        THIS.DoMessage(PICK_SAME_PRINTER_ALL_SETUPS_LOC, ;
```

```
                MB_ICONEXCLAMATION)
            SYS(1037,1)
        ENDIF
    ENDDO
```

我特别希望你观察我对 **INSERT BLANK BEFORE** 语句的使用。是的，我以前并没有找到足够的兴趣可以在现在的环境下使用上述的这个命令，但是，真实的是，当你用 **SYS(1037)**来编码时 **INSERT BLANK BEFORE** 也是个好主意。**SYS(1037)**仅用于表里的第一个记录。

同样，注意在 **DO WHILE** 循环里检查打印机名。正如其名，**MultipleSetupsOnePrinter** 要求你所安装的作为某个集合一部分的所有设置是对同一个打印机的设置（见图 **4**）。你的多个设置可以有你所喜欢的许多不同，但是如果那不是同一个打印机的指令，它就难以来深度发送给同样的打印作业。

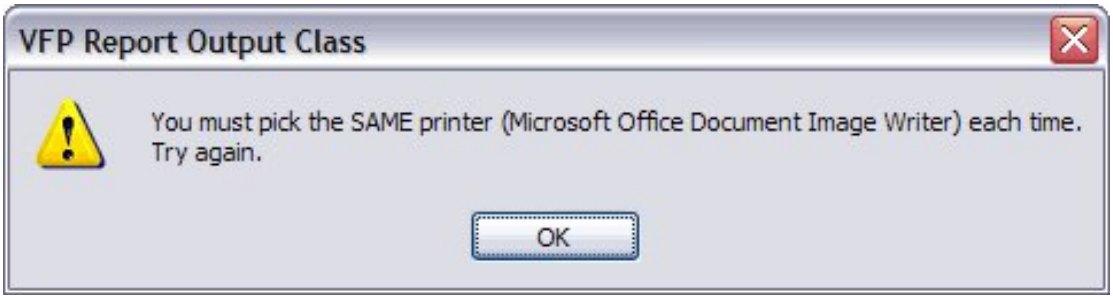


图 **4**：所有的打印作业设置都必须是同样的打印机。

MultipleSetupsOnePrinter 使用一个简单的拆分过程，**GetPrinterSetupDeviceInfo**，每次调用 **SYS(1037)**从 **Expr** 字段读取打印机设置名，因此它可以对照集合里的第一个项目来校验打印机。你可以看到在 **PrintJobListener** 的多个方法里用到这个实用方法。

在 **LoadReport** 代码里，你可以注意到 **MultipleSetupsOnePrinter** 把集合作为一个元素，而不是直接引用 **THIS .FRXPrintSetups**。这个方法是全局的，并非演示使用，你可以在任何报表外调用它，通过一个临时的集合对象或是表单的一个成员属性，如果你喜欢。在应用程序里，你可以提供一个相应的用户接口，让用户知道使用什么，和什么报表，每个要指定的设置。你可以包含关联的报表名或是附加的信息作为 **Name** 字段的一部分，或是为此目的使用其他在 **FRX** 里未用的列。

如果 **MultipleSetupsOnePrinter** 不能独占打开表，或是表是只读的，它就无法在运行时加入设置，但是它仍然可以检查这次运行时你指定的设置是否存在。注意到它并没有再次检查兼容的打印机信息的顺序；管理这个表是你的职责。如果你出了错并指定打印设置不属于同一个打印机，打印作业并不会失败。然而，你或许并不会得到你所期望的结果。

在预先的障碍清除后，**LoadReport** 调用相应的名为 **Setup** 的方法以开始它实际的工

作：

```
PROTECTED PROCEDURE Setup()
    THIS.Declares()
    THIS.OldPrinterName = SET("PRINTER",3)
    THIS.SetFRXDataSession()
    USE (THIS.PrinterSetupTable) IN 0 ;
        ALIAS (THIS.FRXPrintSourceAlias) AGAIN SHARED
    THIS.SetPrinterOptions(THIS.FRXPrintSetups.GetKey(1), .T.)
    SELECT 0
    THIS.ResetDataSession()
    SET PRINTER TO NAME (THIS.CurrentPrinterName)
ENDPROC
```

首先，**Setup** 发出一系列的 **DECLARE-DLL** 语句。你可以发现使用了单独的 **Windows API** 调用；现在我想指出它们的参数并不复杂和深奥。只有其中两个包含有“结构”作为参数，并且你将不会不得不担心其中任何一个。

接着，**Setup** 使用 **SET(“PRINTER”,3)**保存了原始的 **VFP** 打印机名。此时它以共享方式重新打开打印机设置指令表，使用关联于表的别名以作为运行报表的补偿（存储在它的保护属性 **FRXPrintSourceAlias** 里）。然后它调用 **SetPrinterOptions** 方法，传递集合里的第一个打印设置的值。**SetPrinterOptions** 检查打印设置表的相关指令：

```
*&* excerpted SetPrinterOptions
SELECT * FROM (THIS.FRXPrintSourceAlias) WHERE ;
    ALLTR(UPPER(Name)) == ALLTR(UPPER(tcWhichSetup)) ;
    INTO CURSOR (THIS.FRXPrintTargetAlias)
IF USED(THIS.FRXPrintTargetAlias
    IF NOT EOF(THIS.FRXPrintTargetAlias)
        SELECT (THIS.FRXPrintTargetAlias)
        THIS.GetHDC()
        IF THIS.HDC # -1
            IF tlSaveExistingOptions
                THIS.SaveRestorePrinterSetupData(.T.)
            ENDIF
            = ResetDC(THIS.HDC,ALLTRIM(Tag2))
        ENDIF
    ENDIF
    USE IN (THIS.FRXPrintTargetAlias)
ENDIF
```

在上述 **SetPrinterOptions** 的先前版本里，你可以看到对类的 **GetHDC** 方法的调用。这个方法的职责是获得和维护你自己的（非 **VFP** 的）打印作业的句柄。正如你在下面的代码段里见到的，它简单地从你的打印设置指令表的 **Expr** 字段传递以处理相应的信息给 **Windows CreateDC** 函数。与此同时，它提供了重要的信息给其余的 **Setup** 过程，通过

存储在 **CurrentPrinterName** 属性里的值：

```
PROTECTED PROCEDURE GetHDC()
    IF THIS.HDC = -1
        *&* create one
        LOCAL m.lcDevice, m.lcDriver, m.liSelect
        IF USED(THIS.FRXPrintTargetAlias)
            m.liSelect = SELECT()
            SELECT (THIS.FRXPrintTargetAlias)
            m.lcDevice = THIS.GetPrinterSetupDeviceInfo(Expr,"DEVICE")

            IF EMPTY(m.lcDevice)
                m.lcDevice = SET("PRINTER",3)
            ENDIF

            THIS.CurrentPrinterName = m.lcDevice
            m.lcDriver =THIS.GetPrinterSetupDeviceInfo(Expr,"DRIVER")

            IF EMPTY(m.lcDriver)
                m.lcDriver = "winspool"
            ENDIF

            THIS.HDC = CreateDC( m.lcDriver, m.lcDevice, CHR(0), 0 )
            SELECT (m.liSelect)
        ELSE
            THIS.HDC = -1
        ENDIF
    ENDIF
ENDPROC
```

在调用 **GetHDC** 后，句柄就可用了，**SetPrinterOptions** 调用 **Windows** 的 **ResetDC()** 函数。它传递句柄和你的设置指令以代替用户的打印机设置的标准参数选择。

ResetDC 把设置指令作为结构放入表单，但是不用担心。你所需要的结构与 **SYS(1037,1)**的一样完美和复杂，尽管是难以理解的，为你保存于你的打印设置表里的 **Tag2** 字段。你将不得不这么做以传递它。

在这个初始调用里，**Setup** 也传递第二个参数给 **SetPrinterOptions**，指明 **SetPrinterOptions** 应保存它想要使用的打印机的初始状态，在它调用 **ResetDC** 做改变前。**SetPrinterOptions** 为此调用了 **SaveRestorePrinter SetupData**。

SaveRestorePrinter SetupData 使用了 **SYS(1037,2)**来保存，并在报表运行结束时再次调用 **SYS(1037,3)**，以实现恢复。**PrintJobListener** 为此目的有第二个受保护的成员属性，类似于在 **StripAndSaveFRXSetup** 里用过的那个。

在第一次调用 **SetPrinterOptions** 后，并为你本次运行所指定的打印机设置为相应的值，**PrintJobListener** 已经依你想要的方法安排你的报表版面。在最后的动作，**Setup** 同步报表引擎以服从页面大小、方向，以及你的第一个打印设置里等等，就如以下的简单命令：

