

2005 年 2 月刊

Try Catch 模型 — Randy Pearson & Lauren Clarke 著 Fbilo & LQL.NET 译

Page.3

设计模型的一个主要好处，是为面对对象的设计方式采用通俗的名称。与此类似，在讨论和教授新的编程实务的时候，为编程语言使用其专用的模型将会是很有帮助的。在 Visual FoxPro 8 中增加的 TRY...CATCH...FINALLY 结构向我们提供了一个增强编写代码实务的新的重大机会。在这篇文章中，Randy Pearson 和 Lauren Clarke 讲述了这种结构的一些常规用法，并为之进行了命名。

监听报表 — Doug Hennig 著 CY 译

Page.13

微软已经在 VFP9 里开放了报表引擎，它通过新的 ReportListener 基类来通信。利用 ReportListener 子类，VFP 开发者可以创建他们自己的定制输出。这个月，Doug Hennig 思考了 ReportListener，并通过示例展示了如何解决现实问题。

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

Page.22

来自 VFP 开发团队的本月的 TIPS 聚焦在 VFP9 SQL 引擎的速度提升上。VFP9 已经于 2004 年 12 月发布并向 MSDN 订户提供下载。

按 Pixel 对文本进行自动换行和段落重排 —Pradip Acharya 著 Fbilo 译

Page.25

偶尔你会碰到需要根据字符集的情况、对文本进行精确到 pixel 宽度的自动换行和段落重排。VFP 自己内部带有这个功能，例如在报表的溢出时自动扩展选项、在表单标签上、以及在缩放备注窗口的时候。除了已经过时并且无效的 Memowidth 设置以外，这个功能对于程序员来说是被隐藏而无法使用的。Pradip Acharya 提供的这个 WrapText 类是一个高级的精确文本自动换行和段落操作的工具。

在缩放中试试这个 —Andy Kramek & Marcia Akins 著 Fbilo 译

Page.33

这个月，Andy Kramek 和 Marica Akins 公婆俩研究了一下在一个应用程序环境中与缩放表单有关的一些问题。利用 VFP 9.0 的功能，基本的锚定控件任务已经变得非常简单了（参见 Foxtalk 2.0 2004 年 9 月刊专栏文章《The Kit Box: Anchors Aweight》），但锚并不能解决所有的问题。所以当表单缩放的时候，还是经常会碰到有必要使用自己的代码来实现期望结果的情况。问题是怎样去最好的实现这样的代码。

Try Catch 模型

原著: Randy Pearson & Lauren Clarke

翻译: fbilo & LQL.NET

设计模型的一个主要好处，是为面对对象的设计方式采用通俗的名称。与此类似，在讨论和教授新的编程实务的时候，为编程语言使用其专用的模型将会是很有帮助的。在 **Visual FoxPro 8** 中增加的 **TRY...CATCH...FINALLY** 结构向我们提供了一个增强编写代码实务的新的重大机会。在这篇文章中，**Randy Pearson** 和 **Lauren Clarke** 讲述了这种结构的一些常规用法，并为之进行了命名。

在 **8.0** 以前，在 **Visual FoxPro** 中有两种类型的错误处理方式：全局的 **ON ERROR** 设置，和特定类的 **Error()** 方法。从 **VFP 8.0** 开始，**TRY...CATCH...FINALLY** 提供了结构化错误处理，这是一个许多现代编程语言都具备的常用功能。

对于使用第三方应用程序框架来进行开发的程序员来说，结构化错误处理可能是 **VFP 8** 中最重要 的语言增强了。象集合和事件绑定这类的其它新语言功能，看上去在设计应用程序框架内部结构上有着 更高的价值，但结构化错误处理却适用于应用程序任何地方的代码中。当你熟悉了它的用法以后，你会发现 **TRY...CATCH...FINALLY** 就像 **IF...ELSE...ENDIF** 那样到处都是。

这篇文章并不准备介绍 **TRY...CATCH...FINALLY** 的基本语法。在本文末尾的引用部分可以找到一些上佳的介绍文章。我们的目标是通过演示一些结构化错误处理的常用模型来增加你对它的熟练程度、扩展对它的使用范围。

捕捉到“期望”的错误

结构化错误处理最基本的、也许也是最重要的用法是：某一小段代码——或许甚至只有一行——中可以预见会出现一个错误，而你想把对这个错误的处理隔离起来。

这里是一个简单的维护函数的例子，它被用于清理一个错误日志：

```
TRY
  USE ErrorLog EXCLUSIVE
  DELETE FOR ErrDate < DATE() - 7
  PACK
CATCH
  * 现在我们只是简单的结束
```

```

FINALLY

    USE IN SELECT("ErrorLog")

ENDTRY

```

注意这段代码的简朴。我们尝试了某些东西，如果尝试失败则优雅的放弃并结束。我们既没有编写盘绕复杂的代码来预先判定文件是否能够被独占的打开，也没有允许一个通用的 **Error** 方法去处理一个更应被单独处理的特殊问题。

现在让我们对这个例子做一点改动。假定我们在局部只想处理一个或两个特定的错误，如果发生了其它错误，我们将把控制权交给更高级别的错误处理器来处理。在前面的例子中，我们准备对付的是（错误日志表）不能被独占地打开的情况（等于 **Visual FoxPro** 的错误号 **3** 或者 **108**），却并没有对文件丢失或者损坏的情况做好准备。这样的话可以象下面这样编写代码：

```

TRY

    USE ErrorLog EXCLUSIVE

    DELETE FOR ErrDate < DATE() - 7

    PACK

    CATCH TO IoExc WHEN INLIST(IoExc.ErrorNo, 3, 108)
    * 这里只捕捉特定的错误

    CATCH && 任何其它的错误

    THROW && 传递错误

FINALLY

    USE IN SELECT("ErrorLog")

ENDTRY

```

请注意这里实现的对不同错误的控制是多么的简单。对于一个简单的 **THROW** 是否是处理意外错误的最佳选择，这里我们先不讨论。等我们谈到 **SYS(2410)**函数的时候再回过头来讨论。

