PINNACLE
FoxTalk 2.0
Solutions for Microsoft® Visual FoxPro® Developers

2005 年第 6 期

本月，**Doug Hennig** 将讨论一个你所期望的控制锚定的简单方法，以及如何控制报表预览窗口的外观和行为。

来自 VFP 团队的贴士

page.30

作者：**The Microsoft Visual FoxPro Team**　译者：**CY**

本月“来自 **VFP** 团队的贴士”栏目，将展示如何利用 **UpdateListener** 类在报表生成的过程中为用户提供反馈，将展示报表预览，与此同时将利用后继机制在报表运行的同时创建一个 **XML** 输出文件。

步步为营

page.33

作者：**Andy Kramek & Marcia Akins**　　译者：**fbilo**

有一个我们经常会面对的问题，就是保证一系列连续的处理步骤都被顺利的完成。应付这个问题有多种办法，可它们各自又都存在着一些缺点。在本月的专栏中，**Andy Kramek** 和 **Marcia Akins** 试图找出处理这个问题最简单、并且最有效的办法。

PDF 的力量

作者：Lisa Slater Nicholls

译者：fbilo

PDF 文档输出是 Visual FoxPro 最常被要求的增强之一。在这篇文章中，Lisa Slater Nicholls 为你演示了怎样在所有版本的 FoxPro 中生成 PDF，并展示了一些 VFP 9 中的技术和打印相关的功能，它们可以使得现在做 PDF 比以前任何版本都要轻松。她还讨论了“应用程序透明度”的重要性，并提供了可靠的动态安装 GhostScript 的代码，后者是一个免费的基础部件程序。

当你的输出要求包括交付打印文档的精确电子版本的时候，**PDF(Protable Document Format，轻便文档格式)，PDF** 就是今天的“金本位”。只有当接受者手里有微软 **OFFICE** 文档镜像阅读器或者另一个可以阅读 **TIFF** 和 **MDI** 文件的程序时，基于 **TIFF (Tagged Image File Format，可标记图形文件格式)的 Microsoft Document Imaging(MDI)格式才是一个可以接受的替代品。阅读 PDF 的 Adobe Reader** 时免费的，它已经被广泛的使用，并且可以用在各种平台上。

VFP 9.0 的 **ReportListeners** 可以直接生成多页的 **TIFF** 文档、以及其它的镜像页文件格式，但你的用户还是可能会问为什么你的 **FoxPro** 应用程序不能象其它的软件工具那样生成 **PDF** 文档。事实上，只要象这篇文章中我为你展示的那样，利用其它的软件工具所用的同样的资源，**FoxPro** 应用程序也能够生成 **PDF**。

Visual FoxPro 或者其它微软产品交付时是不带这些资源的。它们被忽视是因为它们对“自由软件运动”和开放源代码部件的依赖为我们指出了一种潜在的可能性。**PDF** 是由另一个重要销售商开发的一个事实标准这个事实，使人想到了另一种可能性。

你的用户可不会关心销售商策略的事情，所以我们对这类问题的思考是没用的。从用户的角度来看，你就是销售商。所以那么你还等什么呢？

你想要什么？

用户就是想要 **PDF**。而你作为一个专业的开发人员，还有一些其它的要求：

- 给你的应用程序增加这个功能最好不会增加费用；

- 要能用在任何 FoxPro 报表上；
- 用户不应该知道关于外部程序的事情，并且在过程中——即使是在安装过程中——不需要用户干预。我把这些特征称为“应用程序透明性”。

应用程序的透明性是一种特殊类型的恩惠，这不是笑话。它也许可以被定义为一系列的特征，让你的基于部件的软件解决方案给用户的感觉就好像它是你彻底专门为他们开发的一样。当你建立一个 **FoxPro** 应用程序的时候，你就用无数的途径给那些部件增加了价值，但某些时候，应用程序透明性的存在会令用户在接受或者反对你的增加价值的提议时产生很大的区别。

我住在拉斯维加斯，所以我打赌你能够实现你的需求，并且我还会提高赌注。只要很小的改动，同样的技术应该还能用在任何服从 **SET PRINTER TO** 指令的输出中，而不仅仅是 **FRX** 或者 **LBX** 。它们应该还能用在任何版本的 **FoxPro** 中，甚至是 **FoxPro for Dos** 和 **UNIX** 这样的基于特征的报表或者其它打印机输出中。

考虑你的选择

使用低级或者高级技术，你可以生成 **PDF**，就像你能解决大多数编程问题一样。

在这篇文章中，我说的“低级”的意思是：你可以理解并掌握 **PDF** 格式，并通过使用这个格式、一次一个对象的解释成报表输出。在 **VFP 9.0** 中，如果需要的话，你就可以使用这种办法。简单的在报表运行时使用一个 **ReportListener** 驱动的类来拦截绘制（**rendering**）事件、并为每个对象生成相应的 **PDF** 代码。

如果你已经是一个 **PDF** 格式方面的专家了，你也许可以用这种办法作出一个合理的原报表复件来，但你将需要根据每一种类型的报表绘制对象的不同而去解决大量的问题。当你见到用于生成 **PDF** 的某个产品或者工具受限于特定类型的输出——例如图像或者形状（**shape**）——或者只能生成其来源的一部分时，该产品或者工具所用的就是“低级”的技术。

我说的“高级”指的是你可以用某些很容易得到东西——一个 **PostScript** 文件和一个 **PostScript** 解释器——来制作出一个 **PDF** 文档。**PostScript** 是在 **PDF** 幕后使用的页面描述语言。一个 **PostScript** 解释器可以表示该文档，将所有重要的引用都打包起来塞到一个压缩了的 **PDF** 文件中、根据需要随意的嵌入字体。

有足够能力和精确的 **PostScript** 打印机驱动程序可以轻松的生成 **PostScript** 文档文件。这些驱动程序被发布在每个版本的 **Windows** 中。我使用了免费的 **GhostScript** 软件部件来作为需求的解释器、并提供到 **PDF** 的转换。

GhostScript 时一个稳定、支持良好、并且成熟的产品。它的伸缩性好的足以被用在

许多提供 PDF 转换服务的网站上，并能够动态生成其它 PDF 输出。当你看到一个能够

为什么不使用 XSL:FO?

除了常用于生成 PDF 的低级和高级技术外，在 VFP 9.0 中你还有一个用于报表和标签的其它选择：采用 VFP XML 输出——它可以完整的描述输出——并使用一个 XSLT 转换来将 XML 映射到 XSL:FO，就像是 HTML Listener 使用一个 XSLT 转换器来将 XML 映射到 HTML 一样。XSL:FO 是一个提供了一种更详细的页描述语言的 XML 方言。从 VFP 的 XML 报表转换成的 XSL:FO 大致是这个样子的：

```
<xsl:template match="/Reports/VFP-Report">
<fo:root>
<fo:page-sequence
 master-reference="default-page" format="1">
 <xsl:attribute name="fo:initial-page-number">
      <xsl:value-of select="./Data/PH[1]/@id"/>
 </xsl:attribute>
 <xsl:for-each select="./Data/PH">
 <fo:flow flow-name="xsl-region-body">
      <fo:block break-after="page" top="0in"
      width="100%">
      <!-- contents of a page described here -->
      </fo:block>
 </fo:flow>
 </xsl:for-each>
</fo:page-sequence>
</fo:root>

</xsl:template>
```

如果你通过任何能够识别 XSL:FO 的处理器来运行这个转换的结果，它将会把 PDF 作为一种基本的输出类型来生成。

如果微软的 XML 部件能够包含一个对 XSL:FO 非常了解的处理器的话，XSL:FO 会是一个理想的办法。但既然他们还没做到这一点，那么你还是需要一个象 Apache XML FOP 项目（http://xml.apache.org/fop）这样的外部部件。

每一个可用的 XSL:FO 处理器都有各自的限制、各自的安装和运行时处理负担、以及各自的 Visual FoxPro 程序员学习曲线。我已经发现没有一个 XSL:FO 处理器能比 GhostScript 的最小需求负担更少。因此，尽管我很赞成使用 XSLT、尽管我赞成使用 VFP 报表的 XML 格式作为替代品，（到目前为止）我还是必须评估这个选项在通向应用程序透明性的道路上作为一种迂回的办法的价值。

提供完整的 PDF 功能和输出的商业工具的时候，它的后台使用的技术非常有可能就是 GhostScript。

我想你可以看到我是从哪里起步的。不管你多么的欣赏在代码的内部去弄脏自己手的

感觉，我还是推荐使用高级的程序：利用 **PostScript** 驱动程序去获得高质量的 **PDF** 输出。

如果你以前试验过这种技术，但是却没有成功，那么原因可能是你无法解决某些应用程序透明性的问题。这篇文章将告诉你为什么，并提供了封装在一个 **ReportListener** 驱动的类中的相关代码以作为参考。如果你偶然有什么其它的 **FoxPro** 方面的需求，把额外的代码放在 **PDFListener** 类中应该也能解决问题。如果你对“怎样在一个应用程序中包含开放源代码或者自由软件部件”这件事情还没有把握，请记住不同的部件提供者设立了不同的方案，但请把现在这个看作是一个完全可以工作的示例。

怎样得到你需要的东西

下面的，是用“高级”的办法来生成 **PDF** 所需的步骤。为了方便起见，尽管没有一个步骤的运行要求用到 **ReportListener**，我还是把这些步骤都放在由 **ReportListener** 派生的类定义里面了，你可以在 **PDFLISTENER.PRG** 文件中找到它们：

1. 判定环境中是否有可用的 **GhostScript**。如果重要的话，就安静的安装它。
2. 判定你的特殊打印机设置是否可用。如果重要的话，用不需要用户干预的方式安装一种适用于你的应用程序的驱动程序。
3. 指示 **VFP** 去使用这个驱动程序。
4. 为你的报表生成 **PostScript** 文件，并记录下产生的文件名。
5. 为了维持良好的状态，恢复 **VFP** 的打印环境。
6. 建立一个命令行文件，列出需要被转换的 **PS** 文件。
7. 带着这个命令行文件、需要的输出文件名、以及其它适当的命令行参数调用 **GhostScript**。

步骤 **1** 和 **2** 可以在任何时间去做，而且通常应该被当作是你应用程序安装过程的一部分去完成。**PDFListener** 以下面的办法把它们当作是一个报表运行时的启动程序的一部分来执行：

```
PROCEDURE LoadPrinterInfo()
    IF NOT THIS.VerifyGSLibrary()
        RETURN .F.
    ENDIF

    IF NOT THIS.VerifyPrinterSetup()
        RETURN .F.
```

```
        ENDIF

        IF NOT THIS.AdjustVFPPrinterSetups()
            RETURN .F.
        ENDIF
    ENDPROC
```