```
SET PRINTER TO NAME (THIS.CurrentPrinterName)
```

记住，这个代码在 **LoadReport** 里运行，因此报表引擎并不会开始它自己的工作以创建报表页。

在这个设置代码的主体之后，**PrintJobListener** 在报表运行时对定制报表引擎的处理不再做任何其他事，因此我直接移到 **UnloadReport**，你在这里需要发送页面图像到打印机，就在下一节。

但是首先，花点时间来考虑下这个设置代码给你带来了什么？，通过检查选项条“有多少事需要我来保存和恢复？”

如何打印文档

一旦引发 **UnloadReport** 事件，**PrintJobListener** 首先确定是否有需要处理的页，你可能从 **LoadReport** 事件里返回为**.F.**。假定它有需要打印的页，**PrintJobListener** 必须首先告诉打印机以等待文档。它为此使用了 **Windows** 函数 **StartDoc**，它传递了在 **Setup** 方法里保存的句柄。

在通过遍历你的 **FRXPrintSetups** 集合以发送每组的页给打印机后，当然它也告诉打印机文档已经完成，通过使用 **EndDoc** 函数并传递同一个句柄。最后，它利用 **Cleanup** 方法来恢复它在漫长的设置过程中所保存所有内容。

UnloadReport 的代码，是一个用 **StartDoc** 和 **EndDoc** 调用来“包围”打印过程，它是相对简单的，因此我并不包含 **UnloadReport** 代码在文章的正文里。当你对源代码进行研究时，你将注意到 **StartDoc** 是集合里的第二个 **Windows** 调用（除了 **RsetDC** 以外），它可以创建一个结构作为参数，但是我只是简单的传递一个空值字符串。这个参数发送文档给函数所指向的设备，通过处理它的设备内容句柄。

如果你想要幻想，你可以回到前头，并通过增加信息到它的 **lcDociInfo** 参数以“丰富”**StartDoc** 代码。比如，你可以提供一个名以为你的打印作业显示在 **Windows** 打印队列里，利用 **Reportlistener PrintJobName** 属性来获取上述的值。

在我的经验里，对于控制打印作业的真实有效的活动，**VFP** 开发人员难以理解的一个是，利用 **Reportlistener** 的 **OutputPage** 方法发送文档里的单独页给打印机。虽然这个方法的名字表明你发送一个页到目标设备，记住 **OutputPage** 方法从报表引擎的视图里只

发送一个“页”。打印机没有办法知道你所发送的每个图像，其实，只是一页，你不得不告诉它。正如你以 **StartDoc** 和 **EndDoc** 调用来包围打印作业，你将不得不以 **StartPage** 和 **EndPage** 调用来包围每一页。

回想旧 **FoxPro** 的作为字符输出的 **PRINTJOB...ENDPRINTJOB** 结构或许有些帮助。**StartDoc()**和 **EndDoc()**实现对于输出括号命令的类似功能。还记得你在 **PRINTJOB** 里使用的 **ON PAGE** 和 **EJECT PAGE** 命令吗？你不得不处理某些事以让打印机知道产生输出创建一个页的包，现在正是你自己所要做的。实际上 **OutPage** 把报表引擎包装入一个图像是不重要的。

这里是 **PrintJobListener** 使用的代码。**UnloadReport** 调用这个方法来按照每一页分组处理每一个设置：

```
PROTECTED PROCEDURE ProcessPages(tiSetupIndex, tiPageStart, tiPageEnd)
    LOCAL m.liPage, m.l, m.t, m.w, m.h

    THIS.SetPrinterOptions(THIS.FRXPrintSetups.GetKey(m.tiSetupIndex))
    THIS.SizePages(@m.l, @m.t, @m.w, @m.h)

    *&* passing the coords by reference
    FOR m.liPage = m.tiPageStart TO m.tiPageEnd
        = StartPage(THIS.HDC)
        THIS.OutputPage(m.liPage, THIS.HDC, LISTENER_DEVICE_TYPE_HDC, ;
            m.l, m.t, m.w ,m.h)
        = EndPage(THIS.HDC)
    ENDFOR
ENDPROC
```

你在集合里所指定的每个打印设置，**SetPrinterOptions** 对已存在的句柄以不同的指令集调用 **ResetDC**，利用以相应的打印设置表的记录的 **Tag2** 内容。在它调整指定页范围的打印作业后，**ProcessPages** 需要描绘出相应的左、顶、宽度和高度以匹配于 **OutputPages**。在你的集合里每个打印设置都可以有一个不同的页大小和方向，因此这些值在每个页范围里可以是不同的。这是大多数 **VFP** 开发人员的第二个混淆的地方。

部分混乱的发生是因为 **VFP9.0 RTM** 为它的 **DPI** 使用了意料外的缺省值，当输出到这个类型设备时。当你传递一个设备类型 **0**（**hDC** 或 **GDI** 句柄）给报表引擎时，报表引擎使用这个句柄来创建它所需要的 **GDI+**句柄（类型 **1**）。**Windows** 设置了一个缺省 **DPI** 值，在这种情况下它被转换为 **96**，而且报表引擎简单的接受这个值。**Xbase** 代码没有机会来对设备单位和分辨率选择指令。

在 **SP1** 或更后的版本里，报表引擎可以被调整以询问你传递给 **OutputPage** 方法的句柄，并相应于你使用的打印机设置 **DPI**，可以使得对于大多数你想用于作为 **OutputPage**

参数的坐标值更简单的计算代码。我已经在 **PrintJobListener** 的 **SizePages** 方法里包含了两种版本的所需要的计算代码。如果你没有用到 **RTM** 版本，并且不会出现缩放代码以正常工作，当你运行测试代码时，调整 **DEFINE** 常数 **USE_DEFAULT_DPI_GRAPHICS** 值为**.F.**，就是你在 **PRINTJOBLISTENER.PRG** 顶部所见到的。如果你在你的应用程序里实现了这个功能，在测试后你可以 **VERSION()**检查来改变这个赋值。

不幸的是，以正确的单位来运行只是一小部分的混乱，当你要匹配于你所面对的进修。真实的问题是：你如何缩放每一页？

每次运行时报表引擎严密计算页的尺寸，不论你为此次运行决定创建有多少打印设置（每一页的大小、偏移和方向）。**PrintJobListener** 在运行开始时用你自己的第一个指令集来调整报表引擎，在你先前看到的 **Setup** 代码里。在 **SizePage** 方法里做好匹配打印机指令子序列以作为打印作业过程。它使用 **Windows** 函数 **GetDeviceCaps** 来获取当前每页分组的打印设置信息，然后并确定与这些大小匹配的策略。

PrintJobListener 展示一个属性，**ScalePages**，以允许你来调整这个策略。对于每次报表运行，你可以选择比例缩放，比例裁剪，或者比例拉伸页面（见图 **5**）。这些选择类似于那些你在报表版面上使用的图像控制，并且你在报表里已经发现的，每个选择对于图像和页容器组合都是有效的。然而，这三个选择你却只能选择一个。

图 5：你对每个运行报表可以从三个比例选项里作选择。

比如，你可以提出一个选项以把多个报表页图像放在同一个物理页上。在这个假设里，**ProcessPage** 方法将检查 **ScalePages** 属性，当然，因为它并用 **StartDoc** 和 **EndDoc** 调用来对每个单独报表页作包围。

你也可以为表单文字创建一个报表版面，带标题或总结带处理版面封面，其余的页处理其它内容。在这个假设下，你或许想偏移封面左边和顶边的打印设置以匹配特别要求。如果表单文字都是一页长度，你可以以“奇-偶”系统来代替 **FRXPrintSetups** 的页范围集合。你也可以读取自一个表，表内包括有起始页、终止页，和报表运行时的打印设置名与值。在某些情况下，你甚至可以利用 **Reportlistener** 的 **TwoPassProcess** 属性来在运行时创建打印设置集合或页范围的表，在报表预传递时。

PrintJobListener 的 **ScalePages_Assign** 方法为 **ScalePages** 属性给你限定了三个

可能值，以覆盖我已经包含在 SizePages 代码里的 CASE 计算。在方法里的 CASE 语句包含一个 CASE .F.和某些注释，以作为你可能加入选择的占位符。就象通常用在 VFP 里，这种可能并不绝对。

关注更大和更重要的事

为什么就此停止？所有的情况，以及由 PrintJobListener 所介绍的所有打印作业处理技术，实现了对单一报表的多打印设置，但是你可以做的更多，当你使用类似的代码来从多报表里处理报表页时。

你已经知道可以使用 NOPAGEEJECT 来连接多个 VFP 报表在单个报表作业里。利用你在本文里所学习到的，你可以比 NOPAGEEJECT 所允许的更容易连接多个报表。在运行时对每个报表，简单的利用独立的 Reportlistener，设置 ListenerType 为 3。在每个报表运行完成后，以相应的“包围”代码，调用其 OutputPage 方法，而不是在 UnloadReport 事件里，应用你的新打印设置选项。

当你使用这个方法，你将可以为这个任务相应的指定每个 FRX 以不同的版面大小。设想把每个 FRX 版面作为文档的章节，类似于一个 WORD 文档章节。不巧的是，Word 允许你对每一个章节关联版面和打印指令；Word 的“首页-其它页”使得便利性仅仅是这个特性的一个情形。

在 Sedna 里有什么会改变（预计将在 2007 发布的 VFP 增强版）？简短的回答是“没有什么”，在这样情况下 PrintJobListener 的代码还仍然可以运行。或许某些会变得更容易，因为 Sedna 可能提供一个 Xbase 工具来为你管理细节。但是你有所有需要的工具来开始评估你的选项，和实现它工作的策略。

PrintJobListener 家族明细

PrintJobListener 派生于 FFC _REPORTLISTENER.VCX 类库。它加入一个名为 OLEAwareReport Listener 的“瘦”类层，因为我想强调你必须考虑到其工具类用作为 COM 服务组件的可能。

我并不在 PrintJobListener 祖先里包含 FFC UtilityReportListener 类，因为 UtilityReportListener 的功能主要用途是基于文件的输出，而 PrintJobListener 是与目标打印机通信而不是目标文件。

就象 PDFListener 和它的帮助类 VFPWinAPILib，OLEAwareReportListener 展示出一个全局成员，ThrowOLEError，更利于你考虑 COM 服务的错误处理。PrintJobListener 利用_ReportListener 的 DoStatus 和 DoMessage 成员作为用户反

馈帮助，以达到某种程度，以减轻某些可能的问题。

通常，我并不把 **PrintJobListener** 设计成可以处理 **COM** 服务的所有情况。我把它派生于 **OLEAware ReportListener** 以提升你的认识，并使得它更容易扩展以相应的行为，如果你的情形需要它。

你可以在 **PRINTJOBLISTENER.PRG** 里随同 **PrintJobListener** 一起找到 **OLEAware ReportListener**。这些类的共同祖先是通过在 **PRINTJOBLISTENER.PRG** 开头的三条 **DEFINE** 语句来指定的，如下：