成功/失败 的返回值

把第一个示例简单的改写一下，就可以被用来建立自包含的代码，它简单的返回一个布尔值作为成功或者失败的结果，并自行处理内部的错误：

```

FUNCTION UseExclusive(IcFile, IcAlias)

    LOCAL IIFail

    TRY

        USE (m.IcFile) EXCLUSIVE ;

        ALIAS (EVL(m.IcAlias,JUSTSTEM(m.IcFile)))

```

```

CATCH
    IIFail = .T.
ENDTRY
RETURN NOT m.IIFail
ENDFUNC

```

这个示例还演示了结构化错误处理的另一个优点：它可以被用在用户自定义函数中（UDFs），而不只是类的方法中。

作为功能的错误

对于某些需求来说，结构化错误处理让它们可以把预期的错误当作是自己的功能。在我们 2005 年 1 月 **Foxtalk 2.0** 的文章《用集合设计，第三部分》中，我们演示了怎样建立一个会忽略试图添加重复键值企图的集合类。在那篇文章中，我们的办法是去检查每一个 **Add()** 操作以发现键值是否已经存在。

可以在调用集合的 **Add()** 方法的外部代码中、甚至也可以在 **Add()** 方法本身中使用 **TRY...CATCH...FINALLY** 来作为一个代替的方法：

```

FUNCTION Add(tvItem, tcKey)
    LOCAL loExc
    NODEFAULT
    TRY
        DODEFAULT(m.tvItem, m.tcKey)
    CATCH TO loExc WHEN loExc.ErrorNo = 2062
        * 计划中的“失败”
    ENDTRY
ENDFUNC

```

虽然这种办法很能让人接收，可必须预先警告的是：在 **Visual FoxPro** 中触发错误会对性能产生影响，即使错误是被结构化错误处理捕捉到的也一样。在非正式的测试中，我们发现，要在这个案例中使用结构化错误处理的话，预期错误的出现频率必须相当的低才行。

在抛出好代码之后再抛出坏的

使用结构化错误处理的一个非常好的机会，是在你已经有了一块能够正常运行的重要代码，而又想添加一个新的功能的时候。如果新功能的代码未经过严格的测试或者可能存在着问题，那么，把新代码封装在它自己的 **TRY...CATCH...FINALLY** 结构中将能有助于隔离任何问题，并防止它对应用程序的其它部分产生影响。

举个例子，假设你有一个正常工作的基于 **Web** 的电子商务应用，现在有人通知你，销售部门已经开发了一个 **Web Service**，该应用能够根据客户订单的输入提供实时的付送日期估计，你被告知要把这个信息添加到向用户显示的“感谢”页上。这种类型的新需求总是会成为一些错误的潜在来源。假定你通过简单的在你的控制对象的一个模板方法中添加一个新的步骤来实现这个目标：

```
THIS.ValidateUserInput()
THIS.SaveOrder()
THIS.GetShippingEstimate() && <--被插入的新步骤
THIS.DisplayThankYouMessage()
```

这里的问题是，如果在取得付送估计时出现任何错误，某些更高级别的订单错误处理器将介入进来，而后者通常会导致被向用户显示的是一个错误页面而不是应该出现的感谢页面。由于这个原因，用户也许甚至不清楚订单到底下了没有。

对付象这样的问题有不少结构上的办法，不过，最简单的是象下面这样建立一个安全的环境：

```
THIS.ValidateUserInput()
THIS.SaveOrder()
TRY
    THIS.GetShippingEstimate() && 新步骤
CATCH TO IoExc
    * 在这里执行一些“受控制”的事情
ENDTRY
THIS.DisplayThankYouMessage()
```

撇开你放在 **CATCH** 块中的任何代码不计，即使新方法中有任何的问题，整个系统看起来还是在平滑的运行。然而，**CATCH** 块本身变得重要起来了，因为你必须小心不要把出现的错误忽略的太彻底了以至于根本就没注意到它！可能的行动可以包括将错误记入日志、发送一个 **Email** 报警、甚至把提示信息添加为感谢页的一部分：“你的订单已经被接受。目前我们尚无法提供一个确定的交付日期。”

当然，你在 **CATCH** 块中放的代码越多，代码中就可能存在更多隐藏的错误。这就可能让你试图缓和下来的情况反而变得更糟。下一个部分就是避免这种问题的一些技术。

嵌套的 TRY...CATCH...FINALLY

TRY...CATCH...FINALLY 最有趣的一个部分，是嵌套这个结构的能力，就像可以嵌套 **IF...ELSE...ENDIF** 结构一样。当你第一次接受结构化错误处理的概念的时候，嵌套对你来说也许看起来很难，其实它并非如此。让我们用两个简单的模型来帮助你轻松过渡吧！

复杂的 CATCH

你也许已经从 **ON ERROR** 处理器或者 **ERROR()** 方法中学到了重要的一课：你的错误处理代码必须是完全没有 **bug** 的。不然的话，发生在你的错误处理代码中的错误将会掩盖掉原来的错误，从而导致无数的问题。

结构化错误处理也是如此，虽然程度上要轻一些。尤其是，任何发生在 **CATCH** 或者 **FINALLY** 块中的错误都可能会导致发生掩盖了原错误的问题。这是一个陷阱，不过，处理这类的错误通常可以采取数种操作，而这些操作本身也可能成为错误的来源。例如，如果你想要把错误信息记录到一个日志中去，访问该日志的时候就可能会出现错误。首先，让我们来看一下不用嵌套的一个办法。

```
TRY
DO SomethingThatCanFail
CATCH TO IoExc
WriteToErrorLog(IoExc)
MESSAGEBOX("抱歉，我们现在无法完成任务")
ENDTRY
```

这段代码看起来很直观。然而，如果在访问或者写入到错误日志的时候发生了一个错误，那么用户就永远也看不到后面这个计划向他们显示的消息了。相反，用户看到的将是一个“未处理的结构化意外”错误消息（并且，如果有不管什么捕捉该错误的代码，如果这个代码试图去记录日志，就会导致双重的被掩盖错误！）我们可以使用 **TRY...CATCH...FINALLY** 嵌套来避免这类的问题：

```
TRY
DO SomethingThatCanFail
CATCH TO IoExc
    TRY
        WriteToErrorLog(IoExc)
    CATCH
        * 捕捉并忽略第二个错误
    ENDTRY
MESSAGEBOX("抱歉，我们现在无法完成任务。")
ENDTRY
```