直接设置

在这个方法中开头的两个调用也都是 **Public** 方法，你可以在任何安装程序中独立的调用它们，以确保可以使用 **GhostScript** 和 **PostScript** 驱动程序。请保证你的安装应用程序拥有足够的权限去建立一个目录并安装一个打印机。

你可以在每一个报表处理程序运行的开始时去调用这些方法，就像 **PDFListener** 所作的那样，因为在极少数情况下用户可能卸载了这些外部部件中的某一个。如果当前用户拥有足够的权限，那么丢失的部件将会被重新安装。如果用户没有足够的权限、而又缺少一个部件，那么 **PDFListener.RunReports()** 方法将返回 **.F.**。你的应用程序应该以这样的方式向用户提供容易理解的错误反馈：就好像一个可编辑的 **FRX** 或者一个表被删除了一样。

某些朋友可能会喜欢提供代码以另一个用户的身份去解决这个问题而不是在应用程序中提供错误反馈。我认为这有点过头了，只因为这个原因就把管理员的登录凭证储存在一个应用程序中显然有着潜在的安全隐患。

动态安装 GhostScript

VerifyGSLibrary 是负责检查 **GhostScript** 部件的方法，它使用该类（**GSLocation**）一个公开属性去判定你的目标位置。如果你没有明确的设置过该属性，这个类就使用一个名为 **GetDefaultGSLocation** 的方法来取得一个位置。这个方法就像它的名字那样会在当前运行中的应用程序目录下去找一个合适名称的子目录、或者如果你还没有编译成应用程序的话就从类库所在的目录下找。其目标就是去判定并确保有一个设定的目录可供使用，不管该目录中当前是否包含有 **GhostScript** 的文件。

一旦它已经决定了计划的位置，并且如果还没有可用的文件，如果需要的话，**VerifyGSLibrary** 会建立该子目录。它会使用简单的简直荒谬、而且古老的 **FoxPro** 方法：从备注字段中拷贝出文件，然后放到磁盘上去。稍后我会解释怎样正确的把文件放到 **INSTALL.DBF** 表中去的。

你不需要使用任何注册表调用，并且你不需要运行任何外部程序来安装 **GhostScript**。如果你的应用程序知道文件的位置，你就全部设置好了。你也许会有别的考虑，如果你已

经使用了一个典型的 **Windows** 平台下的 **GhostScript** 发行版的话该怎么办，但是 **GhostScript** 从未作为一个 **Windows** 程序“诞生”过。不管是它的安装还是使用都需要特别的 **Windows** 技巧。

不管怎样，恰当的使用 **GhostScript** 依赖于履行你对它的法律协议的责任。尽管它们不是有偿的，你应该花点时间去熟悉这些协议和各种选择。（参见补充资料“使用自由软件的责任”）

指定你的驱动程序

PDFListener.VerifyPrinterSetup() 是负责检查你所需要的打印机设置和根据需要安装的方法。这个类使用一个公共成员属性 **PSDriverSetupName** 作为你想让用户在安装了的打印机列表中看到的名字。当你分配这个值的时候，它会使用 **APRINTERS()** 函数的一个新的参数来获得驱动程序名称、对你的值和它希望使用的打印机驱动程序进行检查。你可以象许多 **PDF** 工具所做的那样使用一个类似于“ **ABC** 公司 超级驱动”那样的名称，把这个设置用到你的安装程序中，而且不需要有什么对 **PostScript** 明显的关联：

```
PROCEDURE PSDriverSetupName_Assign(tcVal)
IF VARTYPE(tcVal) = "C" AND NOT EMPTY(tcVal)
    LOCAL laSetups[1], liIndex, ;
        llFoundAndNotAppropriate
    FOR liIndex = 1 TO APRINTERS(laSetups,1)
        IF ( UPPER(laSetups[liIndex,1]) == ;
            UPPER(ALLTR(THIS.PSDriverSetupName)) );
            AND ;
            ( NOT (UPPER(laSetups[liIndex,3]) == ;
            UPPER(ALLTRIM(DRIVER_TO_USE)) ) )
            llFoundAndNotAppropriate = .T.
            EXIT
        ENDIF
    ENDFOR
    IF NOT llFoundAndNotAppropriate
        THIS.PSDriverSetupName = ALLTRIM(tcVal)
    ENDIF
ENDIF
ENDPROC
```

如你所见，这个 **assign** 方法允许你赋于任何你喜欢的驱动器名称，即使打印机设置不存在，但它不允许你给一个打印机设置分配一个已经在使用中的、并且所使用的打印机驱动程序与你需要的不同的名称。这个条件防止了由于某些人已经为一个非 **PostScript** 驱动程序使用了与你同样的名称而出现意外的情况。

使用它的默认内部打印机设置名称或者你分配的打印机设置名称，**PDFListener.VerifyPrinterSetup()** 执行一个对位于 **System32** 目录中的 **Windows** 部件 **PRINTUI.DLL** 的调用。这个调用将在一行代码中执行几个任务：

1. 安装需要的基本文件；
2. 将它们与你命名的打印机相关联；
3. 给这个设置加上你的商标、并且可以被使用；
4. 将打印机设置标记为共享，以防你正在一个专用的打印服务器盒子上使用这个应用程序（原文：**Marks the printer setup shared, in case you're using this application on a dedicated print server box**，我也实在搞不清楚这个 **Box** 是个什么 **Box** 了）。

这里是该调用的内容：

```
THIS.DoStatus(INSTALLING_DRIVER_LOC)
lcCmd = ;
    [%windir%\\system32\\rundll32.exe ] + ;
    [printui.dll,PrintUIEntry /if /b ] + ;
    ["] + THIS.PSDriverSetupName + ["] + ;
    [ /f %windir%\\inf\\ntprint.inf /r ] + ;
    [ "lpt1:" /m "] + ;
    DRIVER_TO_USE + [" /Z]
llReturn = THIS.oWinAPI.ProgExecute(lcCmd)
THIS.ClearStatus()
```

看不懂是吧？别怕，你可以在 **DOS** 命令行下运行下面这个命令来找出所有这些参数的意思：

```
%windir%\\system32\\rundll32.exe printui.dll, _
    PrintUIEntry /?
```

你可能注意到我把端口分配成了 **LPT1**。初始的端口分配是不重要的，因为你将总是让 **VFP** 去把 **PostScript** 输出发送给一个文件。

上面的 **THIS.DoStatus()** 调用让你可以向用户提供一个自定义的提醒消息，告诉用户你的活动，因为 **Windows** 调用也许会花一点时间来拷贝驱动程序文件（见图 **1**）。如果你对用户反馈不感兴趣，那么可以使用 **ReportListener.QuietMode** 开关来屏蔽这个消息。最后的结果是一个带有商标的打印机设置（见图 **2**）。

你应该选择什么打印机驱动程序？**PDFListener** 把 **DRIVER_TO_USE** 常量定义为 **"Apple Colol LaserWriter 12/600,"**，但几乎任何 **PostScript** 打印机，只要能被安装有你的应用程序的 **Windows** 版本所识别，就都可以用。在前面命令字符串中用到的系统

文件 NTPRINT.INF 提供了很大的选择范围。参见 www.cs.wisc.edu/~ghost/doc/printer.htm 处的一个可能出现的打印机问题列表，但大多数情况下，任何 PostScript 打印机都能支持你所需要的解析率、页面大小、色彩等等。

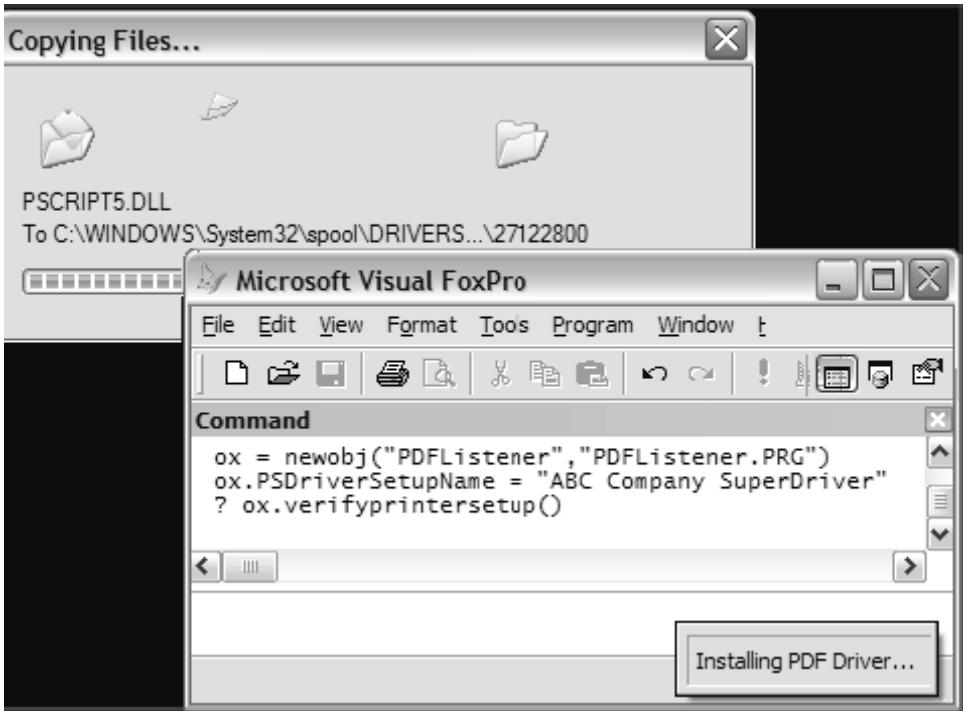
使用自由软件的责任

（译者注：这部分没什么意思，翻译起来倒费劲的很，就不翻译了。有兴趣的看看原文吧！

I use the GNU distribution of GhostScript, provided under the GPL License; an alternative Aladdin GhostScript distribution and a commercial distribution are also available. Refer to www.cs.wisc.edu/~ghost for details on the different distributions available and directions for obtaining them. I accompany my distribution of GhostScript files with a copy of the full GPL license under which artofcode LLC, its copyright holder, distributes GNU GhostScript. Because I don't include the full source files for GhostScript in my INSTALL.DBF, I supply a written offer to provide full source for the GhostScript programs, for no fee except the distribution costs, if requested to do so. You'll find these items in a text file named LICENSE.TXT, copied to disk from INSTALL.DBF with all the GhostScript files.