```
#DEFINE FFC_HOME HOME() + "FFC\"
#DEFINE LISTENER_SUPERCLASSLIB ;
FFC_HOME + "_reportlistener.vcx"
#DEFINE LISTENER_SUPERCLASS "_ReportListener"
```

调整这些行就可以适应你的安装。

PrintJobListener 加入少数几个全局属性和方法到普通 **FFC** 性能里，就如本文里所描述的。作为标准试验，我加入 **Assign** 方法以强制展开属性为有效值，而不仅仅是由类加入到基本集里的每个全局属性，但是也包含任何派生于全局属性并带有重要意义的属性，以使得它可以运行，并使得属性为只读。我也通过调整 **SupportsListenerType** 方法，以指明 **PrintJobListener** 的特别需要：

```
PROCEDURE ListenerType_Assign(tVal)
*&* readonly, do not allow changes
ENDPROC
PROCEDURE SupportsListenerType(tI)
RETURN VARTYPE(tI) = "N" AND ;
INLIST(tI,LISTENER_TYPE_ALLPGS ) AND ;
(NOT THIS.IsSuccessor)
ENDPROC
```

通过更多的代码，**PrintJobListener** 或许就可以支持 **SupportsListenerType** 为 **0** 和 **2**，当然我也可以更容易的调整上述两个方法，如果我找到了一个好的理由来增加代码。

在 **ListenerType** 值要求之外，你也要注意上面 **SupportsListenerType** 方法代码指出的 **PrintJobListener** 并不能作为继承者来运行。**PrintJobListener** 对于继承者并不是个好选择，因为它与报表引擎紧密通信，以设计在给定的运行报表里选择页的大小，就如本文所述。

如果你选择了，你也可以创建一个低性能的变种 **PrintJobListener** 以作为继承者的功能，或者直接接受页的尺寸以处理它。如果你这样做，我推荐你也调整不同的 **ScalePages** 属性初始值，无论值是否适合于你的目的。调整 **ScalePages_Assign** 方法以使得属性只读，类似于我在这里为 **ListenerType** 所做的。

PrintJobListener 的大多数对于标准事件有意义的改变，是出现在 **LoadReport** 和 **UnloadReport**，它是生成文档的分级区域。然而，**PrintJobListener** 也扩充了 **_ReportListener** 的保护方法 **resetDataSession**。这个父类的方法，被设计成返回的对象为它的“原始会话”而不是运行报表所分配的任何会话。

PrintJobListener 类加入了三个全局方法（**StripAndSaveFRXSetup**、**RestoreSavedFRXSetup** 和 **MultipleSetupsOnePrinter**），这些都利用了 **FRXDataSession**。这三个方法调用了父类的 **SetFRXDataSession** 方法，当报表并不在运行时它并没有执行任何事，以开始它们的工作。**PrintJobListener** 版本的 **ResetDataSession** 方法返回“动作”给 **PrintJobListener** 的 **CommandClauses.StartDataSession**，它将是用户调用 **REPORT FORM** 命令的表单会话。这个会话在 **LoadReport** 和 **UnloadReport** 中是可用的，在 **Reportlistener** 的 **CurrentDataSession** 里是不可访问的。

你可以在运行报表外调用这些展示的方法，当 **FRXDataSession** 不存在和 **CommandClauses.StartDataSession** 信息不可访问时；它们在此时并不改变数据工作期，并保护你的当前会话环境。

有多少事需要我来保存和恢复？

在 **VFP** 里处理你自己的打印作业有许多方面的问题。**PrintJobListener** 过程使得你可以更容易保存三个不同项目的状态：

- VFP 的缺省信息不可被改变。SET(“PRINTER”,3)的内容（VFP 的当前打印机）在你的打印作业前后必须一样，更值得注意的是，SET(“PRINTER”,3)的打印机设置状态所有必须完全一样。
- 你所选择的打印机的缺省指令必须被存储。你的打印作业的打印机可能会与 SET(“PRINTER”,3)不同，并且它可能与 SET(“PRINTER”,2)（Windows 的缺省打印机）相同或者不同。是什么打印机不要紧，你的打印作业不可以改变用户缺省打印机和该打印机的参数，从其它程序的角度来说。
- FRX 它自身不可以被改变。你的报表或标签表单可能包含有打印设置指令，并且 PrintJobListener 必须在运行过程中临时移去这些指令。然而，在运行结束时，原始的指令必须与先前的完全一样。

你也可以决定在 **PrintJobListener** 里为 **DECLAREd DLLs** 保存状态。在 **Declares** 方法里，我使用了一个 **LOCAL** 数组以在每条语句前检查当前申明的 **DLLs**。你可以很容易改变这个代码以使用成员数组，并且在报表运行结束时，通过查询数组来 **CLEAR** 你所加入的 **DLLs**（我并不这样做，因为我并没有看到任何趋势来保留 **DECLARE** 调用）。

但是从另一个角度—时间来说，对于这些被 **PrintJobListener** 所保存的每一个项目：同一个项目有多次改变。通过把我的 **Setup** 和 **Cleanup** 代码放入到 **LoadReport** 和

UnloadReport 事件里，我避免了对堆栈问题进行处理的必要。**REPORT FORM** 语句实际上是个程序，并且它不能嵌套。我确保一个简单、可定义的顺序和我的能力以在它的最开始和最后时调用代码。

如果你扩展了 **PrintJobListene** 技术以覆盖了 **REPORT FORM** 命令的顺序，正如我在本文结尾的建议，你可以在 **LoadReport and UnloadReport** 事件里移去外加的 **Setup** 和 **Cleanup** 代码。一旦你明白了相应的入口和出口点，你可以很容易明白保存和恢复动作会在哪儿发生的。

然而，你也可以使用 **SYS(1037)**和这里所展示的技术，在非模块程序过程里来生成由用户指出的未定义的改变。如果你选择这样做，你必须在用户每次改变时保存状态，无论打印机被用户或程序动作所影响要发生什么。对于这样的堆栈，正确的恢复状态就要求应以保存版本的相反顺序来进行。它也要求应以正确的保存版本（对于正确的打印机，或正确的 **FRX**），因此你不能以简单成员属性给 **PrintJobListener** 使用。利用 **PrintJobListener** 的打印机设置表为模型，你可以加入记录以保存打印机的不同状态和 **FRX** 打印设置信息在单一表里。利用其它非必要的 **FRX** 列以指明堆栈顺序，和每个记录所属堆栈。

Raise 你自己的一个小事件

作者：Doug Hennig

译者：fbilo

从 VFP 8 开始增加的事件绑定，当 VFP 程序员们探索了它的强大能力以后，已经越来越多的被用到了。这个月，Doug Hennig 讲述了他自己近来对这个技术的两种用法。

这些日子来，我在自己的应用程序中越来越多的用到了事件绑定。什么是事件绑定？它有什么好处？在 FoxTalk 2003 年 6 月我的专栏文章“Ahoy!Ahchoring Made Easy.”中，我第一次讨论了事件绑定的一种用法。那篇文章讲述的是用事件绑定来为控件们提供抛锚(anchoring)的技术。当然，既然 VFP9 现在有了自己提供的锚，我就不会再用这个了。不过，我最近又利用了事件绑定以优雅的新办法解决了一对过去我必须处理的头痛问题。

首先，是事件绑定的一个简要介绍。

在 VFP 中的事件绑定

当发生了事情的时候，Windows 触发事件。例如，当用户在一个按钮上单击的时候，这个按钮的 Click 事件触发。该事件被你放在按钮的 Click 方法中的代码所处理。(在这里，我是这样区别事件和方法的：事件是当某些事情发生的时候被系统所触发的，而方法则是当事件触发的时候被自动调用的。）你可以认为 Click 方法是被绑定到 Click 事件上的。不过，这种绑定是在单个对象内部实现的。那么如果一个对象想要接收到发生在另一个对象上的一个事件的通知时怎么办？这就是事件绑定的目的。

VFP 7 增加了用 EVENTHANDLER() 函数在 COM 对象中进行事件绑定的能力（而在更早的版本中，你可以用独立的 VFPCOM 工具来完成事件绑定）。这就让我们可以做不少事情，例如，当在 Outlook 中接收到一封邮件、或者一个文档被 Word 打开的时候，相应的在 VFP 中做一些事情。VFP 8 通过给内置的控件提供事件绑定从而扩展了这种能力。这是通过 BINDEVENT() 函数来实现的。这里是一个例子：

```
bindevent(Object1, 'Click', Object2, 'HandleClick')
```

这行代码设立了在对象 Object1 和 Object2 之间的事件绑定。在这里，当 Object1 的 Click 事件发生的时候，Object2 的 HandleClick 方法会自动被调用。

从代码中调用一个方法——比如 **SomeObject.SomeMethod()**——并不会触发一个事件，因此，任何绑定到 **SomeObject** 对象的 **SomeMethod** 方法的东西都不会接收到通知。如果你想让一个事件被触发、并让事件绑定工作起来，请使用 **RAISEEVENT()** 函数而不是调用 **SomeMethod**。例如：

```
raiseevent(SomeObject, 'SomeMethod')
```

这样一来，**SomeMethod** 被调用了，同时，绑定到 **SomeMethod** 事件的任何对象的方法也被调用了。你甚至可以给这些方法传递参数，例如：