这就好多了，至少用户看到了计划向他们显示的消息了。当然，错误没有被登入日志，所以也许你会想要在 **CATCH** 块内部做些什么事情（不过得小心别又触发了第三个错误！）这里是一个办法，它依赖于你的用户是否被允许看到未经处理的错误消息：

```

TRY
DO SomethingThatCanFail
CATCH TO IoExc
    LOCAL lcErrMsg
    lcErrMsg = "Sorry, we couldn't do that just now."
    TRY
        WriteToErrorLog(IoExc)
    CATCH TO IoExc2
        lcErrMsg = lcErrMsg + " Logging also failed: " + ;
            IoExc2.Message
    ENDTRY
MESSAGEBOX(lcErrMsg)
ENDTRY

```

但愿嵌套现在看起来对你不那么吓人了。现在让我们继续下一个嵌套你的结构化错误处理的更有用的模型。

“失败终结者”方式

使用嵌套的 **TRY...CATCH...FINALLY** 的另一个办法是当处理一个问题有两种途径的时候，如果你较喜欢的方式不可用、会导致一个错误时，你也许会喜欢有机会可以采用作为“失败终结者”的第二种办法而不是取消处理进程。

举个例子，假定你需要自动化 **Microsoft Word** 来处理一个文档。如果 **Word** 已经在运行中了，那么，你可以通过使用 **GETOBJECT** 而不是 **CREATEOBJECT** 来更快速的获得一个对象引用。问题是，很难可靠的判断出当前 **Word** 是否正在运行之中，甚至判断的过程本身就可能会导致产生 **OLE** 错误。这就使得这个案例成为了使用 **TRY...CATCH...FINALLY** 的好地方。此外，你也不能保证用 **CREATEOBJECT** 就一定能成功（因为也许目标机器上甚至可能没有安装过 **Word**）。这个问题可以用嵌套来轻松的处理好：

```

TRY
    IIErr = .F.
    TRY
        IoObj = GETOBJECT(, "Word.Application")
    CATCH TO IoExc && error, so try second best:

```



```

        loObj = CREATEOBJECT("Word.Application")

    ENDTRY

    CATCH TO loExc

        IIError = .T.

    ENDTRY

```

这样一来，或者错误标志被设置了、或者你得到一个对 **Word** 的对象引用。

SYS(2410) 的办法

由于 **Visual FoxPro** 提供了三种不同的错误处理机制，因此我们很高兴能看到在 **VFP 8** 中新增的 **SYS(2410)** 函数。这个函数会返回当前有效的错误处理器的类别。

在研究这篇文章的过程中，我们找不到这个函数任何已有的例子。也许，部分原因在于 **Visual FoxPro** 的帮助文件主题中，后者没有提供示例代码，并且还声明：“你可以在你的 **TRY...CATCH...FINALLY** 代码中使用 **SYS(2410)** 以判定一种操作的途径，例如，通过使用一个 **DO CASE** 结构，根据处理该错误的处理器的类型（来采取不同的操作）。”

这个声明也许会误导你，使你以为这个函数应该使用在捕捉到错误的 **TRY...CATCH...FINALLY** 块中。事实上，这是没有意义的，因为在这种情况下，该函数总是会返回一个 **1!**与此类似，当在一个 **Error()** 方法中的时候，**SYS(2410)**总是会返回 **2**，所以如果只在 **Error()** 方法中使用该函数的话也是没有意义的。

使用 **SYS(2410)** 的正确办法，应该是把它放在你结束捕捉一个错误的地方；就是说，应该放在 **ENDTRY** 后面，同时你仍然拥有关于该错误的信息的时候。请看下面的这个简单的用户自定义函数，在这里，**TRY...CATCH...FINALLY** 被用于处理一个特殊的错误：

```

FUNCTION BadUdf()
    LOCAL IISuccess, loExc

    TRY

        aa == bb && 引入的新 bug

        USE SomeFile && “预期”的错误(在下面处理)

        IISuccess = .T. && 只有不出错的时候才有效

    CATCH TO loExc WHEN loExc.ErrorNo = 1

        * 文件不存在

        IISuccess = .F. && 失败，但是错误被处理了

    ENDTRY

    RETURN m.IISuccess

ENDFUNC

```

这是一个典型的增加了用于预防可预见错误（文件没有找到）的代码的例子。然而，考虑一下当一个对代码的改动引入了一个新的 **Bug** 的时候会发生什么事情。我们漏了这段代码会导致一个“未处理的结构化错误”（错误号 2059），该错误会搞乱并掩盖了原来的错误。所以，现在让我们把 **SYS(2410)** 用起来：

```
FUNCTION BadUdf()
  LOCAL IIUnhandled, IISuccess, IoExc
  TRY
    aa = bb && 引入的新 bug
    USE SomeFile && "期望出现的"错误
    IISuccess = .T. && 只有不出错的时候才有效
  CATCH TO IoExc WHEN IoExc.ErrorNo = 1
    IISuccess = .F. && 失败，但是错误被处理了
  CATCH TO IoExc
    IIUnhandled = .T.
  ENDTRY

  * 等待到已经运行出了 ENDTRY 的范围,然后：
  IF m.IIUnhandled
    IF SYS(2410) = "1" && 有另外一个 try/catch 正等着处理错误
      THROW(m.IoExc)
    ELSE && 做我们力所能及的事情
      ERROR "Error " + TRANS(IoExc.ErrorNo) + ;
        " in line " + TRANS(IoExc.LineNo) + ;
        " of " + PROGRAM() + ": " + IoExc.Message
    ENDIF
  ENDIF
  RETURN m.IISuccess
ENDFUNC
```