Note that these license terms don't require me to surrender my application's source code upon request. In the GPL view, as defined by the Free Software Foundation (www.gnu.org), I am aggregating my application with GhostScript but I don't derive my application from GhostScript. I don't link my code modules with, or modify, GhostScript source. The acid tests: An end user could call the GhostScript executable outside my application, and ? if my application were removed but the GhostScript files remained, GhostScript would still work.）

使用 PRINTUI.DLL 来编程的安装一个打印机驱动程序也许并非在所有版本的 Windows 中都能见效。在这里展示的版本工作在 XP 中，并且稍微修改一下就可以工作在 Windows 2000 中（参见微软的 189105 号 KB 文章以了解细节）。如果你能够使用 Windows 的 Shell Scirpting 的话，编程的安装打印机驱动程序也并非是唯一的办法。检查你的 Windows 目录中带有 PRN*.VBS 这样名称的文件，你将找到一个用于控制打印机的脚本工具的宿主。



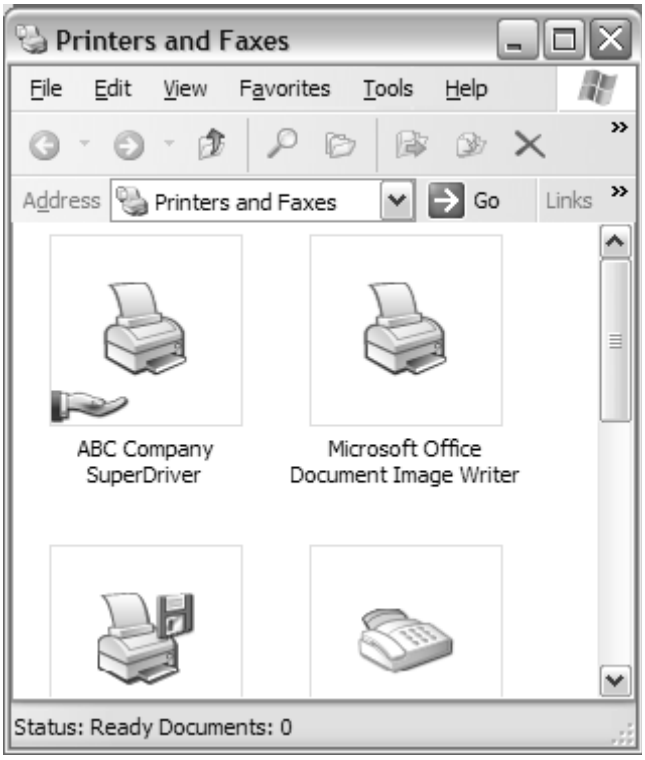


图 1 和图 2：如果你使用 PDFListener 来动态安装打印机设置，用户将会看到的标准 Windows 安装对话框、带有你的商标的 ReportListener 状态消息。以后他们只会看到你的应用程序的专用打印机设置

记得我说过这个方法可以工作在任何版本的 **FoxPro** 下吗？在 **FoxPro** 程序员使用 **Windows** 打印机驱动程序之前，我们使用 **_GENPD** 来为 **FoxPro** 基于特征的输出提供打印机说明。默认的 **_GENPD** 应用程序带有一套相当好的 **PostScript** 机制，包括一系列基本的 **PostScript** 字体描述。你可以 **SET PDSETUP TO <**你的 **PostScript** 设置**>**，**SET PRINTER TO<**文件名**>**——而不需要专门的安装，你的 **DOS** 应用程序可以逐字节的生成象在 **Windows** 中那样好的 **PostScript** 文件。

在运行时设置

假定你在使用 **Visual FoxPro** 而不是 **Dos** 或者 **UNIX**，那么现在你就走到第三步：告诉 **FoxPro** 去使用你指定的驱动程序了。你已经给 **PDFListener** 指定了你的打印机设置的名称，因此很容易就可以执行一个标准的 **FoxPro** 命令来告诉 **VFP**：

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

因为 **PDFListener** 工作在 **VFP 9.0** 下，它增加了一些特别的魔法来比过去的版本更完整的保存和恢复打印机设置。在 **SET PRINTER TO NAME** 前，它在保护方法 **PrepareFRXPrintInfo** 中使用了 **SYS(1037,2)** 来把当前 **VFP** 打印机环境保存到指定的 **FRX** 数据工作期中的一个游标中。当打印过程完毕后，它使用了另一个保护方法 **UnloadPrinterInfor** 和 **SYS(1037,3)**来恢复用户的整个打印机环境。使用这种技术可以被用来保护任何非默认选项，例如页面大小和方向之类用户只用在他或者她的 **VFP** 环境中的设置。

PDFListener 在你从 **LoadPrinterInfo** 方法中可以看到的第三个方法——**AdjustVFPPrinterSetups**——中去处理这些任务。我给它取了这么个名字是为了强调它的“不碰 **Windows** 环境，只改动你的 **VFP** 打印行为”这个事实。跟在 **LoadPrinterInfo** 中前面的那两个安装方法调用所不同的是，这是一个保护（**PROTECTED**）方法，因为你只会想把它当作是一个定义好的输出序列的一部分来调用。它的“**Push(**译者注：记得菜单中的‘推进堆栈’吗？）”机制总是应该跟常见于报表处理过程末尾的 **UnloadPrinterInfo()** 所使用的“**Pop**（象菜单一样的弹出）”回 **VFP** 打印机环境相匹配。如果你决定增加 **FRX** 和 **LBX** 打印机环境的清除和恢复代码，你可以把它象我在下一节中介绍的那样添加到这些方法中去。

确保在报表和标签中使用你的打印指令

如果你正在用报表和标签来输出，那么只对那些没有保存过打印机环境的 **FRX** 和 **LBX** 文件使用这种技术。否则的话，**SET PRINTER TO NAME** 指令将不会对报表系统产生影响。（在 **VFP 9.0** 中，新的报表和标签默认是不会储存打印机环境信息的）当你在自己的源代码中加入 **PDFListener** 以后，**PDFListener** 并不会自动为你处理好这个细节问题。请确保只对那些没有它们自己的打印机设置的报表和标签使用这里提供的 **TEST.PRG**。你可以修改 **PDFListener** 去删除这些用于以后恢复的数据，使用一个私有的游标和一种类似于它处理外部 **VFP** 打印环境的技术。

如果你将一个打印机环境存储在一个报表中，并且后来又通过 **VFP** 自己的菜单选项清除了它，请注意，该选项在顶上一条记录的 **Expr** 字段中保留了一个 **COLOR** 开关。对于 **PDF** 的输出，你应该考虑一下你需要的是颜色还是灰度。如果你想要颜色，那么请确保 **Expr** 字段包含有 **COLOR =2** （表示打开）。

> 如果你正在用 **VFP 9.0** 工作，你也可以不去管这个 **Expr** 字段，只要在头一条记录的 **Picture** 字段中增加一个 **COLOR =2** 就可以覆盖原来的设置。要了解更多情况的话，请阅读帮助文件中的 **“Understanding and Extending Report Structure(理解和扩展报表结构)”**这个主题，并熟悉一下 **FILESPEC\60FRX.DBF** 专用表。

得到结果

现在你走到了第四步：生成文件并调用 **GhostScript** 来转换它。这是比较简单的部分了。**PDFListener** 是从 **FFC** 基础类库中的 **_ReportListener** 派生出来的，因此它会维护一个报表的集合。它扩充了 **_ReportListener.AddReport** 方法来为你的集合中的每个报表建立临时 **PostScript** 输出文件名，并为你计划为每个报表所使用的 **Report FORM** 命令子句增加一些智能的分析，以确保 **REPORT FORM** 命令包含有足以生成文件输出的重要信息。它还用两个后处理操作扩充了 **_ReportListener.RunReports** 方法：使用前面讲过的 **UnloadPrinterInfo()**（第五步）来恢复 **VFP** 的打印机环境，并运行它的 **ProcessPDF** 方法。

ProcessPDF 首先建立一个 **GhostScript** 命令文本文件，列出在你的报表运行时的所有临时 **PostScript** 文件（第六步）：

```
#DEFINE GS_COMMAND_STRING_1 " -q -dNOPAUSE " + ;
```

```
    " -I./lib;./fonts " + ;
    "-sFONTPATH=./fonts " + ;
    "-sFONTMAP=./lib/FONTMAP.GS " + ;
    "-sDEVICE=pdfwrite -sOUTPUTFILE="

#DEFINE GS_COMMAND_STRING_2 " -dBATCH "

PROTECTED PROCEDURE MakeGSCommandFile()
LOCAL lcFile, lcContents, lcFileStr, liH
lcFile = FORCEPATH("C"+SYS(2015)+".TXT", ;
   THIS.GSLocation)
lcFileStr = THIS.GetQuotedFileString()
* GetQuotedFileString uses PDFListener's
* collection of temporary file names
* associated with each report in the run

lcContents = GS_COMMAND_STRING_1 + ;
   ["]+THIS.TargetFileName+["]+ ;
   GS_COMMAND_STRING_2 + ;
   + lcFileStr

* use forward slashes or doublebackslashes:
lcContents = STRTRAN(lcContents,"\","/")
IF FILE(lcFile)
   ERASE (lcFile)
ENDIF
liH = FCREATE(lcFile)
FPUTS(liH,lcContents)
FFLUSH(liH,.T.)
FCLOSE(liH)
RETURN lcFile
```

这里可能是代码中另一个引起读者惊慌的地方，因为这行调用 **GhostScript** 的代码可能会有许多的参数，并且对于文件路径方面要求特别的精确。我在上面指定的大多数参数是为了给 **GhostScript** 可执行文件指定它的库的。如果你改动了我用在 **INSTALL.DBF** 中的发布 **GhostScript** 库的策略，你将需要这段代码中的命令部分。

当你阅读了“在完整的 **PDFListener** 源代码中运行这个命令”的代码以后，你会注意到我临时把当前目录改动成了你放置 **GhostScript** 可执行文件的目录。我的目标是最大限度的减少任何以大写字母开头或者查找路径之类的问题，这些是在使用并非专门为 **Windows** 写的代码时可能会出现的标准危险。**FoxPro** 在对待文件名时有点漫不经心，为了提防正确的大小写的问题，我希望能减少潜在的危险。运行完 **GhostScript** 以后，我就立即把默认目录恢复回去。