```
raiseevent(SomeObject, 'SomeMethod', MyParameter1, MyParameter2)
```

注意，当你使用 **RAISEEVENT()** 的时候，从事件代码或者绑定到该事件上的任何方法返回的值都将被忽略，**RAISEEVENT()**总是返回 **.T.**。

还有两个与事件绑定相关的函数：**UNBINDEVENTS()**和 **AEVENTS()**。象它们的名字所表示的那样，**UNBINDEVENTS()**解除事件的绑定，而 **AEVENTS()**则以关于事件绑定的信息填充一个数组。

BINDEVENT()、**UNBINDEVENTS()**、和 **AEVENTS()**拥有多个参数用于准确的指定它们应该怎么做。这里我不会讨论关于事件绑定更多的细节，请自行查看 **VFP** 相关的帮助主题。

状态消息

我发现的其中一种事件绑定的用法是不需要对象交叉引用的对象消息。例如，比如说你有一个对象会执行很长的处理过程，最好经常的把这个处理过程当前的状态告诉用户，这样用户就不会以为应用程序已经死掉了。

在 **SAMPLES.VCX** 中的 **ProcessEngine** 是这么一个类的简单例子。它有两个方法 **Process1** 和 **Process2**，这两个方法其实并不做任何事情，它们只是通过在多个“任务”之间停顿一秒来假造一个多步骤的处理过程。这里是 **Process1** 中的代码：

```
* 声明 Windows 的 API 函数 Sleep 以实现停一下的功能
declare Sleep in Win32API integer nMilliseconds

* 执行每一个任务，并更新状态
This.oStatus.ShowStatus('Performing task 1')
Sleep(1000)
This.oStatus.ShowStatus('Performing task 2')
Sleep(1000)
This.oStatus.ShowStatus('Performing task 3')
Sleep(1000)
This.oStatus.ShowStatus('Done')
```

这个类可以被用于各种地方，例如一个表单。这里是从 **PROCESS.SCX** 的命令按钮的 **Click()** 方法中取来的一个例子：

```
loProcess = newobject('ProcessEngine', 'Samples.vcx')
loProcess.oStatus = Thisform
loProcess.Process1()
```

Process1 调用这个表单（通过在 **oStatus** 中对它的对象引用）的 **ShowStatus()** 方法，该方法去更新一个 **ActiveX** 控件 **StatusBar** 来告诉用户当前的状态。

这段代码的问题在于：它假定在 **oStatus** 属性中存储着一个对象引用，而且 **oStatus** 还得有一个 **ShowStatus** 方法，最后，在状态发生变动时只能有一个对象能得到通知。如果程序员忘了把对该表单的引用放到 **oStatus** 属性中去了怎么办？如果调用表单没有一个 **ShowStatus** 方法的话怎么办？如果我们把 **ProcessEngine** 编译进了一个 **COM** 部件、并从一个没有用户界面的 **Web** 服务中来调用它的话怎么办？如果有多个对象需要在状态发生变化时得到通知的话该怎么办？

此外还有一个问题：客户端有对 **ProcessEngine** 对象的一个引用，同时 **ProcessEngine** 对象则有着对客户端的一个引用（或至少有另一个要显示当前状态的对象存在）。这会在释放这些对象的时候出现问题：客户端对象无法被释放（**destory**），因为在 **ProcessEngine** 中有一个对它的未决对象引用**(OutStanding reference)**存在；而 **ProcessEngine** 对象也不能被释放，因为在客户端也有着一个对它的未决引用。当然，这并非是一个不能克服的问题——你需要在释放对象之前先将未决引用设置为 **NULL**——但确实是一个你需要小心的地方。

从 **VFP 8** 开始，我们可以利用事件绑定来减少耦合。这里是在 **ProcessEngine** 的 **Process2** 方法中的代码：

```
raiseevent(This, 'ShowStatus', 'Performing task 1')
Sleep(1000)
raiseevent(This, 'ShowStatus', 'Performing task 2')
Sleep(1000)
raiseevent(This, 'ShowStatus', 'Performing task 3')
Sleep(1000)
raiseevent(This, 'ShowStatus', 'Done')
```

注意，这里没有对任何其它对象的引用，除了对自己的以外。这就是说，该对象不知道、也不关心状态改变的时候有哪个客户端需要被通知。这里对 **oStatus** 是否包含一个有效的对象引用、或者该对象引用是否有一个 **ShowStatus** 方法也没有要求。唯一要求的是：这个类得象调用 **RAISEEVENT()**时指定的那样自己有一个 **ShowStatus** 方法，并且这个方法得接收一个字符型参数。在 **ProcessEngine** 的 **ShowStatus** 方法中，除了包含一个 **LPARAMETERS** 语句以外就没别的东西了，因为也没什么要在这里做的事情。从

RAISEEVENT() 的角度来说，也只不过是要求有这个方法存在而已。

这里是在 Process.scx 中调用 Process2 的命令按钮 Click 方法中的代码：

```
loProcess = newobject('ProcessEngine', 'Samples.vcx')
bindevent(loProcess, 'ShowStatus', Thisform, ;
'ShowStatus')
loProcess.Process2()
unbindevents(loProcess)
```

通过绑定到 Process 类的 ShowStatus 事件，该表单能够在状态改变时得到通知而又不在我们前面看到的那些对象之间建立耦合。如果该表单的方法是调用 UpdateStatus 而不是 ShowStatus，我们也只要改一下 BINDEVENT() 语句就行了，根本不去动 ProcessEngine 类。如果我们想从一个不要求有任何用户界面的对象去使用 ProcessEngine，那么只要根本不用 BINDEVENT()，就不会有任何状态改变的消息发生。如果当消息变动时需要通知多个对象，所要做的也不过是多加几条 BINDEVENT() 语句而已。

这种办法最酷的事情是：实现它并不需要大量的重构：

- 给“Server”类添加一个作为“事件”的方法。不要求有什么代码（除了有参数要被传递时要加一条 LPARAMETERS 语句以外），虽然你可能想要让某些行为在此时发生。
- 在 Server 类的那些方法中，把对其它对象的调用（在这个例子里就是“This.oStatus.ShowStatus”）改成 RAISEEVENT()。
- 在 “Client”对象中，不要在某个属性中存储对 Server 对象的一个对象引用（在这个例子里就是“oStatus”），并使用 BINDEVENT()和 UNBINDEVENTS()来将服务器事件绑定到该对象的方法。

我花了不到一小时就重构了整个应用程序来使用这种形式的消息而不是耦合。结果，我的类们变得比以前灵活得多了。

改变消息

有一天，VFP 领袖 Tammar Granor 和我正在聊“当用户做了某些改动时，怎么让一个表单中的对象们知道”的最佳途径的话题。这通常是为了使能(Enable)在一个数据输入表单中“保存”和“取消”按钮、或者在一个向导表单中“下一步”和“完成”按钮、或在任何类型表单上的相关控件们。

在过去，我用过了多种方法来处理这个问题。

控件调用一个表单方法

在一个表单上的每一个控件的 **InteractiveChange** 和 **ProgrammaticChange** 方法都调用在表单上的一个 **RefreshOnChange** 方法（我自己的象 **SFTextbox** 和 **SFEditBox** 这样的基类，其 **InteractiveChange** 和 **ProgrammaticChange** 方法都会调用该基类自己的一个自定义方法 **AnyChange**，通常我都把代码放在这个方法里面）。然后 **RefreshOnChange** 根据需要使能或者禁止象保存和取消按钮这样的控件。

这种办法有几个问题。首先，我必须把调用放在 **AnyChange** 里面，通常我会在类的层次上这么做以避免在每个控件上手工的输入，而这就意味着建立一整个新的类层次，而且这些类的用途还受限制。其次，每个表单必须清楚有哪些控件是要刷新的，以及控件的层次（例如在页框中的哪个页、或者相关的工具栏）。解决第二个问题有个变通办法，每个控件自己的 **Refresh** 办法可以被用来在用户修改了某些东西的时候使能或禁止该控件，但这就意味着你得刷新整个表单，而这可能对当前拥有焦点的控件造成影响。

会调用一个表单的方法的计时器

在表单上的一个计时器控件的 **Timer** 方法会调用表单的 **IsRecordChanged** 方法，该方法使用 **GETFLDSTATE()** 来判定表单上控件们被绑定到的当前记录中是否有什么改动，如果是的话，则调用表单的 **RefreshOnChanged** 方法，这个方法会采用与前面的机制类似的代码处理通知所有其它控件的事情。

这种办法解决了上一种办法存在的第一个问题，但还是无法避免上一种办法的第二个问题，此外，它还造成了一个新问题：控件们仅当 **Timer** 事件触发的时候被刷新。而且，这种办法只在控件被绑定到一个游标的字段上时才有效，而不能用在控件被绑定到一个对象（例如一个商业对象）的属性上的情况下——除非有什么办法可以判定这些属性已经被改动过了。

只刷新那些需要刷新的对象们

在“我于一九九八年五月的文章《**Build Your Own Wizards**》中提供的那些用于建立基于向导的表单的”类上有一个自定义属性 **lRefreshSteps**。如果该属性为 **.T.**，则当表单上发生了什么改动的时候该控件就应当被刷新。这些控件的 **InteractiveChange** 和 **ProgrammaticChange** 方法调用表单的 **RefreshSteps** 方法，该方法会遍历表单上的所有控件，调用每个 **lRefreshSteps** 属性被设置成了 **.T.** 的控件的 **Refresh** 方法，然后每个控件的 **Refresh** 方法会采取适当的行动。

这种方案消除了第二个问题（表单仅刷新那些被指定为需要刷新的控件），但仍然存在着第一个问题。不过，因为这些基于向导的控件拥有其它的跟向导相关的行为，并且通常总能被正常使用，这还不算是个大问题。

使用事件绑定

Tammar 和我逐渐认识到事件绑定会提供一个对这些问题的最简洁的解决方案。这里就是我实现它的办法。

如我前面所提到的那样，我的那些基类的 **InteractiveChange** 和 **ProgrammaticChange** 方法调用自定义的 **AnyChange** 方法。在新方案中，这两个方法现在换成这么做了：