现在，如果发生了一个意外的错误，那么这个自定义函数就会试图去判断当前的错误处理器类别是什么，然后将错误抛出，或者触发一个带有足够错误信息的错误。如果你想要在一个你不清楚其内部错误处理方式的特殊环境（例如一个第三方的应用程序框架）中“轻舞飞扬”的话，这种技术可能会非常有价值。与此类似，你也许会试图写一套能够在不同的错误处理环境下都能正常工作的“黑盒子”代码。我们正开始考虑这种有趣的新函数的潜在用途。

注意 **SYS(2410)** 返回的都是字符型值。它很容易在你的错误处理代码中引起“操作符/操作类型不匹配”的错误，这样的话，情况可能反而变得比根本不去捕捉原错误来得更糟。

向后兼容性

这是对那些“由于其产品需要同时能在 **Visual FoxPro 7.0** 和 **8.0** 中操作、从而延误了采用结构化错误处理”的程序员所说的。虽然这会让你的选择受到某些限制，但并非总是如此。有时候，某些编译器指示符能够帮你过关。

```
#DEFINE TRY_CATCH VERSION(5) >= 800

#IF TRY_CATCH
    LOCAL loExc
    TRY
#ENDIF

    SELECT 0
    USE AlreadyInUse EXCLUSIVE
    PACK

#IF TRY_CATCH
    CATCH TO loExc
        MESSAGEBOX("Caught: " + loExc.Message)
    FINALLY
#ENDIF

    USE

#IF TRY_CATCH
    ENDTRY
#ENDIF
```

这也许看起来很难理解，但是通过安排好指示符，对 **Visual FoxPro 7** 来说，我们所拥有的代码是：

```
SELECT 0
USE AlreadyInUse EXCLUSIVE
PACK
USE
```

同时，对 **Visual FoxPro 8** 及其以后版本来说，有效的代码是：

```
LOCAL loExc
```

```
TRY
    SELECT 0
    USE AlreadyInUse EXCLUSIVE
    PACK
CATCH TO IoExc
    MESSAGEBOX("Caught: " + IoExc.Message)
FINALLY
    USE
ENDTRY
```

这种做法很酷，它既能够向后兼容，又在可能的时候利用上了 **TRY...CATCH...FINALLY** 的优点。显然这样的代码写起来比较困难，并且并非总是有效，但在一个高附加价值的情况下，这种技巧还是值得了解的。

监听报表

原著: Doug Hennig

翻译: CY

微软已经在VFP9里开放了报表引擎，它通过新的ReportListener基类来通信。利用ReportListener子类，VFP开发者可以创建他们自己的定制输出。这个月，Doug Hennig思考了ReportListener，并通过示例展示了如何解决现实问题。

正 如我所确信你到现在所知道的，VFP9里得到的最大改进是报表系统。报表设计器和报表引擎（负责运行报表）是得到的戏剧性和激动人心的改进和新的特性。

在VFP9 以前，报表引擎是很单一的：它处理所有的事情—数据处理，对象放置，重绘，预览和打印。VFP9里新的报表引擎在报表和报表引擎间作了职责区分，它仅仅作数据处理和对象放置，而另一个新的VFP基类，ReportListener，它处理重绘和输出。VFP9包括旧的和新的报表引擎，所以你可以在你所认为合适的报表引擎下运行你的报表。微软把新的报表引擎归类于“辅助式对象”报表。

使用辅助式对象报表

你有三种方法来告诉VFP使用新的报表引擎：

- 例化ReportListener（基类或子类），并且把它指定在REPORT 或 LABEL 命令的新的OBJECT子句里。这是最灵活的方法，因为你可以精确指定使用哪一个监听器类，但是它要求你修改你的应用程序里现有的REPORT 和 LABEL命令。

```
loListener = createobject('MyReportListener')  
report form MyReport object MyReportListener
```

- 利用OBJECT TYPE子句指定使用的监听器类型。这里有几个内置的方法：0表示打印，1表示预览，4表示XML输出，5 表示HTML输出。你也可以定义和使用定制类型。

```
report form MyReport object type 1 && preview
```

- 在运行报表前，发出新的SET REPORTBEHAVIOR 90命令；通常，这将会放在你的应用程序靠近开始的位置，于是所有的报表都使用新的报表引擎。在REPORT FORM命令里指定TO PRINTER会导致VFP使用内置的类型0监听器；同样的，PREVIEW关键字将会引用类型1监听器。这明显比其他方法更方便，但是在你例化你自己的监听器时它不会让你获得控制权。

当你以以上两种方法来运行报表时，应用程序将指定新的系统变量_REPORTOUTPUT（默认，ReportOutput.APP是在VFP的根目录下）表示为哪个监听器类将被调用，为指定的类型所例化。ReportOutput.APP是最主要的对象参数，它简单的例化对应的监听器。然而，因为它仅仅是一个VFP应用程序，因此你可以通过设置_REPORTOUTPUT用你自己的应用来代替它。

要确保发布ReportOutput.APP给你的用户，以使得你的应用程序可以使用辅助式对象报表。注意：在非VFP开发环境的运行时模式下，在你的代码里_REPORTOUTPUT必须被准确设置指向ReportOutput.APP或者你自己的代替者。这也同样适用于系统系统变量_REPORTPREVIEW（ReportPreview.APP），如果你的运行程序需要，还有_REPORTBUILDER（ReportBuilder.APP）。还有其他方法来处理这些要求，比如，把ReportOutput.APP代码加入你的项目里。可以在VFP的帮助文件索引“报表输出程序”里找到非常有用的系列标题。

深入ReportListener

因为ReportListener是个VFP基类，你可以把它子类化以实现你所希望的任何报表行为。在你创建你自己的监听器类前，你必须理解它的有效属性，事件和方法（PEMs）。因为篇幅关系，我在这里将仅仅讨论最重要的PEMs；更详细的完整内容请参见VFP帮助文件。