研究一下 **ProcessPDF** 方法后，你会发现，在 **MakeGSCommandFile()** 返回完整的命令之后，它使用了与 **PDFListener** 原来用于安装打印机驱动程序时所用的方法不同的一个方法（**THIS.oWinAPI.ProgExecuteX**）来执行这个命令行。我发现使用 **Windows** 的 **Shell** 是最好的为不同类型的程序处理不同的错误处理策略和超时策略的办法。你可能有你自己的办法。你也许会更喜欢通过使用 **DECLARE DLL** 调用来调用由 **GhostScript** 的 **DLL** 支持库所提供的功能以回避整个问题。**GhostScript** 有一个拥有上佳文档的 **API**，所以你其实并不一定需要去执行 **GSWIN32C.EXE** 文件。不管怎么说，我更喜欢命令行接口的方式，因为它的那些命令行开关同样开发的很好，并且它的使用强调了这样一个事实：你可以在任何“能够 **RUN** 一个批处理文件或者调用一个外壳（**shell**）脚本”的应用程序中执行这些任务。

无论你决定怎么去做，调用 **GhostScript** 都会把你带到第七步。在上面这些工作完成以后，你现在有了一个包含有全部你让 **PDFListener** 去运行的报表的 **PDF**。

评估 VFP 9.0 功能的使用

你也许会感到疑惑：在 **VFP 8.0** 或者 **9.0** 中，为什么我没有选择在 **REPORT FORM** 命令上使用 **NOPAGEEJECT** 以将所有的报表绑定在单个 **PostScript** 文件中。由于将不需要去维护一个多个临时文件的集合，这样做将简化 **PDFListener**。如果不需要担心带着多个源文件的命令行长度的话，我还可以不用一个命令行文本文件就运行 **GhostScript** 了。

与 **NOPAGEEJECT** 相比，使用 **_ReportListener** 的报表集合或者一种类似的机制，能给你几个好处。最重要的好处是连接多个页面方向和大小都不同的报表的能力。相比之下，**NOPAGEEJECT** 是为多个报表打开对一个打印机的一个连接，只能使用由系列中第一个报表指定的一种页面布局方式。

使用 **GhostScript** 来添加文件还允许你可以混合来自于不同的输出处理技术的多个 **PostScript** 源文件，后者可以并不都是 **REPORT FORM** 命令的产物，它们甚至可以是不同的应用程序或者不同的 **FoxPro** 实例生成的 **PostScript** 文件。一旦你让 **PDFListener** 中提供的代码去建立了命令行文件，你就可以满足这些要求。

在 **Web** 服务器上，添加来自不同的应用程序、或者不同的实例的输出的能力可能尤其有用。你也许知道，**VFP 9.0** 已经经过了调整，以使它能够更容易的从一个 **DLL** 中运行 **Report forms**（多线程 **DLL** 还不行）。**PDFListener** 在这里增加了一些对 **OLE** 错误处理和 **_VFP.StartMode** 的认识以对你有所帮助。

如果你在一个 **Web** 应用程序的背景下运行这些代码，你也许会想去手动的运行安装程序、并确保适当的用户拥有对其功能的权限，包括输出目录和打印设置。不管怎么说，

在一个服务器背景条件下，你不需要担心应用程序透明度的问题！

你可能注意到了 PDFListener 向外提供了一个成员属性 generateNewStyleOutput，它表示你是否想要一个与你运行的报表相关联的 ReportListener 对象引用。尽管 _ReportListener 会运行这些 REPORT FORM 命令，但是这些命令本身可能分别属于旧样式或者新样式的 VFP 输出。PDFListener 默认把这个属性设置成了 .F.，这样生成的 PostScript 文件们将小得多而质量却一样得高。当你使用旧的报表引擎的时候，文件中包含的是由 PostScript 指令包围起来的你的报表数据的文本；而在新的那个中，每一页都是一个图形。

一些我忽略了的 GhostScript 功能

如果你熟悉 GhostScript 的话，你也许会问为什么我没有提到 RedMon ——一个 Windows 下用于 GhostScript 的端口重定向工具。据它的主页（www.cs.wisc.edu/~ghost/redmon）上说，RedMon 理论上提供了“透明 PostScript 打印”。因为它在一个打印过程中通过 GhostScript 来输送输出，RedMon 能够消除在一个后处理步骤中调用 GhostScript 可执行文件的需要。

运行时处理也许是透明的，但是透明的应用程序设置却无法解决。安装和设置 RedMon 是困难的任务，在一个不明的环境中既不实用又不可靠。阅读一下它的主页以了解已知的问题。

象 GhostScript 的 SETUP.EXE 一样，RedMon 是一个 Windows 专用的插件。据官方解释，对于成功的使用 GhostScript 来说，这两者都不重要。对 GhostScript 的交叉平台基础源代码作出贡献的那些人们在编写代码的时候都没有把 Windows 放在心上。

如果你正在使用一个 ReportListener 来提供象通过使用 GDI+ 来将 chart 图直接绘制到报表上那样的动态效果，请将 generateNewStyleOutput 设置为 .T.。如果你需要的话，_ReportListener 的 AddReport 方法给了你一个机会以为集合中的每个报表分别命名 ReportListener 对象引用。

交付和发布安装文件

你的源代码文件包含一个短的例程 CREATEINSTALLDBF.PRG，它能从我准备发布的不管什么版本的 GhostScript 中建立安装表。它会向你提问源目录，并建立用于存储我将发布的每个文件的记录——包括自由软件基金会要求的法律协议文本文件。

请注意最终结果的文件集并非完整的原始 **GhostScript** 文件集，而只是 **GhostScript** 需要的运行时文件集。其中包括可执行文件和 **DLL** 库、**GhostScript** 专用的设置和命令文件、以及字体文件。

当这个文件集被 **PDFListener** 从 **INSTALL.DBF** 中取出来以后，并没有被按照 **GhostScript** 发布包原来使用的目录结构放置。我使用了一个与我计划要给 **GhostScript** 的指示相匹配的优化方案以让 **GhostScript** 在运行时能找到它的文件。所有这一切看起来特别麻烦，但它保证了你的程序将会使用这个版本的 **GhostScript**、它所有必须的部件都在适当的位置上、而且没有其它什么要在机器上注册的东西。

即使这不是完整的 **GhostScript** 发布包，**INSTALL.DBF** 还是一个大文件（大概有 **8MB**）。占用空间较大的主要是字体定义文件；如果需要的话，你可以去掉其中的一些。访问 **GhostScript** 的网站，你会看到不同的发布包带有不同的字体集。你可以根据需要向字体集中增加、或者删除部分 **PostScript** 字体，但是你在这里看到的基本字体集带有 **GNU GhostScript** 中的默认字符映射，它将为大多数 **PostScript** 的输出提供适合的匹配。

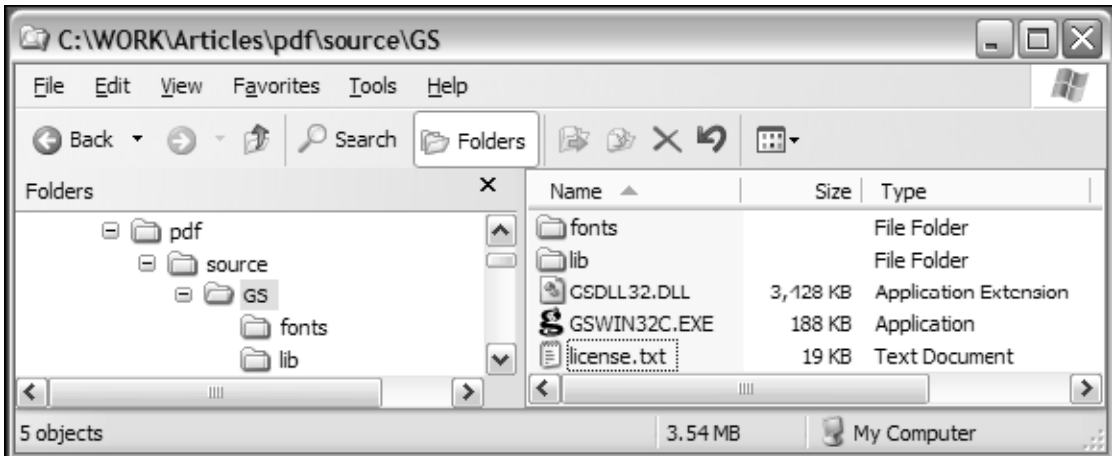


图 3、简化的 GhostScript 运行时目录和文件集

你可以选择将 **GSLocation** 成员属性的目标重新设定为指向 **INSTALL.DBF** 的位置以及 **GhostScript** 文件们的最终位置。在这个方案中，**PDFListener** 将总是将库安装在与 **INSTALL.DBF** 的位置有特殊关联的一个地方。

你也许更喜欢使用一种完全不同的机制。例如，你可以把 **INSTALL.DBF** 的内容作为一些独立的文件绑定到你的 **SETUP.EXE** 中去，以作为你的应用程序的支持目录。你也可以建立单独的 **VFPSTARTUP.EXE**，让它为你的应用程序在磁盘上重新建立这些文件、校验你的打印机驱动程序、将结果的名称和位置存储套设置文件中，以供报表运行时使用。我喜欢建立一个 **CONFIG.XML** 文件来处理这个任务；它可以被当作单个参数被拖放给我的 **EXE** 文件。在幕后，当我的 **EXE** 被当作带有一个“它可以加载为一个 **XML** 文件”参数的命令行调用时，它知道需要从该文件中读取它的设置值，并执行它的安装工作。

假定你修改了 **GhostScript** 命令行的参数以匹配你的文件的最终位置，对主要功能来说，这些选择没有一个是必须的。这就是为什么在源代码中提供的 **PDFListener** 期望有可供使用的 **INSTALL.DBF**、却又没有做什么努力来处理找不到这个文件的情况。

可供选择的商业产品

如果你已经有了自己喜欢的 **PDF** 生成办法，我当然不会推荐你再换一个了。现在有大量的可以生成 **PDF** 的 **VFP** 插件。

还是 **VFP 9.0 Beta** 版的时候，我曾与不少第三方工具的开发商一起工作过，他们再重构他们的产品以使用新的报表系统结构来在 **VFP 9.0** 中更快、更好的建立 **PDF**。结果相当令人满意——事实上，这些超级用户为我们当时正在使用的办法给予了真正的验证。**VFP 9.0** 的出现，使得“为每个人发明创造性的报表扩展、并且将它们平滑的集成到应用程序中去”变得更容易了、并且我希望也是更有趣了。这些伙计很快的就进入了状态。