```
raiseevent(This, 'AnyChange')
```

对对象的角度来看，效果是一样的：都是调用 **AnyChange** 方法而已。然而，区别在于：现在，其它的对象可以通过绑定到这个对象的 **AnyChange** 来实现当这个对象发生改动时得到通知的效果，而这个对象则根本不需要知道关于其它对象的任何情况、需求。

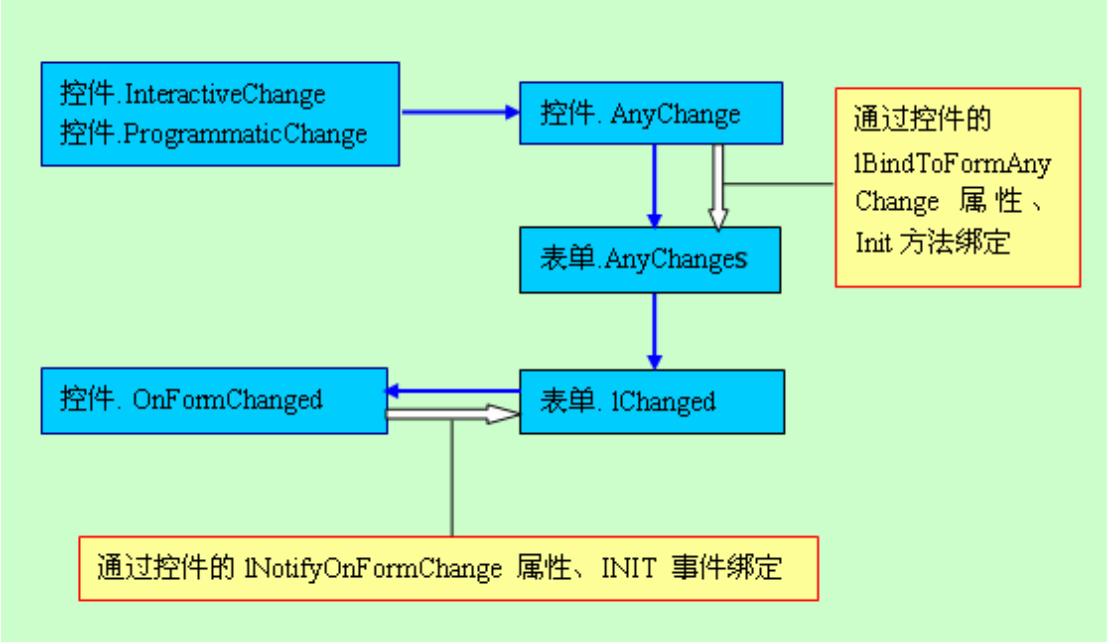
这些类的 **Init** 方法中有这样的代码：

```
if This.lBindToFormAnyChange and ;
    vartype(Thisform) = 'O' and ;
    pemstatus(Thisform, 'AnyChanges', 5)
    bindevent(This, 'AnyChange', Thisform, 'AnyChanges')
endif This.lBindToFormAnyChange ...
```

如果自定义的 **lBindToFormAnyChange** 属性为 **.T.(**默认是 **.F.)**，这段代码会把该对象的 **AnyChange** 方法绑定到表单的 **AnyChanges** 方法（如果表单有这么一个方法存在的话）。我的表单基类 **SFForm** 对象的确有这么一个 **AnyChanges** 方法。这个方法只是简单的把表单的 **lChanged** 属性设置为 **.T.** 来表示有什么事情发生了变动。

所有的控件都有一个自定义的 **lNotifyOnFormChange** 属性，如果为 **.T.(**默认是 **.F.)**，则由 **Init** 方法把控件的 **OnFormChanged** 方法绑定到表单的 **lChanged** 属性（没错，你可以绑定到一个属性，无论何时该属性的值发生了变动，都会对象的绑定事件被触发）。注意指定在 **BINDEVENT()** 中额外的“**1**”，这是用来告诉 **VFP** 应该在 **lChanged** 已经被改动之后而不是之前去调用 **OnFormChanged**。这是重要的，因为控件也许会用 **lChanged** 来做些事情，而我们想要看到的，是新的值而不是旧的那个。

（译者注：我也有点绕晕了，所以画个流程图整理一下：



)

```
if This.lNotifyOnFormChange and ;
    vartype(Thisform) = 'O' and ;
    pemstatus(Thisform, 'lChanged', 5)
    bindevent(Thisform, 'lChanged', This, ;
        'OnFormChange', 1)
endif This.lNotifyOnFormChange ...
```

默认情况下，**OnFormChanged** 调用 **This.Refresh**，不过你可以根据需要在子类或者实例中修改它。

那么，这里就是这个机制如何工作的全部内容了。我给表单添加一个控件。如果表单或者其它控件需要在该控件的值变动时得到通知，我就把该控件的 **lBindToFormAnyChange** 设置为 **.T.**。如果该控件需要在表单上别的什么东西发生变动时得到通知，我就把它的 **lNotifyOnFormChange** 属性设置为 **.T.**，并在它的 **Refresh** 或者 **OnFormChanged** 方法中放入适当的代码。

在示例文件 **CUSTOMERS.SCX** 中，每个文本框的 **lBindToFormAnyChange** 设置成了 **.T.**，因此，在这些控件中的改动将导致表单会被通知到，而 **Save(**保存**)**和 **Cancel(**取消**)**按钮的 **lNotifyOnFormChange** 属性被设置成了 **.T.**，这样它们就会在表单被通知时得到通知。下面是在这些按钮的 **Refresh** 方法中的代码，当用户编辑当前记录的时候会使能它们。

```
This.Enabled = Thisform.lChanged
```

更佳的改动控制

如果用户修改了在一个文本框中的值，然后又把值改回原样了时，这该怎么办？在这样的情况下，我们并不真的希望使能保存和取消按钮。问题在于，任何一个改动都会导致表单的 **lChanged** 属性被设置成 **.T.**，即使改动已经被还原了也是这样。我们真正需要去做的，是检查后台数据中是否有任何改动，然后再相应的设置 **lChanged**。

我的数据录入表单基类 **SFMaintForm** 覆盖了 **AnyChange** 方法，它首先去检查导致方法被触发的控件（可以使用 **AEVENTS()** 函数来取得该控件）是否被绑定到了游标中的一个字段，如果是的话，再检查该字段的值有没有被改动过。接着代码会调用一个自定义的 **UpdateChanges** 方法（这里我们不会讲这个方法的内容）来更新一个“包含有所有已经被改动了的字段的名称的”集合。如果一个字段被改动过了，**UpdateChanges** 会把该字段添加到这个集合中去。如果一个字段又被还原成了它原来的值，**UpdateChanges** 就将该字段从集合中删除掉。如果这个集合是空的，也就是说没有任何改动，那么 **lChanged** 就被设置成 **.F.**。这里是 **AnyChanges** 的代码：

```
local laEvent[1], ;
    loControl, ;
    lcField, ;
    lnPos, ;
    lcAlias, ;
    luValue, ;
    luOldValue, ;
    llChanged

aevents(laEvent, 0)
loControl = laEvent[1]

do case
    * 如果触发 AnyChange 的控件有一个 ControlSource 属性
    * 并且该属性还指向了什么东西
    * 就先把它搞清楚
    case pemstatus(loControl, 'ControlSource', 5) and ;
         not empty(loControl.ControlSource)
         lcField = loControl.ControlSource
         lnPos = at('.', lcField)
         lcAlias = left(lcField, lnPos - 1)
         lcField = substr(lcField, lnPos + 1)

    * 如果该控件有一个 Value 属性，就取出该属性的值
         if pemstatus(loControl, 'Value', 5)
              luValue = loControl.Value
```

```
            * 如果该控件被绑定到游标中的一个字段，
            * 则检查这个值是否与字段中的不同。
            * 我们可以使用 GETFLDSTATE(),
            * 但即使字段被同一个值替换了，这个函数还是会认为字段被改动过了。
            * 因此我们需要把当前值跟 OLDVAL() 比较一下
                    if used(lcAlias) or empty(lcAlias)
                        luOldValue = oldval(lcField, lcAlias)
                        llChanged = (isnull(luOldValue) and ;
                            not empty(luValue)) or ;
                            (not luOldValue == luValue)

            * 我们可以使用象 "This.lChanged =This.lChanged OR llChanged"
            * 这样的代码，但一旦 This.lChanged 为 .T.，
            * 将一个字段还原为它原来的值后，由于这里的 OR，
            * This.lChanged 的值将仍然是 .T.。
            * 所以，我们将以一个被改动了的所有字段的集合来解决。
            * 如果该字段被还原了，我们就把它从集合中删除掉。
            * 如此一来，只有当集合中不为空的时候 This.lChanged 才会为.T.。
            * 这步工作是在 UpdateChanges 中完成的。
                        This.lChanged = This.UpdateChanges(lcAlias ;
                            + '.' + lcField, llChanged)

            * 控件被绑定到了别的什么东西上（不是字段）
            * 所以只能做个标记表示什么东西发生了变动
                    else
                        This.lChanged = .T.
                    endif used(lcAlias) ...
                endif pemstatus(loControl, 'Value', 5)