最重要的一个属性是ListenerType。这个属性告诉报表监听器如何来输出。这个属性默认值为-1，它将会无输出。设置为0表示输出到打印机，1表示输出到预览窗口。指定为2或3将会产生有趣的结果：报表在运行，页面在内存里重绘，但是实际却没有输出。你可以在你想控制创建的输出类型时来使用这两个值。比如，当ListenerType为2，VFP将重绘第一页，并调用OutputPage方法，然后重绘下一页和调用OutputPage方法，一直重复。当ListenerType为3，将会在内存里重绘所有的页，但是并不会为每一页自动调用OutputPage方法，允许你按照需要为内存里的页来调用OutputPage方法（并可按任意的顺序）。

在报表真正运行前，报表引擎在私有数据工作期以只读的命名为FRX的游标方式来打开报表的副本。数据工作期的ID将存储于ReportListener的FRXDataSession属性。如果你需要存取报表的数据，CurrentDataSession属性可以告诉你所使用的数据工作期。

CommandClauses property属性包含了一个指针，指向于对包含报表如何运行的属性信息的对象。比如，如果报表被预览，那么它的Preview属性为.T.，如果报表被打印，它的OutputTo属性为1。

当报表运行时报表引擎引发报表监听器的事件。它有部分方法可以在你需要的时来调用。部分重要的事件和方法列表如表1。

表1: ReportListener的部分事件和方法

Event	Description
BeforeReport	在报表运行前引发。
AfterReport	在报表运行后引发。
EvaluateContents	在字段重绘前引发。
AdjustObjectSize	在图像或形重绘前引发。
Render	在每个对象重绘时引发。
OutputPage	调用它来输出指定的页到指定的设备。
CancelReport	调用它来取消报表。

EvaluateContents, AdjustObjectSize, 和 Render是非常有用的, 因为他们可以让你在对象重绘前改变它。另外对于其它的参数(我们将在后面讨论), 这些事件接收当前对象在FRX游标里的记录号。你可以在游标里找到这个记录, 以判断哪个对象将被重绘。

ReportListener

VFP根目录下的FFC子目录包含有一个新的VFP9类库: _ReportListener。这个类库包含数个ReportListener子类。你可以考虑使用这些中的任何一个, 以作为你自己的ReportListener子类, 因为他们比基类增加了一些非常有用的功能。

一个最有用的改进是支持通过后续机制来连接不同的监听器在一起。设置一个监听器的Successor属性指向到另一个, 允许它们在报表处理中互相影响。这意味着你可以编写小的监听器, 它只完成一件事, 让他们按照需要连接起来。IsSuccessor属性告诉你哪一个监听器是领头的(这个与报表引擎通信, 因为它是在REPORT 或 LABEL命令的OBJECT子句里指定的)。

_ReportListener也提供数个实用方法。SetFRXDataSession用于切换到FRX游标的数据工作期。SetCurrentDataSession用于切换到报表数据所在的数据工作期。ResetDataSession用于恢复数据工作期ID给监听器。现在你有了报表监听器的后台, 该是做实练的时候了。

动态格式化

我所想到的第一件事是利用监听器来动态格式化字段。我确信在这以前你遇到这样的事: 你的客户想要在某些条件下把一个字段打印为红色, 在其他情况下为黑色。你在以前的VFP版本里可以这样做, 为某些字段创建两个副本, 一个为红色另一个为黑色, 设为互斥的Print When条件(比如, AMOUNT >= 100 和 AMOUNT < 100), 并且重叠在同一个报表上。当它工作时, 它是需要强制维护, 特别是当你在报表上有许多字段时。

通过报表监听器, 你可以在报表运行时而不仅是在报表设计器里改变字段的格式。这个的关键是EvaluateContents事件, 它在每个对象重绘前引发。这个事件传递当前对象在FRX游标里的记录号, 和指

向一个包含这个字段信息属性的对象的指针（见表2）。

表2: oObjProperties对象属性被传递给EvaluateContents事件。

Property	Type	Description
FillAlpha	N	填充色的字母名、或透明度。取值范围从0（表示透明）到255（表示不透明）。
FillBlue	N	用于填充色的RGB()值的蓝色部分。
FillGreen	N	用于填充色的RGB()值的绿色部分。
FillRed	N	用于填充色的RGB()值的红色部分。
FontName	C	字体
FontSize	N	字号
FontStyle	N	表示字型的值。附加值为1(黑体)，2(斜体)，4(下划线)，128(删除线)。
PenAlpha	N	笔色的字母名。
PenBlue	N	用于笔色的RGB()值的蓝色部分。
PenGreen	N	用于笔色的RGB()值的绿色部分。
PenRed	N	用于笔色的RGB()值的红色部分。
Reload	L	设置为.T.来通知报表引擎，你要更改一个或多个的其它属性。
Text	C	输出到字段对象的文本。
Value	Varies	字段输出的实际值。

DynamicFormatting.PRG包含在伴随的下载文件里，它定义了三个类。DynamicListener定义了动态监听器所必须做的事，还有两个子类，DynamicForeColorListener 和 DynamicStyleListener，用于改变背景色和字型，各个字段的指令分别存储在对应的USER备注字段里（你可以从Field Properties对话框的Other页里存取字段的USER备注）。这个示例的指令如下：

```
*:LISTENER FORECOLOR = ColorExpression
```

```
*:LISTENER STYLE = StyleExpression
```

ColorExpression是一个用于计算RGB值的表达式，比如，IIF(AMOUNT > 50, RGB(255, 0, 0), RGB(0, 0, 0))，它意味着如果金额大于50为红色，否则为黑色。StyleExpression是一个计算有效字型值的表达式（参见表2的FontStyle属性），比如，IIF(AMOUNT > 50, 1, 0)，它意味着如果金额大于50为黑体，否则为正常。

监听器必须要做的第一件事是识别哪一个字段有指令。DynamicListener是在BeforeReport方法里来做的，而不是每次字段计算时都做。它通过调用SetFRXDataSession来选择FRX游标的数据工作期，然后遍历游标，以寻找有对应指令的记录（在.cDirective属性里指定的）的USER备注字段，并把跟随在记录元素的指令表达式传送进数组。