那么，你也许会疑问，为什么你不选择一个第三方产品来在 **VFP** 中生成 **PDF** 呢？

我不知道。你对这个问题也许有大量的答案，包括实现所需的花费、法律协议的限制和发布的问题等等。如果你真的这么想的话，那这篇文章就给了你另一个可供考虑的选择。

使用 VFP 9.0 的报表事件

如果你总是使用支持对象的报表，并且你不需要去连接多个带有不同的页面大小的报表的话，那么，可以考虑一下为 PDF 输出使用 LoadReport 和 UnloadReport 事件。

你可以用 LoadReport 来检查任何报表的打印机环境，以及切换到一个“干净”的包括一个 PDF 所需的 COLOR 开关参数的环境版本。在 VFP 帮助文件中 LoadReport 事件主题里的一个示例 RLSwap 类可以帮助你上手。如果你为了报表的连续运行仔细查看了 CommandClauses 属性主题的话，你也可以使用 LoadReport 和 UnloadReport 来处理打印机设置和安装 GhostScript。

我使用了 _ReportListener 的 .Run 方法来连接报表，因此，根据用于每个报表的对象引用的不同，PDFListener 的 LoadReport 和 UnloadReport 事件也许会触发一次、几次、或者一次都没有。如果我希望调整 FRX 或者打印设置，我会在 PDFListener 的 .Run 方法之前或者之后去做这个事情。

有不少可供选择的策略，但即使是 VFP 9.0 报表的勇敢新世界也有自己的限制。也许同时使用一个报表集合和报表事件会看起来比较自然：

1、调用第一个 REPORT FORM 命令时带上一个 PDFListener，将额外的报表们储存在它的集合中。

2、在 LoadReport 中，执行安装，并继续运行最初的报表；

3、在 UnloadReport 中，运行其它的报表，并执行清理。

这种次序是无法工作的。不管是新还是旧样式的支持对象的 REPORT FORM 命令都不能被嵌套在一个 ReportListener 事件中，包括 UnloadReport。

它听起来比实际要做的工作多许多

你的源代码中包含一个作为示例的 TEST.PRG。这个程序中包含大量有用的“关于使用 PDFListener 以及它的大量非必要功能的建议”的注释。它还允许你测试这个过程。

该测试专用类会要求你提供一些 FRX 文件的名称，然后在一个循环中将它们添加到 PDFListener 的 报表集合中去（别忘了选择那些不带有保存了的打印机环境信息的 FRX 文件！）。接着它会运行 PDFListener 的 RunReports 方法，在这个过程中，PDFListener 将检查它需要的外部部件，如果需要的话则透明的安装这些部件，运行这些报表以生成 PostScript 文件，最后的后处理中则将 PostScript 报表输出到一个 PDF。

接着，**TEST.PRG** 的测试专用类使用一个对 **ShellExecute** 的调用来向你显示产生的 **PDF** 文档。

在这个过程之前和之后检查你的打印机设置，你会看到 由 **PDFListener** 增加的设置。检查你的打印对话框，你将看到你的默认 **Windows** 打印机和你自己的 **VFP** 打印设置保持不变。检查 **Test.PRG** 和 **PDFLISTENER.PRG** 所在位置下的目录，你会看到 **GhostScript** 的那些文件。查看你的手表，只要你没有选择了 **100** 个报表而且每个报表都有 **100** 页的话，你会发现并没有花多少时间。查看输出的内容，你会发现它正是你想要的东西。

现在，开始向你的用户提供他们所要求的东西吧！

下载：**506NICHOLLS.ZIP**

你的路线图是什么？

作者：David Stevenson

译者：fbilo

今天，在我脑子里有两个问题：微软为 Visual FoxPro 的未来设计的路线图是什么、以及我自己的专业和个人开发的路线图是什么？

微软的产品经理 Ken Levy 最近在一个在线论坛上提到 6 月 1 号将会有一个“VFP 路线图”文档被发布在 Visual FoxPro 的网站上，它将告诉 VFP 的开发人员们关于微软计划在 9.0 后怎样去增强 VFP 的更多详细的信息。

虽然我很乐意听到关于开发 VFP 10.0 的消息，但我也同样会乐意听到某些重要的新功能将通过更多的 Xbase 插件或者 Server Packs 的形式被增加到产品中去的消息。根据到目前为止微软对“下一个版本”所做过的含糊不清的回答、他们关于继续增强 VFP 的意见、以及将发布一个路线图的新闻等等种种情况来看，我认为后者更有可能成为事实。

我对听到的“关于增强 VFP 与其他技术的协作能力、以及它是否将被增强以利用将来到的 Longhorn 版本 Windows 的新功能”尤其感兴趣。

我将乐意听到更多关于直接绑定到 .NET Framework、允许 VFP 程序员轻松的从 VFP 中使用 .NET 功能的消息（这是过去已经作为一种未来的可能性被讨论过的东西了）。

你也许会想要在 6 月的第一天就去查看微软网站上的路线图文档。相信到时候我们了解了该文档的内容后，我会在后续的杂志中再谈它一下的。

你的专业和个人技能的成长路线图是什么？

最近，我读了一本很棒的关于怎样自我管理、并将注意力集中在与有意义的目标相关联的下一个关键行动上——并且怎样使用微软 Outlook 的任务列表和组来更好的组织和管理信息的书。

当我尝试书上的一些建议的时候，我惊讶的发现我可以把一封 Email 邮件拖放到任务按钮上，就可以把这封邮件嵌入到一个新任务的主体部分中或者作为该新任务的一个附件。这么一个小小的新发现就改变了我对需要处理的邮件作出反映的方式。这本书是 Sally McGhee 写的《Take back Your Life:Using Microsoft Outlook to Get Organized

and Stay Organized》（由 **Microsoft Press** 出版）。

我相信，在我每天用到的软件（包括 **VFP**）中肯定存在着大量我所未知的精彩功能，并且我觉得受到了很大的启发：我可以通过不断的学习用新的办法去使用常用工作软件来继续我的专业开发生涯。在过去的一周中，我学到了怎样在 **Outlook** 中使用 **VBA** 来与 **Access** 进行交流，同时我正在阅读一本关于优化 **SQL Server** 查询的书。

关于生成 PDF 的一个简介

请一定花点时间阅读一下本期杂志中由 **Lisa Slater Nicholls** 写的关于在任何版本的 **Foxpro** 中生成 **PDF** 文档的精彩文章。在这篇文章中，**Lisa** 放入了她在处理 **PDF** 文档时曾花费了她大量时间探索所得到的经验，而且她对关于透明的安装 **GhostScript** 的解释和示例代码将会填补那些曾经尝试过这种东西但却没有成功的人脑中的空白。

她解释了“应用程序透明度”的概念以及它在处理那些“必须被可靠的集成到你的应用程序安装过程中去”的插件工具时的重要性。通过将安装文件保存在 **FoxPro** 的备注字段中的办法，她取得了对 **GhostScript** 安装过程的完整控制，并且可以安装这个产品而不需要运行一个外部的 **SETUP** 程序。

仔细的看一下这篇文章以及经过丽莎详细注释的免费源代码，你的专业知识就会得到成长。还有，当你读完它以后，再看看 **VFP 9** 帮助文件中大量关于 **VFP 9** 中新的报表引擎的信息，这些内容是由最理解其设计和内部工作原理的人所写就的。

现在，长大吧！

控制锚定和报表预览

原著: Doug Hennig

翻译: CY

本月，Doug Hennig 将讨论一个你所期望的控制锚定的简单方法，以及如何控制报表预览窗口的外观和行为。

这里有一对我将要在某时讨论的问题，它并不是很长的整篇文章。我一直把它推迟着，希望它能够适合于另一个篇的所有主题。那还没有发生，并且由于它的重要性，我决定把它们组合成一篇文章，即使它们并不是完全相关的。然而，它们间有一个共同的主题：**VFP** 没有自带的控制某些对象的外观和行为的方法。这些主题涉及有疑问的锚定问题，以及如何控制报表预览窗口的外观和行为。

锚定的问题

VFP9 里增加的 **Anchor** 属性是个有趣新增物，它可以让你消除大量繁琐的代码（如果你过去是手工处理表单缩放），并且当它的容器被缩放时，可以很容易正确指定表单上每个控件的行为。然而，当 **VFP** 开始缩放或移动控件时，有时 **VFP** 似乎有它自己的想法。问题是它记得控件的原始大小和位置，和表单的原始大小以及它基于那些值作出判定。如果你通过程序改变了这些，缩放将不会再按预期那样工作。

这里有个示例。在确定的条件下（通常取决于安全、配置，等等），有时我隐藏了控件并移动其它控件到不同的位置，因此在表单上的可见控件间就不会有间隙。而问题是当控件的 **Anchor** 被设为非零值时，**VFP** 将记得被移动控件的原始位置，而表单看起来就不对了。图 **1** 展示了包含于下载文件里的 **ResizeDemo1** 表单在表单设计器里的样子。

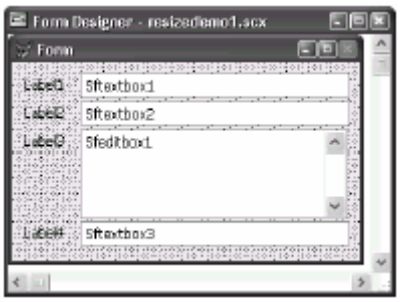


图 1：ResizeDemo1.SCX 在表单设计器里的样子。

下列代码在 **Init** 方法里隐藏了第二个标签和文本框，并移动了下面的控件。然而，正如你在图 **2** 里所见，编辑框和第三个文本框并没有在表单运行时移动到正确的位置。

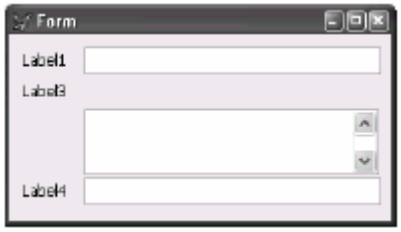


图 2：控件并没有放置正确，是因为它们的 Anchor 属性导致的没有按预期的移动。

```
with This
    store .F. to .Sflabel2.Visible, .Sftextbox2.Visible
    .Sflabel3.Top = .Sflabel2.Top
    .Sfeditbox1.Top = .Sftextbox2.Top
    .Sftextbox3.Top = .Sfeditbox1.Top + ;
    .Sfeditbox1.Height + 3
    .Sflabel4.Top = .Sftextbox3.Top + 3
    .Height = .Height - 25
endwith
```

解决这个问题的方法很简单：在移动前对每一个控件设置 **Anchor** 属性为零，然后再恢复 **Anchor** 属性为相应的值。当然，说的比做的容易，代码将不得不重复处理表单上的每一个控件，包括如页框等容器里的，保存当前的 **Anchor** 值在某处（比如数组），然后在移动控件后反过来做一遍。