            * 别的情况也可能导致这个事件被触发，比如一个自定义属性 cControlSource
            * 如果我们不知道要对它作什么特别的处理,就只做一个标记表示什么东西发生了
变动
            otherwise
                dodefault()
        endcase
```

运行 **CUSTOMERS.SCX** 表单，并随便对某些字段做一些改动。注意，从你做出第一个改动开始，保存和取消按钮就变成启用了。接着取消你做过的改动，你会看到这些按钮又变得不可用了。

结论：

事件绑定是给 **VFP** 增添的一个强大的功能，因为它让你可以将不同的对象挂钩到一起而不需要知道彼此之间的情况，从而避免了在没有这种技术情况下会出现的耦合问题。我相信，随着时间的流逝，我会找到它更多的用途。

下载文件： **510HENNIG.ZIP**

在 VFP 9 表单上的 GDI+：操作文本

作者：Craig Boyd

译者：fbilo

这是由 Craig Boyd 所写的关于 GDI+ 及其在 VFP 中应用系列文章的第三篇。跟前两篇一样，这篇文章中大量使用了 VFP9 自带的 FFC 基础类库中的 _gdiplus.vcx，以及来自 GDI+ 平台 API 中的一些其它函数、还有 Craig 为上一篇文章而建立的 gdipdoublebuffer 类（从那时起，该类已经被增强过了）。

在前一篇文章中，我说到我们更关心图像的属性、画笔、纹理、区域、路径等等内容。要完成这些至少还要花费两到三篇文章的内容，因此，在这篇文章中我们将开始处理文本以及一些简单的效果。下一篇文章将探讨高级文本绘制的更多细解。

增强质量并且/或性能

我想要谈述的第一件事情，是你可以如何影响自己正在绘制的图形和文本的质量、以及相反地如何去影响绘制它们的性能。一般来说，质量越高，对性能的影响就越大，但对我将向你演示的某些类型的事情来说，通常尽可能的使用最佳质量却是一个好注意，因为对性能的影响是几乎可以忽略不计的。GDI+ 图形对象有三个基本的设置可以增强或者降低质量和性能。

插值模式（Interpolation Mode）影响 GDI+ 如果缩放和旋转图画。当一幅图画被缩放或者旋转的时候，GDI+ 必须使用多种算法对图像进行重新取样（Resample）来实现希望的效果。可选的八个模式是：Default(默认)、Low Quality(低质量)、High Quality(高质量)、Bilinear(双线性)、Bicubic(双立方)、Nearest Neighbor(译者注：从网上搜索了一下，这个词一般都是不翻译的，中文文章都是直接称呼为 Nearest Neighbor 算法)、High Quality Bilinear (高质量双线性)、Hight Quality Bicubic(高质量双立方)。详细讨论这八种算法中的每一种超出了本文的范围，不过，高质量双立方算法通常是整体质量的最佳选择。

平滑模式（Smoothing Mode），如它的名字所示，会在绘制内容时影响 GDI+ 如何处理粗糙的边缘或者失真。它有五种不同的模式：Default(默认)、High Speed(高速度)、High Quality(高质量)、None(无)和 Anti-Alias(抗锯齿)。这些设置的名称都很好的说明了自己的用途是给你最高的质量还是速度，但我更喜欢使用 Anti-Alias （抗锯齿），它能

够很好的消除图像边缘的锯齿。不过 **Anti-Alias** 有时候会造成一种模糊失真的图像，所以当这样的情况发生的时候我会换用高质量模式。

最后，还有一个 **Text Rendering** （文本绘制）的设置。跟前面两个设置不同的是，**Text Rendering** 只影响文本。它有四种可选的设置：**System Default(**系统默认**)**、**Single Bit Per Pixel Grid Fit(**每像素单比特表格填充**)**、**Anti-Alias Grid Fit(**抗锯齿表格填充**)**、和 **Clear Type Grid Fit(**清晰字体表格填充**)**。象对平滑模式一样，我更喜欢抗锯齿设置。这么做会生成确实非常平滑的字体，虽然部分进行过大量在线阅读的人们会发现抗锯齿文本会让眼睛更累。

对这些以及其它一些 **GDI+** 常量的定义在这篇文章附带的 **vfpgdiplustext.h** 包含文件中。如果你想要一个更权威的列表，可以查看这个 **Blog** 条目：**www.sweetpotatosoftware.com/SPSBlog/PermaLink,guid,2d3a877d-3502-4650-bff3-c06c736dc177.aspx**

我已经为这篇文章专门增强了 **gdipdoublebuffer** 类，新的增强包括包含了三个新的用于脱屏位图的插值模式、平滑模式、文本绘制的属性：**interpolationmode**、**smothingmode**、和 **textrenderinghint**。此外，我为此使用的 **GDI+** 平台 **API** 函数调用放在了 **gdipdoublebuffer** 类的 **paintparent** 方法中。

常见文本效果——示例 #1

Okay，是时候看看这篇文章中的示例以及它们是如何完成的了。从图 **1** 中你可以看到，第一个示例演示了各种可以很轻松的通过使用 **GDI+** 来实现的常见文本效果。

图 1、常见文本效果

阴影效果（Drop shadow）技术

阴影效果也许是所有文本效果中最常用的、也是最简单的一个了。只要简单的把文本绘制作一种（相对于原始颜色）更暗的、补偿性的颜色、然后再以原始颜色再次绘制文本、并放在阴影的上面偏右一点（原文如此，我觉得应该是偏左边吧！）：

```
*!* 先绘制阴影
THIS.renderstring(thisform.text1.value, 18, 38, ;
    "Times New Roman", 22, FontStyleBold, UnitPoint, ;
    RGB(121,121,255), 255, 0) &&更暗的/补偿性的颜色

THIS.renderstring(thisform.text1.value, 15, 35, ;
    "Times New Roman", 22, FontStyleBold, UnitPoint, ;
    RGB(255,255,255), 255, 0) && 正式的文本绘制在上面
```

你会看到我是以坐标（**18，38**）作为阴影的坐标原点的。然后我以一种更浅的颜色在**(15,35)**上绘制同一个文本。另外一种办法是绘制一个模糊且放大了的文本作为阴影。虽然我没有展示一个这么做的示例，但它可以以我在这篇文章中将会讲到的技术来实现。

凸出的浮雕(Raised-Embossed)效果技术

建立凸出的浮雕效果的文本要求你绘制文本三次。

```
THIS.renderstring(thisform.text2.value, 13, 119, ;
    "Arial", 22, FontStyleBold, UnitPoint, ;
    RGB(255,255,255), 255, 0) && 以较浅的/补偿性的颜色绘制

THIS.renderstring(thisform.text2.value, 15, 121, ;
    "Arial", 22, FontStyleBold, UnitPoint, ;
    RGB(0, 0, 0), 255, 0) && 以较暗的/补偿性的颜色绘制

THIS.renderstring(thisform.text2.value, 14, 120, ;
    "Arial", 22, FontStyleBold, UnitPoint, ;
    RGB(121,121,255), 255, 0) && 在中间的文本
```

首先，该文本以一种浅的、补偿性的颜色被绘制；接下来被绘制的同样文本要比第一个向右、向下分别偏离两个像素。最后一个文本则被绘制在前两个文本中间。

雕刻（engraved）效果技术

这个技术非常类似于凸出的浮雕效果，只是阴暗处的文本其顺序和位置与凸出浮雕效果的示例略有不同：

```
THIS.renderstring(thisform.text3.value, 13, 204, ;
    "Arial", 22, FontStyleBold, UnitPoint, ;
    RGB(128,128,128), 255, 0) &&以较暗的/补偿性的颜色绘制

THIS.renderstring(thisform.text3.value, 15, 205, ;
    "Arial", 22, FontStyleBold, UnitPoint, ;
    RGB(255,255,255), 255, 0) && 以较浅的/补偿性的颜色绘制

THIS.renderstring(thisform.text3.value, 14, 206, ;
    "Arial", 22, FontStyleBold, UnitPoint, ;
    RGB(236,236,236), 255, 0) && 在中间的文本
```

这里我首先绘制的是三个文本中最暗的那个、然后较浅的那个、最后是在中间的真正文本。

突出的石块（block）效果技术

突出的石块文本技术包括建立一系列不断增高、偏右的绘制文本，然后把文本绘制为显示在之前的所有文本顶上的工作。我通过使用如下所示的一个简单 **FOR** 循环来实现：

```
*!* 突出的石块文本技术
LOCAL lnStartX, lnStartY

*!* 这里是我们将开始绘制突出石块的 X 和 Y 坐标
lnStartX = 10
lnStartY = 292

*!* 通过在递增的位置绘制 5 个字符串来画突出的石块
FOR lnCounter = 1 TO 5
    THIS.renderstring(thisform.text4.value, ;
        lnStartX, lnStartY, ;
        "Arial", 22, FontStyleBold, UnitPoint, ;
        RGB(121,121,255), 255, 0)
    lnStartX = lnStartX + 1
    lnStartY = lnStartY - 1
ENDFOR

*!* 现在把字符串绘制到石块的顶上
THIS.renderstring(thisform.text4.value, ;
    lnStartX, lnStartY, ;
    "Arial", 22, FontStyleBold, UnitPoint, ;
    RGB(255,255,255), 255, 0)
```

我从由 **lnStartX** 和 **lnStartY** 指定的坐标开始将文本绘制为一种模糊的阴影五次。然后，当 **For** 循环结束的时候，我再次以白色在前述文本的顶部绘制文本。

反射（Reflection）效果技术

你也许已经注意到了，我正在演示的文本效果和技术在变得越来越难。为了建立一个逼真的反射文本，我将会将图形的表面变成顶端向下翻转、并且还要修剪它。

```
LOCAL lcBoundingBox, lnCodePoints, lnLines, lcRectF,;
    logpRect, logpFont, logpStringFormat, lcString, ;
    lnStringLen, lnTextTop, lnTextLeft, lnMatrix