因为DynamicListener继承了_ReportListener的后续特性（它自动调用所有后续监听器的BeforeReport方法），DynamicListener的每一个子类都将有它自己关于与.cDirective 属性相匹配的FRX记录的aRecords数组。于是，每一个监听器都可以很容易判断在报表运行哪一个字段要被激活。

下一个任务是按需应用指令。EvaluateContents检查当前字段的数组元素是否有表达式，如果有就计算。然后它调用ApplyDirective方法，它是从DynamicListener里提取的，但是在两个子类里实现的。比如，DynamicForeColorListener设置toObjProperties对象适当的色彩属性，并且设置它的Reload属性为.T.，以使得报表引擎可以知道字段的格式已经被改变。最后，由于_ReportListener类的基类的EvaluateContents方法并没有处理后续者，代码就调用它的后续者的EvaluateContents方法，如果有。

这时还有另一个琐事：确保ListenerType设置正确。它的默认值为-1，不产生输出，并且在 REPORT 或 LABEL 命令里指定PREVIEW 或 TO PRINTER 不变。于是，如果有需要，LoadReport将设置ListenerType为适当的值。这里是这些类的代码：

```
define class DynamicListener as _ReportListener of ;
home() + 'ffc\_ReportListener.vcx'
dimension aRecords[1]
&& an array of information for each record in the
&& FRX
cDirective = "
&& the directive we're expecting to find
```

* 如果ListenerType还没有被设置，那么就根据它是被打印或是预览来设置它。

```
function LoadReport
with This
do case
case .ListenerType <> -1
case .CommandClauses.Preview
.ListenerType = 1
case .CommandClauses.OutputTo = 1
.ListenerType = 0
endcase
endwith
dodefault()
endfunc
```

- * 在我们运行报表前，我们遍历FRX游标并把任何字段的我们所期望的
- * 在USER备注里的指令信息存储到aRecord数组里。

```
function BeforeReport
dodefault()
with This
.SetFRXDataSession()
dimension .aRecords[reccount()]
scan for .cDirective $ USER
.aRecords[recno()] = strextract(USER, ;
.cDirective + ' =', chr(13), 1, 3)
endscan for .cDirective $ USER
.ResetDataSession()
endwith
endfunc
```

- * 如果有指令的字段要被重绘，就应用它。

```
function EvaluateContents(tnFRXRecno, toObjProperties)
local lcExpression, ;
luValue
with This
lcExpression = .aRecords[tnFRXRecno]
if not empty(lcExpression)
luValue = evaluate(lcExpression)
.ApplyDirective(tnFRXRecno, ;
toObjProperties, luValue)
endif not empty(lcExpression)
```

- * 如果我们有后续者，就让它继续。

```
if vartype(.Successor) = 'O'
.Successor.EvaluateContents(tnFRXRecno, ;
toObjProperties)
endif vartype(.Successor) = 'O'
endwith
```

endfunc

* 提取方法来应用指令。

```
function ApplyDirective(tnFRXRecno, ;  
toObjProperties, tuValue)  
endfunc  
enddefine
```

```
define class DynamicForeColorListener ;  
as DynamicListener  
cDirective = '*:LISTENER FORECOLOR'
```

* 应用指令。

```
function ApplyDirective(tnFRXRecno, ;  
toObjProperties, tuValue)  
local InPenRed, ;  
InPenGreen, ;  
InPenBlue  
if vartype(tuValue) = 'N'  
InPenRed = bitand(tuValue, 0x0000FF)  
InPenGreen = bitrshift(bitand(tuValue, ;  
0x00FF00), 8)  
InPenBlue = bitrshift(bitand(tuValue, ;  
0xFF0000), 16)  
with toObjProperties  
if .PenRed <> InPenRed or ;  
.PenGreen <> InPenGreen or ;  
.PenBlue <> InPenBlue  
.PenRed = InPenRed  
.PenGreen = InPenGreen  
.PenBlue = InPenBlue  
.Reload = .T.  
endif .PenRed <> InPenRed ...  
endwith
```

```

endif vartype(tuValue) = 'N'
endfunc
enddefine

define class DynamicStyleListener as DynamicListener
cDirective = '*:LISTENER STYLE'

```

* 应用指令。

```

function ApplyDirective(tnFRXRecno, ;
toObjProperties, tuValue)
if vartype(tuValue) = 'N'
toObjProperties.FontStyle = tuValue
toObjProperties.Reload = .T.
endif vartype(lnStyle) = 'N'
endfunc
enddefine

```

TestDynamicFormatting.PRG举例说明了如何把用于同一个报表的监听器连接到一起。

```

use _samples + 'Northwind\Orders'
loListener = newobject('DynamicForeColorListener', ;
'DynamicFormatting.prg')
loListener.Successor = ;
newobject('DynamicStyleListener', ;
'DynamicFormatting.prg')
report form TestDynamicFormatting.FRX preview ;
object loListener

```

图1展示了这个程序运行的结果。在Shipped Date里部分记录显示为红色，其他的为黑色。这是因为它在USER备注里有跟随的指令。

```

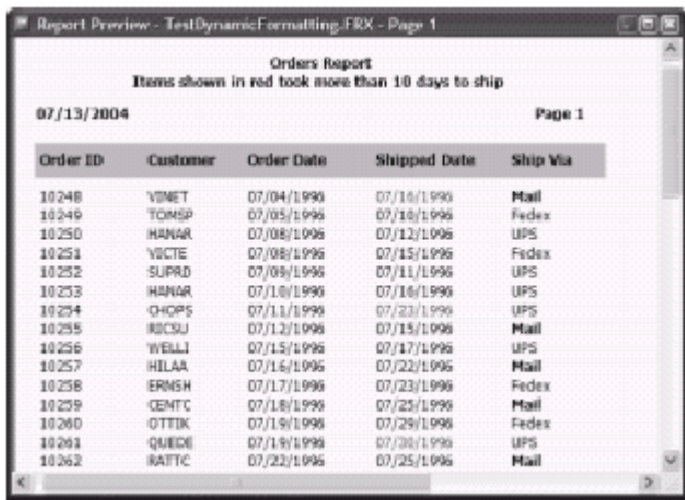
*:LISTENER FORECOLOR = iif(SHIPPEDDATE > ORDERDATE +
10, rgb(255, 0, 0), rgb(0, 0, 0))