为了使处理这些表面上的琐碎事更容易些，我加入了 **nSavedAnchor** 和 **lSaveAnchor** 属性到我的每一个带 **Anchor** 属性的基类里（在 **SFCtrls.VCX** 里）。**LSaveAnchor** 属性有个带下列代码的赋值方法：

```
lparameters tuNewValue
if tuNewValue
    This.nSavedAnchor = This.Anchor
```

```
    This.Anchor = 0
else
    This.Anchor = This.nSavedAnchor
endif tuNewValue
```

这两个属性的目的是为了按要求保存和恢复 Anchor 值。除了有 SaveAnchor 和 RestoreAnchor 方法，还有一个简单的属性（lSaveAnchor）带有赋值方法，意味着你可以利用 Container.SetAll('lSaveAnchor', .T.)来将容器里的每一个控件保存它的 Anchor 当前值，Container.SetAll('lSaveAnchor', .F.)可以用于恢复保存的值。

我在许多地方利用 lSaveAnchor 来避免缩放的问题：

◆ 为了解决 ResizeDemo1.SCX 的问题，在 Init 里移动代码前加入 This.SetAll('lSaveAnchor', .T.)，并在移动后加入 This.SetAll('lSaveAnchor', .F.)。这两句是其实是有的，但是被注释的，把注释去掉，再次运行表单，这次它将显示正常（见图 3）。

图 3：在移动控件前关闭锚定结果出现在正确的位置。

◆ 我有一个类（SFWizardForm）作为向导对话框的基类。SFWizardForm 有一个页框可以缩放以适应表单，并且 Tabs 设置为.F.，给出了在向导里每页不显示制表符的不同步骤的外观。在设计时，表单实际上比页框略宽，因此会留有部分空间可以双击带出表单的代码窗口而不是页框的。在运行时，Show 方法设置了表单的宽度和页框的宽度。当然，这个问题是导致页框中每页里的控件都会被缩放，而这却不是必须。因此，Show 方法利用下列代码来缩放表单：

```
with This
    .SetAll('lSaveAnchor', .T.)
    .MinWidth = min(.MinWidth, .pgfWizard.Width)
    .Width = .pgfWizard.Width
    .SetAll('lSaveAnchor', .F.)
endwith
```

◆ 因控件是动态例化的，不同于在表单或类设计器被加入表单，如果它们的 Anchor 值没有被设为非零值，它们将不会被正确放置。如果你在例化控件前缩放表单，它甚至会是更恶劣的。如果这声音只是人为的假想，看看下面：在向导对话框里，在第一步用户所做的选择决定了在第二步里那些控件可见。因此，控件没有被例子化，一直到页框里的页被激活。为了支持这个，我的 SFPage 类（页框里每一

页的基类，定义于 SFCtrls.VCX），包含下列代码在它的 Activate 方法里（这时省略了其它代码）：

```
with This
    if not empty(.cMemberClass) and ;
    type('.oMember.Name') <> 'C'
    if '\' + upper(.cMemberLibrary) $ set('CLASSLIB')
    .AddObject('oMember', .cMemberClass)
    else
    .NewObject('oMember', .cMemberClass, ;
    .cMemberLibrary)
    endif '\' ...
    * llSave = pemstatus(.oMember, 'lSaveAnchor', 5)
    with .oMember
    if llSave
    .lSaveAnchor = .T.
    endif llSave
    .Top = 10
    .Left = 10
    .Width = .Width + This.Parent.Width - ;
    Thisform.nInitialWidth
    .Height = .Height + This.Parent.Height - ;
    Thisform.nInitialHeight
    .Visible = .T.
    if llSave
    .lSaveAnchor = .F.
    endif llSave .ZOrder(1)
    endwith
    endif not empty(.cMemberClass) ...
endwith
```

在一个特定的向导表单，我越过这个方法根据第一页上的选择来设置 **cMemberClass** 和 **cMemberLibrary**，然后以 **DODEFAULT()**来动态例化指定的控件。然而，如果你运行 **ResizeDemo2.SCX**，缩放表单，然后选择第 **2** 页，你将会注意到页面上的控件并没有缩放，位置也不正确。为了修复这个问题，取消在 **SFPage.Activate** 里注释行的注释，并再次运行表单。这时，当你缩放表单并选择第 **2** 页时，控件将会被缩放，就象是当你在缩放时它们就已经是表单的成员。还要注意的是应当记得表单的原始大小：**nInitialWidth** 和 **nInitialHeight** 属性。我的基类表单，**SFForm**，在它的 **Load** 方法里初始化了这两个自定义属性：

```
with This
    .nInitialWidth = .Width
```

```
    .nInitialHeight = .Height
endwith
```

◆ •我喜欢在用户关闭表单时保存表单的大小和位置，然后在用户下次运行时再恢复这些设置。我在 2000 年一月份的文章“坚持不出汗”里讨论过一个帮助类，SFPersist，和许多的子类。通常，我只是加入一个 SFPersist 子类，设置一些属性，等等（我所使用的特定类，取决于我想把设置保存到何处，有保存为表，INI 文件，还有 Windows 注册表）。SFPersist 的 Init 方法，关注于恢复先前所保存的设置以及清除保存的设置，因此它会自动的运行。然而，如果表单的控件有非零的 Anchor 值，自动恢复表单的大小时就会有问题。任何在 SFPersist 对象(因为它们的 Zorder 排序较低)之后例化的控件，将不能正确的缩放，因为表单在它们例化前被缩放了。为了解决这个问题，SFPersist 有一个 lRestoreOnInit 属性，仅当这个属性为.T.，那么 Init 将调用 Restore 方法。于是，在我拖入一个 SFPersist 对象到表单上时，我把它的 lRestoreOnInit 属性设置为.F.，然后再从表单的 Init 方法里调用它的 Restore 方法。可以确保所有的控件在表单缩放为它的大小前都被完全例化。

控制报表的预览窗口

在先前的 VFP 版本里，控件报表预览窗口的外观是不容易的：你将不得不创建一个你所需要特性的虚窗口，然后再在 REPORT 命令的 WINDOW 子句里指定这个窗口。即使有这些繁琐的代码，你也仍然只能对预览窗口外观进行小小的控制。

```
loForm = createobject('SFPreviewForm')
with loForm
    .Caption = 'My Preview Caption'
    .Width = _screen.Width
    .Height = _screen.Height
endwith
keyboard '{CTRL+F10}'
if wexist('Print Preview') and ;
    not wvisible('Print Preview')
    show window 'Print Preview'
endif wexist('Print Preview') ...
report form MyReport to printer prompt preview ;
nodialog noconsole window (loForm.Name)
if wvisible('Print Preview')
    hide window 'Print Preview'
endif wvisible('Print Preview')
```

```
define class SFPreviewForm as Form
ScrollBars = 3
DoCreate = .T.
Caption = ""
HalfHeightCaption = .T.
MinButton = .F.
BackColor = rgb(255,255,255)
Name = "SFPreviewForm"
Enddefine
```

在 **VFP9** 里控制报表预览窗口是非常容易的，因为它是个 **VFP** 的表单，因此它有通常的属性，如 **TOP**、**Left**、**Width**、**Hight**、**Caption**，等等。另外，它还有一些定制的属性可以控制它的外观和行为。

控制预览窗口的关键是例化你自己的副本而不是让 **VFP** 创建一个预览窗口。这可以让你在运行报表前来设置窗口的部分属性。注意到你实际上并没有例化一个预览表单，而是一个控制表单的代理对象。你可以通过调用 **ReportPreview.APP** 来实现，传递给它一个包含对代理对象指针的变量。一旦你创建代理对象，你就可以设置你所要的属性。

为了使用自定义的代理对象和表单，你必须例化一个报表监听对象，或者是 **ReportListener** 基类，或者是你所需要的子类，并把一个对代理对象的指针存放入它的 **PreviewContainer** 属性。当你运行使用报表监听器的报表时，报表引擎将明白 **PreviewContainer** 已经包含一个预览对象，然后它将使用这个而不是例化它自己。

这里有一个示例，来自于 **TestPreview.PRG**。注意这个代码设置预览窗口的标题、大小和位置为期望值：

```
local loPreview, ;
loListener

* 创建预览代理对象并设置其所要的属性

do (_reportpreview) with loPreview
with loPreview
    .Caption = 'This is my report'
    .Top = 10
    .Left = 10
    .Height = 500
    .Width = 500
endwith

* 创建监听器对象并告诉它使用我们的预览对象
```

```
loListener = createobject('ReportListener')
loListener.ListenerType = 1
loListener.PreviewContainer = loPreview

* 运行带监听器的报表

use _samples + 'Northwind\Orders'
report form TestPreview.FRX preview object loListener
```

控制其他设置

在 **VFP** 帮助文件的“支持默认预览容器”的主题里讨论了预览代理对象的其他你可以控制的属性，包括：

- ToolbarIsVisible-设置为.F.可以隐藏工具栏。
- CanvasCount-决定在预览窗口可见的页数。比如，设置为 4 在预览窗口里一次可显示 4 页。
- ZoomLevel –指定预览的缩放级别。列举值为 1（10%）到 10（整页）；VFP 帮助文件里有这些值。
- CurrentPage –在窗口中初始显示的页。

禁止工具栏的打印按钮

在 **Universal Thread (**www.universalthread.com**)**上经常的问题是如何禁止工具栏的打印按钮，而解决办法，在 **Universal Thread** 上如 **Colin Nicholls** 所解释的，是非常简单：

设置非正式的 **AllowPrintFromPreview** 属性为**.F.**，参见示例 **TestPreview.PRG**。

停靠工具栏

你可能想让预览窗口的工具栏自动停靠，而不是让它浮动。这比改变其它设置更需要点技巧，因为代理对象没有这个属性。由于预览窗口和它的工具栏一直到报表显示时才被例化（在代理对象的 **Show** 方法里），当其他代理属性被设置时，我们无法直接处理表单或工具栏。

看看代理对象的 **Show** 方法代码来想想办法，我注意到如果 **oForm** 属性已经包含一个预览表单的指针，代理对象并不会再例化另一个。我也发现工具栏是在预览表单类的 **Init**

方法里例化的。因此，类似于通过例化你自己的代理对象来获取更多的控制，如果你例化了你自己的预览表单，将可以让你在报表运行前就可以更多的控制工具栏。

我加入下列代码到 **TestPreview.PRG** 的 **WITH loPreview** 块：

```
.oForm = newobject(.PreviewFormClass, .ClassLibrary)
.oForm.oToolbar.Dock(0)
```