*!* 首先绘制将要被反射的字符串
THIS.renderstring(thisform.text5.value, 15, 375, ;
    "Arial", 22, FontStyleBold, UnitPoint, ;
    RGB(255,255,255), 255, StringFormatFlagsNoWrap)

*!* 测量我们刚刚绘制好的字符串
lcBoundingBox = REPLICATE(CHR(0), 16)
```

```
lnCodePoints = 0
lnLines = 0
logpFont = CREATEOBJECT("gpFont", "Arial", 22, ;
    FontStyleBold, UnitPoint)
logpRect = CREATEOBJECT("gpRectangle", 15, 375, ;
    thisform.Width, thisform.Height)
lcRectF = logpRect.gdipRectf
logpStringFormat = CREATEOBJECT("gpStringFormat", ;
    StringFormatFlagsDirectionRightToLeft ;
    + StringFormatFlagsNoWrap, 0)
lcString = UPPER(ALLTRIM(thisform.text5.value))
lnStringLen = LEN(lcString)
GdipMeasureString(this.gpgraphicsdb.gethandle(), ;
    STRCONV(lcString,5), lnStringLen, ;
    logpFont.gethandle(), @lcRectF, ;
    logpStringFormat.GetHandle(), @lcBoundingBox, ;
    @lnCodePoints, @lnLines)

*!* lcBoundingBox (RECTF structure)
*!* 现在记录下这个字符串的大小
*!*
*!* 返回该字符串的 X 坐标
*!* CTOBIN(SUBSTR(lcBoundingBox,1,4),"N")
*!*
*!* 返回该字符串的 Y 坐标
*!* CTOBIN(SUBSTR(lcBoundingBox,5,4),"N")
*!*
*!* 返回该字符串的 WIDTH
*!* CTOBIN(SUBSTR(lcBoundingBox,9,4),"N")
*!*
*!* 返回该字符串的 HEIGHT
*!* CTOBIN(SUBSTR(lcBoundingBox,13,4),"N")
lnTextTop = CTOBIN(SUBSTR(lcBoundingBox,5,4),"N")
lnTextTop = thisform.Height - (lnTexttop + ;
    CTOBIN(SUBSTR(lcBoundingBox,13,4),"N")) - 16
lnTextLeft = CTOBIN(SUBSTR(lcBoundingBox,9,4),"N") ;
    + 100 - (.70 * (LEN(lcString) - 15))

*!* 在绘制镜像文本之前，先顶端向下翻转位图
this.gpbitmap.ROTATEFLIP(6)

*!* 修剪顶端朝下的位图
*!* 以便当我们恢复它时，镜像文本是倾斜的
lnMatrix = 0
```

```
GdipCreateMatrix2(1, 0, -2, 1, 0, 0, @lnMatrix)
GdipSetWorldTransform(THIS.gpgraphicsdb.gethandle(),lnMatrix)

*!* 在位图上画这个字符串
THIS.renderstring(lcString, lnTextLeft, lnTextTop,;
    "Arial", 22, FontStyleBold, UnitPoint, ;
    RGB(121,121,255), 255, ;
StringFormatFlagsDirectionRightToLeft + ;
    StringFormatFlagsNoWrap) && 先画较暗的阴影