```

Ship Via字段有时显示为黑体，有的为正常，这是因为它在USER备注里有跟随的指令。

```
*:LISTENER STYLE = iif(SHIPVIA = 3, 1, 0)
```

（注意虽然这个字段为数字，但它显示为“Fedex,” “UPS,” 和 “Mail”，是因为字段里有表达式。）



Order ID	Customer	Order Date	Shipped Date	Ship Via
10248	YONET	07/04/1996	07/16/1996	Mail
10249	TOMAS	07/05/1996	07/16/1996	Fedex
10250	HANAR	07/08/1996	07/12/1996	UPS
10251	YICTE	07/08/1996	07/15/1996	Fedex
10252	SUPRD	07/08/1996	07/11/1996	UPS
10253	HANAR	07/10/1996	07/16/1996	UPS
10254	CHOPS	07/11/1996	07/23/1996	UPS
10255	RICSU	07/12/1996	07/15/1996	Mail
10256	WELLI	07/15/1996	07/17/1996	UPS
10257	HILAA	07/16/1996	07/22/1996	Mail
10258	ERMNH	07/17/1996	07/23/1996	Fedex
10259	CEMTC	07/18/1996	07/25/1996	Mail
10260	OTTEK	07/19/1996	07/29/1996	Fedex
10261	QUICK	07/19/1996	07/26/1996	UPS
10262	RATTC	07/22/1996	07/25/1996	Mail

图1：报表监听器可以动态格式化字段的文本。

我们还能做什么？

可以是你所知道的任何事。在今后的文章里，我将为你展示其他的监听器，它可以输出到图像文件，旋转的标签，通过目录表输出到HTML，并且有许多其它的输出类型。

VFP领袖Ed Leafe已经创建一个网站 (<http://reportlistener.com>)，以提供作为报表监听器类的中心存储库。那里已经有数个监听器示例，当VFP开发者开始领会他们能够对报表监听器可以做的最酷的事，以后也将会有更多的示例上载。

总结

微软已经开始以许多方法来刮去VFP9扩展性的限制，这包括报表引擎。因为它是个基类，你可以对它子类化，ReportListener可以让创建你自己的定制输出。如果你对监听器有任何的主意或创建了任何很酷的监听器，请告诉我。

来自 VFP 开发团队的 TIPS

原著：微软 VFP 开发团队

翻译：LQL.NET

来自 VFP 开发团队的本月的 TIPS 聚焦在 VFP9 SQL 引擎的速度提升上。VFP9 已经于 2004 年 12 月发布并向 MSDN 订户提供下载。

梦幻般地提升 MIN/MAX 运算速度

分别在 VFP8 和 VFP9 下运行下面的样例（附带的下载文件中）你可以看到 SQL SELECT 语句中 MIN 和 MAX 性能的提升。在同一台机器上测试，这段代码 VFP9 上运行要快 100 倍！

```
* SQL Speed VFP9 vs. VFP8
* Demonstrates improvements to MIN/MAX and
*   correlated sub-query optimization
SET COLLATE TO "MACHINE"
SET TALK OFF
SET ESCAPE OFF
SET NOTIFY OFF
SET NOTIFY CURSOR OFF
CLEAR

?VERSION()

IF .T. OR !FILE("tmp.dbf")
CREATE TABLE tmp (nNum I, dDate D, nCode I)
* ?"Table is created"
FOR ln1 = 1 TO 1000
  FOR ln2 = 1 TO 60
    INSERT INTO tmp (nNum, dDate, nCode) VALUES ;
      (m.ln1, DATE() - m.ln2, ;
```

```

        IIF(m.ln2 > 30, m.ln2 - 30, m.ln2 + 30))
    ENDFOR
ENDFOR
* ?"Records are added"
    INDEX ON nNum TAG nNum
    INDEX ON dDate TAG dDate
* ?"Indexes are created"

ENDIF

CLOSE TABLES ALL
lnSec = SECONDS()
SELECT * FROM tmp t1 ;
    WHERE t1.dDate IN ;
        (SELECT MAX(t2.dDate) ;
            FROM tmp t2 ;
            WHERE t2.nNum = t1.nNum) ;
INTO CURSOR xxx
lnSec1 = SECONDS()
? "Time ", m.lnSec1 - m.lnSec, "records ", ;
    RECCOUNT("xxx")
CLOSE TABLES ALL
SET TALK ON
SET ESCAPE ON
SET NOTIFY ON
SET NOTIFY CURSOR ON
ERASE tmp.*
RETURN

```

VFP9 的 DELETE 快速删除记录

下面的代码展示了 DELETE 新的能力。

```

* Demonstrate correlated delete.
* (DELETE ... FROM ... WHERE...)
* DELETE records for items marked discontinued

```

```

* in the MFG_MSRP table
CLOSE DATABASES ALL
OPEN DATABASE Northwind
CLEAR
=SYS(3054, 0)
*=SYS(3054, 12)
?"* Scenario 1, loop through mfg table, " + ;
    "DELETE FOR matching records"
lnStart = SECONDS()
USE Products IN 0
USE Mfg_msrp IN 0
SELECT Mfg_msrp
SCAN FOR discontinu
    SELECT Products
    DELETE FOR Products.ProductID = mfg_msrp.ProductID
    SELECT mfg_msrp
ENDSCAN
SELECT Products
COUNT FOR DELETED() TO nCount
? nCount, 'records deleted in ', SECONDS() - m.lnStart
RECALL ALL
CLOSE TABLES ALL
?"* Scenario 2, correlated DELETE"
lnStart = SECONDS()
DELETE products ;
    FROM mfg_msrp ;
    WHERE mfg_msrp.productID = products.productID;
    AND mfg_msrp.discontinuu = .t.
SELECT Products
COUNT FOR DELETED() TO nCount
? nCount, 'records deleted in ', SECONDS() - m.lnStart
RECALL ALL
CLOSE DATABASES ALL
RETURN
*例程文件：502TEAMTIPS.ZIP