这个 **PreviewFormClass** 属性包含了要例化作为预览表单的类名，并且由于同一个的 **VCX** 包含了代理类和预览表单类，这应该是最需要的。不幸的，这有两个问题：

- PreviewFormClass 类是保护的，因此第一个语句产生了“PreviewFormClass 属性没有发现”的错误。
- 即使 PreviewFormClass 是全局的，它的值是在代理对象的 Show 方法里动态设置的，因为同样的类根据不同的参数来使用，比如是否必须是顶层表单。正如我先前所讨论的，Show 太迟了。

幸运的，有另一个方法可以实现，通过使用预览扩展句柄。微软考虑到我们想要对预览窗口作更多控制，而不仅仅只是在框框里，因此他们通过提供一个钩子对象来打开结构。如果这个调用预览扩展句柄的钩子对象有存在，预览表单就可以在多处调用对象的适当方法：

- 当预览表单被显示或关闭时
- 当用户按键时
- 当快捷菜单显示时
- 当报表当前页预览窗口绘出时

在 **VFP** 帮助文件里同样的“支持默认预览容器”的主题里讨论了扩展句柄对象的接口。这里有一个说明是当预览窗口显示时它会自动停靠工具栏（注意必须定义空的方法，因为使用了扩展句柄对象，所有的这些方法在同时被调用）：

```
define class SFPreviewExtensionHandler as Custom
PreviewForm = .NULL.

* 停靠工具栏（如果有）

procedure Show(tnStyle)
if vartype(This.PreviewForm) = 'O'
    This.PreviewForm.Toolbar.Dock(0)
endif vartype(This.PreviewForm) = 'O'
endproc

* 在翻译时清理

procedure Release
```

```
    if type('This.PreviewForm') = 'O'
        This.PreviewForm.ExtensionHandler = .NULL.
        This.PreviewForm = .NULL.
    endif type('This.PreviewForm') = 'O'
    endproc

    procedure AddBarsToMenu(tcShortcutMenuName, tnBarCount)
    endproc

    procedure HandledKeyPress(tnKeyCode, tnShiftAltCtrl)
    endproc

    procedure Paint
    endproc
    enddefine
```

告诉代理对象以它的 **SetExtensionHandler** 方法来使用这个预览扩展句柄。**TestPreview.PRG** 在它的 **WITH loPreview** 块里使用了下列代码：

```
    loHandler = createobject('SFPreviewExtensionHandler')
    .SetExtensionHandler(loHandler)
```

总结

锚定是 **VFP9** 里一个极大的扩展，但是，除非你明白它的特性，否则你可能要抓破头才能面对它来做你想要的事。类似的，不再要苦对着预览窗口，通过例化你自己的副本来控制它，并且让它按你所需要的来显示和动作。

来自 VFP 团队的贴士

原著:The Microsoft Visual FoxPro Team

翻译: CY

本月“来自 VFP 团队的贴士”栏目，将展示如何利用 UpdateListener 类在报表生成的过程中为用户提供反馈，将展示报表预览，与此同时将利用后继机制在报表运行的同时创建一个 XML 输出文件。

在创建 XML 时从 UpdateListener 里驱动报表预览

本例(在下载文件里的 **VFP9_successor.PRG**)可以让你以辅助对象模式来运行报表，通过利用 **UpdateListener** 来作为驱动监听器，并且指定它的 **ListenerType** 属性为提供预览。它也利用了后继属性在报表运行的同时来钩住**(hook)XML** 输出的生成。

```
LPARAMETERS tlQuietMode, tlDeletedOFF, ;
tlIncludeXMLListenerFeedback, ;
tlIncludeXMLListenerRDLOutput
#DEFINE XMLLISTENER_TYPE 4
#DEFINE PRVLISTENER_TYPE 1

 IF NOT tlQuietMode
    MESSAGEBOX( ;
         "This demo shows you two Listeners paired "+ CHR(13)+ ;
         " to do several different things on one report run:" +;
         CHR(13)+CHR(13)+ ;
         "(1) XMLListener turns off its simple WAIT WINDOW UI"+;
         CHR(13)+ ;
         "and silently generates its XML output. " +;
         CHR(13)+ CHR(13) + ;
         "(2) At the same time, UpdateListener "+ CHR(13)+;
         "responds to reporting events to provide " + CHR(13) +;
         "a more sophisticated interface." + CHR(13)+ CHR(13) +;
```

```
        "(3) UpdateListener also hosts a report preview." + ;
        CHR(13) + CHR(13) + ;
        "We need to use a LONG report for this demo !" + ;
        CHR(13)+CHR(13) + ;
        "You will be able to cancel the report before " +;
        "it's finished if you like.")
ENDIF

LOCAL cDir, aDummy[1], cDeleted, cPath, oListener
cPath = CURDIR()
cDeleted = SET("DELETED")

CD (JUSTPATH(SYS(16)))

IF NOT (FILE(_REPORTOUTPUT) AND FILE(_REPORTPREVIEW))
    MESSAGEBOX(;
        "Need _REPORTOUTPUT and _REPORTPREVIEW for this demo!")
    CD (cPath)
    RETURN
ENDIF

IF tlDeletedOFF
    SET DELETED OFF
ELSE
    SET DELETED ON
ENDIF

DO (_REPORTOUTPUT) WITH XMLLISTENER_TYPE, 2
IF FILE(_oReportOutput[;
    TRANSFORM(XMLLISTENER_TYPE)].TargetFileName)
    ERASE (_oReportOutput[;
        TRANSFORM(XMLLISTENER_TYPE)].TargetFileName) ;
        NORECYCLE
ENDIF

oListener = NEWOBJECT("UpdateListener","Listener",;
    _REPORTOUTPUT)
oListener.ListenerType = PRVLISTENER_TYPE
oListener.Successor = ;
    _oReportOutput[TRANSFORM(XMLLISTENER_TYPE)]

IF tlIncludeXMLListenerFeedback
    _oReportOutput[;
        TRANSFORM(XMLLISTENER_TYPE)].QuietMode = .F.
```

```
ELSE
    _oReportOutput[;
        TRANSFORM(XMLLISTENER_TYPE)].QuietMode = .T.
ENDIF

IF tlIncludeXMLListenerRDLOutput
    _oReportOutput[;
        TRANSFORM(XMLLISTENER_TYPE)].XMLMode = 2
ELSE
    _oReportOutput[;
        TRANSFORM(XMLLISTENER_TYPE)].XMLMode = 0
ENDIF

CD (cPath)
REPORT FORM (GETFILE("FRX")) OBJECT oListener

IF ADIR(aDummy, _oReportOutput[;
    TRANSFORM(XMLLISTENER_TYPE)].TargetFileName) > 0
    IF NOT tlQuietMode
        MESSAGEBOX("Your report XML file is " + CHR(13) + ;
            TRANSF(aDummy[1,2]) + " bytes." + CHR(13)+ ;
            "The printed output for this report "+ CHR(13) + ;
            "is " + TRANS(_PAGENO) + " pages long.")
    ENDIF
    MODI FILE (_oReportOutput[;
        TRANSFORM(XMLLISTENER_TYPE)].TargetFileName) NOWAIT
ENDIF
```

步步为营

作者：**Andy Kramek & Marcia Akins**

译者：**fbilo**

有一个我们经常会面对的问题，就是保证一系列连续的处理步骤都被顺利的完成。应付这个问题有多种办法，可它们各自又都存在着一些缺点。在本月的专栏中，Andy Kramek 和 Marcia Akins 试图找出处理这个问题最简单、并且最有效的办法。

安迪：我最近又在写一些关于将一系列表中的数据迁移到另一系列结构完全不同的表中去的代码，并且碰到了一个以前从未好好考虑过的问题。我不知道你是否考虑过这类对一系列按顺序执行的步骤之执行结果进行检查的题目。

玛西亚：你能把你的问题的本质说得更清楚些吗？

安迪：考虑一下以下情况。我必须运行一系列按指定顺序执行得独立步骤。但是当然，如果其中一个步骤执行失败了，那么我就不希望再继续执行后面的那些步骤。完成这个任务最明显的办法是做在一个 **IF...ELSE** 层次结构中，就像这样：

```
*** 按顺序调用这些步骤
llRetVal = FirstStep()
IF llRetVal
   llRetVal = SecondStep()
   IF llRetVal
      llRetVal = ThirdStep()
      IF llRetVal
         llRetVal = Fourthstep()
      ENDIF && 第三步
   ENDIF && 第二步
ENDIF && 第一步
```

玛西亚：这会有什么错？

安迪：这里有两件事情不对。首先，如果你只需要对付象上面这样的两三个步骤，这个结构是 **OK** 的。但是，在我的实际情况中却要对付 **30** 个步骤！这样一来，写出来的代码就完全没法看了，光是跟踪匹配的 **ENDIF** 语句就是个麻烦。

玛西亚：哦，是的，我能够理解这确实会很成问题。那么第二件事情呢？

安迪：好吧，当这个层次结构运行到最后的时候，我能够知道是否所有的步骤都完成了、或者其中某个步骤失败了。然而，我并不知道到底是哪个步骤有问题，于是我就不清楚该采取什么相应的措施。当执行失败的时候，在这个函数中的任何数据库事务都将被回滚，可不是什么事情都能在仅仅一个事务中就能处理好的。

玛西亚：嗯，知道是什么地方出错了总是很重要的，尤其是在这种数据迁移的程序中。你需要判定到底是源数据中有错误呢、还是升迁的逻辑中有着漏洞、或者干脆是新的数据结构中有问题。最简单的办法是错误一发生就弹出一个错误消息，就象这样：

```
*** 按顺序调用这些步骤
llRetVal = FirstStep()
IF llRetVal
    llRetVal = SecondStep()
    IF llRetVal
        llRetVal = ThirdStep()
        IF llRetVal
            llRetVal = Fourthstep()
            lcErr = IIF( llRetVal, '', '第四步出错' )
        ELSE
            lcErr = '第三步出错'
        ENDIF && 第三步
        ELSE
        lcErr = '第二步出错'
    ENDIF && 第二步
ELSE
    lcErr = '第一步出错'
ENDIF && 第一步
```

安迪：可这样一来，代码就会比我前面的东西还要难读了。我是想知道哪个步骤出错了，可它应该是个简单的办法。到目前为止，我们有 **30** 个步骤要做，而且我可能在一个 **IF...ENDIF** 块中就会上百行代码。这会很麻烦的。