*!* 恢复位图，这样它就不再顶端向下、且是倾斜的了
GdipCreateMatrix2(1, 0, 0, 1, 0, 0, @lnMatrix)
GdipSetWorldTransform(THIS.gpgraphicsdb.gethandle(),lnMatrix)
GdipDeleteMatrix(lnMatrix) && 释放矩阵
this.gpbitmap.ROTATEFLIP(6)&& 把它翻转回去
```

这里有一小部分代码我相信需要更多的解释。首先是 **GDI+** 平台 **API** 函数 **GdipMeasureString** 的使用。为了在正确的位置绘制镜像文本阴影，我需要知道垂直文本的长度和高度。**Visual FoxPro** 的 **FontMetric** 和 **TxtWidth** 函数会给出一个接近该文本宽度的值，但在这个示例里我需要某些更精确的东西。

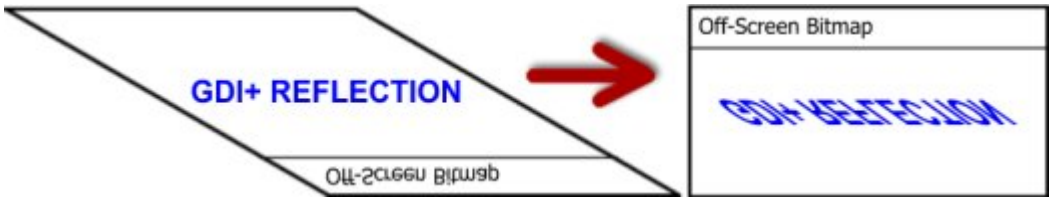


图 **2**、通过顶端向下翻转图像的表面，并在绘制翻身文本之前修剪它，那么，当我把它返回回垂直、并未经修剪状态的时候，就能得到期望的文本效果了。

GdipMeasureString 返回的东西中，除了别的以外，有一个被称为“范围框（**bounding box**）” 的 **16** 位 **RECTF** 结构。这个结构包含了字符串的左边、右边、宽度和高度属性的信息。你只需要知道如何从这个结构中取出这些信息就是了。**Visual FoxPro** 的 **CTOBIN()** 函数做得非常好，尽管在这个示例里面我只需要其中的宽度和高度，我还是附上了注释过的取得所有四个值的代码。

注意我在这里使用了 **Visual FoxPro** 的 **StrConv()**函数。**GDI+** 主要对 **Unicode** 字符串工作，而由于 **Visual FoxPro** 的字符串是由单字节的字符组成的，所以我使用 **StrConv()** 来把它们转换成 **Unicode**。

上述代码中最困难的部分是如何绘制反射了的阴影。就是从我 **RotateFlip()** 位图开始、到我再 **RotateFlip()** 回来之间的那部分代码。我需要让反射成为一个镜像图像，并且我还需要它给出一些指示，说明浅色的源图像来自于上面、左手边。因此，我还需要该

文本有一些轻微的歪斜。我使用了 **gpBitmap** 对象的 **RotateFlip()** 函数来获得这幅镜像图像、一个 **matrix**（矩阵）对象来实现修剪了的效果。这有点难以解释，所以请看一下图 **2**，它演示了我让 **GDI+** 在内部所做的东西。

结论：

这就是示例 **#1** 的全部内容了。在表单上我还提供了一些文本框，所以你可以输入其它的字符串来看看出现的效果。这些文本框还服务于“演示使用 **GDI+**来动态建立文本效果而不只是预先画在图形中”的目的。只要以不同的颜色在不同的位置上绘制文本多次，你就可以建立一些相当大方的文本效果，但当你下个月再看到这个系列的下一篇文章时，你会发现这不过是你所能实现效果的开始而已。

下载：**510BOYD.ZIP**

别再打搅我！

作者：Andy Kramek & Marcia Akins

译者：fbilo

任何应用程序的测试阶段都是棘手的。而让测试员和最终用户精确的报告出了什么错（任何比一个“Hello World”复杂一点的应用程序都会出错）更是棘手。这个月，Andy Kramek 和 Marcia Akins 讨论了如何最好的帮助他们的测试人员们通过以一种使得问题的识别、指定优先级、结束变得尽可能无痛的方式帮助开发团队。

安迪：我要杀了我的 **Beta** 版测试员。我只是发布了用于测试的一个新的模块的第一版而已，却从测试员那里得到了一大堆毫无意义的 **e-mail** 回信。

玛西亚：为什么它们毫无意义？

安迪：这是今天早上我收到的一封邮件：“客户联系人的显示屏破了”。

玛西亚：这还不是什么可怕的帮助，不是吗？我还真碰到过对我也这么做过的客户，当我问他什么意思的时候，他回答：“它就是不工作了”。那真是毫无意义。

安迪：我能够理解，但我该怎么应付这样的情况？我觉得可能我必须实现某种正确的 **bug** 跟踪系统，而不是仅仅依赖于 **e-mail**。

玛西亚：你说的“依赖于 **email**”是什么意思？

安迪：哦，我的错误处理器会自动发一封 **email** 给我，它带有精确的系统状态（调用堆栈、内存堆栈、代码的行位置、甚至所有打开表的状态）。那么我就不需要依赖于用户告诉我出了什么错误了。

玛西亚：这种办法用来对付系统崩溃是挺好的，但如果问题是功能不正确、用户抱有抵触情绪的行为、一个界面上的错误、或者一个增强的需求的话，它就没多大用处了。你是否收到了来自那个“告诉你关于联系人屏幕的事情”的客户的 **Email**？

安迪：不，要不是你现在问起这件事情，我还没意识到我没收到过邮件呢！

玛西亚：这就是说，其实问题并非是代码中的一个 **bug**。你需要做两件事情：首先，训练你的测试员怎样正确的报告 **bug**；其次，实现某种形式的 **bug** 跟踪系统以便 **bug** 报告不会丢失。

安迪：我觉得你是对的。那么第一个问题，就是由哪些东西组成一个“正确的” **bug** 报告。

玛西亚：我们需要问题发生所在地的表单、报表、或者函数的名称，以便我们知道到哪里去找到问题。

安迪：软件版本呢？毕竟，如果某人正在使用“**Build 17**”，而当前编译版本却已经是 **18** 了，那么问题可能已经被解决了。

玛西亚：可为什么某些人还要用一个旧版本呢？

安迪：嗯，假设我发布了一个新的编译版本而这个版本却造成了一个全新的问题使得该版本完全没法用。那么现在只好让用户们回头再用老的那个版本，不过在老版本中的 **bug** 也可能已经被修正了。

玛西亚：**Okay**，那么我们就加上发生错误的编译或者版本号，以保证不浪费时间去捕捉一个已经被修正了的 **bug**。下一件绝对必要的事情，是对问题的详细说明。简单的“它出错了”不能算是一个详细说明。

安迪：我不赞同这么做。为什么不用一张屏幕快照呢？那样不是更有效吗？

玛西亚：不完全是，因为并非所有的问题都能从屏幕快照上看出究竟来。

安迪：也许吧，但至少它可以让我清楚当错误发生时用户所看到的东西，并且避免了让用户必须写下来象这样的东西：“当你在那个通过选择“新建客户”打开的表单上单击“新建”按钮以后，弹出了一个小的红色屏幕**...**”。

玛西亚：当 **beta** 版测试的时候，我用过的一个技巧是：给每个表单的标题栏添加一个表示该表单的运行路径的屏幕 **ID**。比如数字“**123**”表示该表单是由从第一个菜单面板的第二个菜单项下的第三个子菜单项调用的。

安迪：那么，那些从别的地方调用出来的弹出表单该怎么办？

玛西亚：简单。它们就使用来自调用它们的父表单的 **ID**、再加上一个“**P.**”。但是，照我说的话，这些是你并不准备放到最终产品的代码中去的东西，它只是在 **beta** 版测试的过程中有用。

安迪：可你在说的是：有屏幕快照是不错，但并不真正解决问题。

玛西亚：没错。总之，一个针对用户希望出现结果的详细说明才是本质的东西。毕竟，如果程序被观察到的行为是不正确的功能、或者用户不满意的表现，除非我们知道“应该发生什么”，否则就不可能知道“该怎么去修正它”。

安迪：我不是很明白你说的“用户不满意的表现”的意思，而这是你第二次用到这个术语了。

玛西亚：我能给你最好的例子，是多年前在 **FoxPro** 中困扰我的一个东西。想像一下当你正在一行一行的跟踪调试某些代码的情况。你找到了问题，并按下了 **debug** 菜单上的 **fix** 菜单项。**VFP** 首先会问你是否想要“取消并从内存中清除所有对象”。为什么要用这种事情来麻烦我？我刚刚才明确的选择了 **Fix** 菜单项——难道 **VFP** 认为我是个傻瓜吗？

安迪：嗯，我猜它是个“安全”开关。但默认的按钮是 **Yes**，所以你需要做的只是按下回车键罢了。

玛西亚：可这样的结果更糟！一旦代码被打开了，并且你开始编辑，**VFP** 会弹出另一个菜单问你是否想要从内存中释放所有的类，而且默认的按钮是 **No!**到底什么让 **VFP** 认为我会在一个编辑窗口中打开源代码却又不想要能够编辑它？

安迪：我也同意这个设定绝对是令程序员不满意的。那么，你在谈的是“在我的应用程序中与此类似的东西是‘用户不满意的’”吗？

玛西亚：**Yep!**

安迪：那么，现在我们有了关于错误发生在哪里、发生了什么、以及本来应该发生的信息。还有别的吗？

玛西亚：当然有啦。至少还必须要知道“**What and where is how**”。

安迪：你说什么？麻烦再说遍？

玛西亚：我们需要能够再现问题的准确步骤。如果不能再现问题，就没法修正它。

安迪：这么做的问题在于：当错误发生的时候，大多数用户都并不准确的知道自己正在做什么，我也有点同情他们。推测起来，当他们正在系统上工作的时候并不知道会发生一个错误，于是也就没有注意导致问题发生的准确情况。

玛西亚：是啊，可这是个关键性的问题。如果造成触发了错误处理器的是代码或者数据中的一个错误，你总能知道发生了什么事情。麻烦的是那些不会触发错误的问题。用户所应理解的，是重现问题的重要性——在这里，指的就是把这些步骤记录下来。

安迪：听起来你在说这是一个训练问题。

玛西亚：是的。毕竟，如果他们不能再现问题，那又能指望我们做什么呢？

安迪：可如果他们真的就是没法再现问题的话怎么办？难道那就意味着这个问题根本不存在吗？当用户报告了某些事情的时候，如果我告诉他们说我不相信他们说的，我想我能理解用户会感到一点被羞辱的感觉。

玛西亚：嗯，你并没把他们叫做骗子。但这确实有助于我们从用户那里得到想要的东西—

—问题应该属于哪一种类。

安迪：我上钩了(bite，表示上钩、咬住等等。这里应指安迪同意了玛西亚的说法)。你说的“种类”是什么意思？

玛西亚：我们需要能够对收到的报告中的问题进行分类、并区分优先顺序，是吧？这就有赖于每个问题所属的类别。例如，崩溃 **bug...**

安迪：你是怎么定义“崩溃 **bug**”的？

玛西亚：崩溃 **Bug** 是一个或者导致 **C0000006** 错误并导致 **VFP** 自行崩溃的、或弹出一个 **VFP** 的“**Cancle**、**Suspend, or ignore(**取消、挂起、或者忽略**)**”对话框的、或导致你的全局错误处理器介入并关闭应用程序的 **bug**。

安迪：我痛恨这个词，我想我肯定不希望我的用户们谈论“崩溃 **bug**”的话题，不过我明白你的意思。

玛西亚：象我刚才说的那样，崩溃 **bug** 显然应该比象标签中的拼写错误之类的“化妆”问题拥有更高的优先权。

安迪：问题在于：这些种类是什么？这些是用户应该看到的唯一内容，并且我们需要让它们尽量保持简单而明确。

玛西亚：这里是我做的按优先级从高到低的分类：

1. 崩溃 **bug**；
2. 不正确的功能；
3. 用户不友好（令用户不满）的特性；
4. “化妆”问题；
5. 间歇性出现的问题（无法再现的）；
6. 增强的需求；

安迪：这就是你准备让我向我的用户们展示的东西？

玛西亚，当然了，为什么不？

安迪：老天爷，你怎么能希望他们真的理解这些术语真正的含义呢（甚至我自己都没弄明白“用户不友好的特性”）？也许这些对开发人员们来说没问题，但我肯定不会为最终用户使用它们。我参考过大量的商业 **Bug** 跟踪软件系统，发现它们大多数都只向用户提供一个非常简短的列表——其中部分系统甚至只采用两个类别：“**Bug**”和“功能”。

玛西亚：我明白你的意思。我列出来的其实是程序员将对那些被报告上来的东西进行的分类。我明白用户有时当作一个用户不友好特性报上来的东西其实应该是对一个新功

能的需求。

安迪：正确！我建议限制用户只能从“问题”或者“增强”中选择一个类别。这样的话就不会混淆了。现在只剩下问题的优先级要处理了。

玛西亚：嗯？你这么说什么意思？我给你的单子已经是按优先级排序的了。

安迪：我是更多的从修改程序的时间线角度来考虑的：

1. 要求立即编译；
2. 下一次计划中的编译；
3. 下一次修改；
4. 功能编译；
5. 如果时间允许；

玛西亚：那么第一条就是我称作崩溃 **Bug** 的东西喽？

安迪：是的，但它也可能是“不正确的功能”，这要看它是怎么不正确了。这里的关键是：它严重的足以让我们停下手头所有计划好的工作去修正这个问题、并发布一个新的编译版本。

玛西亚：我明白，但第二条和第三条之间有什么区别呢？

安迪：“下一次修改”表示在代码的某些部分有一个错误，但并不紧迫的需要我们立刻就去关心它。不过，我们需要将它指定为：如果某人处于某些原因将要再次使用那块代码的时候，这个问题就应该先被修正好。加上在哪里、什么、如何、以及预期将会发生什么事情，现在我们就有了用来判定分类和优先级的依据。还有别的什么吗？

玛西亚：是的，我们需要记录为解决问题已经做了哪些工作，并且我们需要一种途径来结束该问题。

安迪：这容易。当我修正了问题以后，我简单的把我所做过的工作做个记录，然后在问题上做个“已结束（**Closed**）”的记号。

玛西亚：不，决不！**Bug** 跟踪的黄金准则是：只有那个最早发现问题的人才能决定该问题是否已结束。

安迪：我还不能确定是否同意这种观点。如果用户报告了一个问题，而我通过跟踪代码发现其中特定的某行中有一个错误（例如在某些公式中把 **+** 号写成了**-** 号），毫无疑问这就是问题的原因所在，所以毫无疑问该问题现在就被修正了。那么，为什么我不能结束这个问题呢？

玛西亚：因为是我说的。原因当然是，即使你已经发现了特定的某行中有一个错误，也并不能保证这就是用户看到的问题的原因。就以你所说的例子而言，你怎么知道把加

号改成减号就会导致用户看到的结果？只有用户才能决定这一点，所以说，只有始发者才能最终认可一个问题的结束。

安迪：我想你是对的。我剩下的唯一问题是：我是否应该自己写一个错误跟踪系统呢、还是出门去买一个。

玛西亚：象以前一样，对这个问题的回答是“要看情况”。如果你找到了一个你喜欢的跟踪系统，并且买得起它，那么为什么要“重新发明轮子”呢？问题是：现在市场上有数百个这样的系统，而它们的价格从完全免费到数百美元每用户不等。但说到底，它们都只是简单的数据库（而我们拥有所有关于数据库方面的知识）和一个用户界面而已。大多数系统包括一个 **Web** 界面，而这在处理远程客户端的时候使得事情变得跟简单了。

安迪：**Wow**，市场上肯定多得是。在 **www.sqatester.com/bugtracking** 甚至有一个相当全面的应用程序名单可供挑选，可我就是不清楚它们之间相比都有什么优缺点。我想我会去试用一些演示版来看看是否能找到我想要的东西。

玛西亚：而且如果你找不到，你总是可以写一个自己的东西！