玛西亚：嗯，你知道，我还从来没碰到过我不喜欢的数据。那么把所有这些步骤的名称都放在一个表里面（目前为止具体的表结构还不重要）、并象这样调用它们怎么样？

```
LOCAL lcErr, lnCnt, lcProcName, llRetVal
USE proclist
SET ORDER TO procseq
lcErr = ""
FOR lnCnt = 1 TO RECCOUNT()
    lcProcName = ALLTRIM( proclist.procname ) + "()"
    llRetVal = &lcProcName
    IF NOT llRetVal
        lcErr = lcProcName + " 执行失败"
```

```
        EXIT
        ENDIF
    ENDFOR
    IF NOT EMPTY( lcErr )
        *** 我们知道是哪个步骤出错了
    ENDIF
```

安迪：唔，这样当然清晰多了，但我还是有一些问题，比如怎样去处理参数。

玛西亚：我想，你应该清楚那些参数的名字吧？那么为什么不就把它们做成一个逗号分隔的参数列表字符串、放在步骤表中的一个备注字段里，然后把它们插入到代码中的括号里面去呢？

安迪：**Okay**，可现在还有一个问题，就是并非所有的步骤都会返回同样数据类型的值。例如，当一个步骤执行成功的时候它可能会返回一个空值，而另一个步骤返回的却可能是一个非空的值来表示执行成功。

玛西亚：你怎么不早说呢？数据驱动的办法适用于处理那些只有一种数据类型的返回值的情况，但是当你在对付返回值有特别的含义而不是简单的表示成功或失败的案例的时候，数据驱动的办法就会像 **IF...ELSE** 层次结构那样麻烦。

安迪：那么你是在说 **IF...ELSE** 层次还是最好的解决办法喽？

玛西亚：我什么时候这么说过？我只是说数据驱动的解决办法没有更好，但那并不代表着就没有别的办法了呀！我们用一个“串连的 **CASE**”语句怎么样？

安迪：什么？我还从来没听到过这种东西——难道它是我没注意到的 **VFP 9.0** 中的新功能吗？

玛西亚：不，它不是什么新东西，但它是我们在 **VFP** 中可以得到的最接近 **C++** 中的“**Switch**”结构的东西，在（**C++**中）这种结构里，通常情况下，每个 **CASE** 条件都会被运算、直到其中一个失败为止（与 **VFP** 的 **DO Case** 相反，后者会运行每个 **CASE** 直到其中一个成功）。

安迪：我想我得看到代码才能理解这种结构会怎么对我们有所帮助了。

玛西亚：好吧，它让你可以为每个步骤单独设置“成功”的条件，就象这样：

```
    DO CASE
        CASE NOT FirstStep()
            *** 应该返回 .T.
            lcErr = '第一步执行失败'
        CASE SecondsStep() <= 0
            *** 应该返回一个大于 0 的数值
            lcErr = '第二步执行失败'
```

```
        CASE EMPTY( ThirdStep() )
            *** 可以返回任何非空的值
            lcErr = '第三步执行失败'
        OTHERWISE
            *** 每一步都顺利完成了
            lcErr = ""
    ENDCASE
```

安迪：哦，这样当然要比嵌套的 **IF...ELSE** 层次可读的多了，并且它确实能够解决返回值数据类型不同的问题。

玛西亚：但是？我想马上会听到你说出个“但是”来了。

安迪：但是它没有解决我的第三个问题：有些时候，一个步骤的返回值并不只是表示执行成功或者失败，它还有别的意义。例如，一个用于建立主记录的步骤会返回一个新生成的主键，以便用这个主键来建立子表的记录。

玛西亚：你知道，你并非必须用这种办法来做。如果建立主记录的步骤执行成功了并且返回了一个 **true**，那么完全可以假定在主表中的当前记录就中就有你需要的那个主键。

安迪：是这样吗？如果我是在向一个 **SQL Server** 表插入数据的话，这样也行吗？

玛西亚：哦！我明白你的意思了。在这种情况下，你不能就用 **@@IDENTITY** 来返回在当前连接上生成的最近一个主键值吗？

安迪：我也希望这样，可即使在 **VFP** 中你其实也不应该完全相信这样的假定：只要插入数据成功，记录指针就一定会放在正确的位置上。有太多的事情可能会导致记录指针被隐蔽的移动，所以太容易取回错误的值了。那么，你怎么看？

玛西亚：不管哪种办法都有点跑题了。如果我理解正确的话，你真正的问题是你想要根据步骤返回的值来分别判定步骤的各种状态，是吗？

安迪：大致上是这样的。

玛西亚：唔，我觉得我们可以做一个约定：所有的步骤都应该返回一个参数对象。这样一来，我们就可以返回多个值了，其中一个值就表示执行是否成功这个状态。不过，由于必须要显式的去检查状态属性，所以你还是会碰到原来的代码不可读的问题。

安迪：那么你是在说我们要把这个：

```
    llRetVal = FirstStep()
    IF llRetVal
        *** Do whatever is next
    ELSE
```

```
    lcError = "FirstStep Failed"
ENDIF
```

换成象这样的东西：

```
loRetVal = FirstStep()
IF loRetVal.lSuccess
    *** Do whatever is next, including
    *** reading any required values from
    *** the parameter object
ENDIF
lcError = loRetVal.cErrorText
```

我猜这样会好一些。

玛西亚：不，这样好处很多。首先，我们排除了全部的 **Else** 条件，代码的可读性将会大大提高。其次，错误消息就在出错的地方弹出，这样就有意义的多。第三，我们可以毫无问题的处理多个不同数据类型的返回值。要我说的话，这种办法好多了！

安迪：我表示怀疑。它还是会留给我太多的 **IF** 语句。很遗憾我们不能用一个象 **TRY...CATCH** 这样的东西来做。

玛西亚：哦！为什么我没想到这个呢？你为什么说我们没法用 **TRY...CATCH**？

安迪：它是用于本地化错误处理的，不是吗？这里的问题是：错误发生在子步骤里面，但是我们却希望在父步骤里面去检测错误。

玛西亚：没错！**TRY...CATCH** 的设计目标就是这样的。如果在父步骤中的调用代码是在一个 **TRY...CATCH** 里面的话，那么，任何发生在子步骤里面的错误就被回传给父步骤的 **CATCH**。只要你不在子方法里面加入一个低级别的 **TRY...CATCH**，就不需要做任何专门处理，只要在需要的时候使用 **ERROR** 命令来生成一个错误就行了。看看这个（见下载文件中的 **TryBase.PRG**）：

```
*** 按正确的顺序调用所有的步骤
TRY
    FirstStep()
    lnRetVal = SecondStep()
    loRetVal = ThirdStep( lnRetVal )
CATCH to loErr
    ? loErr.Message
ENDTRY

FUNCTION FirstStep
IF something_went_wrong
    ERROR "由于某种原因，第一步执行失败"
ENDIF
```

```
FUNCTION SecondStep
IF it_broke
    ERROR "由于 it_broke，第二步执行失败"
ENDIF

FUNCTION ThridStep( tnInVal )
RETURN .T.
```

安迪：这个我喜欢。没有过多的 **IF** 语句，没有 **ELSE** 条件，而且我可能根据需要返回值或者参数对象。这种办法也可以用在多层的步骤调用上。提一个问题：如果一个低级别的步骤有它自己的 **TRY...CATCH** 的话会发生什么事情？它还能继续正常工作吗？

玛西亚：是的，它可以的，但这种情况有两个不同的处理办法。其中一个是使用 **THROW** 命令来将任何错误回传给最高级别的步骤（见下载文件中的 **UseThrow.PRG**）：

```
LOCAL loErr
TRY
    FirstStep()
    SeventhStep()
CATCH to loErr
    ? "由顶层 CATCH 捕捉"
    ? loErr.Message
ENDTRY

FUNCTION FirstStep
    SecondStep()

FUNCTION SecondStep
LOCAL loErr
    TRY
        ThirdStep()
        FifthStep()
    CATCH TO loErr
        THROW
    ENDTRY

FUNCTION ThirdStep
LOCAL loErr
    TRY
        FourthStep()
        SixthStep()
    CATCH to loErr
        THROW
```

```
    ENDTRY

FUNCTION FourthStep
    ERROR "第四步出错"
```

安迪：看看我是否理解这个了。顶层的程序在一个 **TRY...CATCH** 中调用 **FirstStep()**，接着，后者再调用 **SecondStep()**。**SecondStep()**则在另一个 **TRY...CATCH** 中调用 **ThirdStep()**，而最后 **ThirdStep()** 也用了另一个 **TRY...CATCH** 来调用 **FourthStep()**，后者会生成一个错误。当这段代码运行的时候，我们看到的结果将是"由顶层 **CATCH** 捕捉"消息加上“第四步出错”消息。我还注意到接下来的几层函数（**FifthStep()**、**SixthStep()**和 **SeventhStep()**）既不存在、也从未真正被调用过。

玛西亚：没错。这里唯一的问题是它隐藏了这么个事实：实际上，**FirstStep()**是唯一从头到尾运行完的程序。可我也希望知道任何其它步骤失败的信息。

安迪：是的，我当然明白知道哪里出错是非常有用的。那么你怎么解决它？

玛西亚：去掉在每一个低级层 **CACTH** 中用的 **THROW** 命令，代之以生成一个新的“本地”错误，在该错误的错误消息中加入更多的信息——就象这样（见下载文件中的 **GenError.PRG**）：

```
LOCAL loErr
TRY
    FirstStep()
CATCH to loErr
    ? "由顶层 Catch 捕捉"
    ? loErr.Message
ENDTRY

FUNCTION FirstStep
    SecondStep()

FUNCTION SecondStep
LOCAL loErr
    TRY
        ThirdStep()
    CATCH TO loErr
        ? "捕捉的是 " + PROGRAM()
        ERROR "第二步失败，原因是 " + loErr.Message
    ENDTRY

FUNCTION ThirdStep
LOCAL loErr
```

```
        TRY
            FourthStep()
        CATCH to loErr
            ERROR "第三步失败，原因是 " + loErr.Message
        ENDTRY

    FUNCTION FourthStep
        ERROR "第四步失败"
```

安迪：我明白了。现在，最后得到的消息就是“第二步失败，原因是第三步失败，原因是第四步失败”了。

玛西亚：没错。并非简单的将错误回传，通过在调用链条上回传的时候逐步对错误消息进行扩充，我们可以更清楚的知道发生了什么错误、错误发生在哪里。

安迪：我确实喜欢这样！它将会极大的简化我的代码、并使之更容易跟踪错误发生在那些连续运行的步骤中的什么地方。

下载文件：**596KITBOX.ZIP**