```


按 Pixel 对文本进行自动换行和段落重排

原著: Pradip Acharya

翻译: Fbilo

偶尔你会碰到需要根据字符集的情况、对文本进行精确到 **pixel** 宽度的自动换行和段落重排。**VFP** 自己内部带有这个功能，例如在报表的溢出时自动扩展选项、在表单标签上、以及在缩放备注窗口的时候。除了已经过时并且无效的 **Memowidth** 设置以外，这个功能对于程序员来说是被隐藏而无法使用的。**Pradip Acharya** 提供的这个 **WrapText** 类是一个高级的精确文本自动换行和段落操作的工具。

你 何时会碰到需要将文本重排到特定 **pixel** 宽度范围内的情况？让我们来看两个例子。第一个，我需要一个单页的维修工作订单上打印一个任务说明。如果说明超过了框子的大小，那么我将需要建立一个独立的多页附件。为了完成这个目标，我需要根据使用的字符集，计算出文本中有多少行、以及在报表表单上框子的大小。

另一种情况，是当我直接从我的 **VFP** 应用程序中在线编程的更新 **Web** 页上交替出现的颜色带区时，我需要能输出一些适当长度的个别的单行文字。**VFP** 会让我做到这一点吗？据我所知，答案是否定的。因此，精确到 **pixel** 宽度的对文本进行智能的自动换行和段落重排的需求就出来了。

```
LineCount = 9
```

```
REFLOW PARAGRAPH TO 350 PIXELS:
```

```
A more powerful way to interact with other applications is to  
use Automation. Using Visual FoxPro programs, you can  
access the objects exposed by other applications, and  
then control them by setting their properties and calling  
their methods. For example, Microsoft Excel exposes an  
application object as well as worksheets, columns, rows,  
and cells within the application object. You can directly  
manipulate any of these objects, including getting or  
setting data in them.
```

```
LINE BY LINE TO 400 PIXELS:
```

```
A more powerful way to interact with other applications is to use  
Automation. Using Visual FoxPro programs, you can access the  
objects exposed by other applications, and then control them by  
setting their properties and calling their methods. For example,  
Microsoft Excel exposes an application object as well as  
worksheets, columns, rows, and cells within the application object.  
You can directly manipulate any of these objects, including getting  
or setting data in them.
```

图 1、WrapText.PRG 的输出

首先，看看演示

让我们先来看看由 **WrapText** 类完成目标的演示。解压并将文件放在你的路径中的某个地方。通过在命令窗口中输入下面的命令来运行 **WrapText.PRG** 程序：

```
WrapText()
```

使用在演示中的段落是从 **VFP** 帮助文件中拿来的。你将在图 1 中看到的输出基于 10 点的 **MS Sans Serif** 字体。

这个演示表演了计算行数、重排整个段落、以及一行一行的输出。400 pixel 宽度的输出含有 8 行，而比它窄的输出则含有 9 行。使用类 **WrapText.VCX** 包含在下载文件中，它不依赖于 **VFP** 的文件。

你可以用 **MemoWidth** 设置来做到同样的事情，是吗？

错！**MemoWidth** 在古老的 **DOS** 年代里服务于某些受限的目的，那时候 **FoxPro** 实际上主要用一个 10 点、固定间隔的 **FoxFont** 来处理所有的字体。**MemoWidth** 是专用于 foxel 而不是 pixel 的，并且被设置为一个在 8~8192 之间的整型值。因此，定义一个精确到 pixels 的文本自动换行宽度是不可能的。

默认的 **Memowidth** 设置是 50。如果你正在缩放一个备注窗口，那么，出现在窗口中的文本将根据输出窗口的字体属性来自动换行到这个宽度。但是，当我用 **MEMLINES** 函数来统计一下行数的话会发生什么事情？问题在于：不清楚用于判定行数的是哪种字符集。它是基于 **VFP SCREEN** 的字符设置呢还是基于一个 10 点的 **FoxFont**？也许会是后者，但这没有什么意义，因为它与我的目标——报表上的文本框或者 **Web** 页——无关。此外，我们能用 **MemoWidth** 设置来完全控制文本换行吗？在命令窗口中输入下面的内容试一下：

```
Set MemoWidth to 50
? Replicate([ABCDEFGH I ], 9)
```

所有 90 个字符都出现在单行中而没有进行任何换行。这是因为系统只对大于 254 个字符的文本字符串进行换行（除非你把鲜为人知的过时系统参数 **_WRAP** 设置成了 **.T.**）。把上面代码中的 9 改成 30，你就会看到自动换行起作用了。结论：对于我们的目的来说，**MemoWidth** 是过时且无用的。

怎样实现 **WrapText** 类

下面是一个典型的建立 **WrapText** 类的实例的例子：

```
Private oWrapText
```

```
set ClassLib to WrapText additive
oWrapText = CreateObject("WrapText", 350, ;
"MS Sans Serif", 10)
```

这个类的 **Init** 方法接收下面 6 个参数：

```
LPARAMETERS p1pixwidth, p2fontname, p3fontsize, ;
p4style, p5margin, p6force
```

- **p1pixwidth**——以 pixel 为单位的宽度（整型）
- **p2fontname**——字体名称（字符串）
- **p3fontsize**——字体大小（整型）
- **p4style**——字体样式（字符——例如“BI”表示粗体、斜体）
- **p5margin**——.T.或者.F.，用于报表输出，稍后解释
- **p6force**——.T.或者.F.，回到前面的断词处，如果为.T.，则强制立即换行。

Reflow（自动重排）方法

在 **Reflow** 中的第一个参数是一个内存变量或者一个备注字段的名称。建议你明确指定一个内存变量或者表别名：

```
m.oWrapText.LineCount("m.TaskDetail")
* 或者
m.oWrapText.LineCount("Product.SafetyMemo")
```

在用内存变量的情况下，请确保该变量没有被声明为 **Local**。**Reflow** 方法会返回被重新格式化过的文本。

LineCount 方法

LineCount 方法的第一个参数是一个内存变量或者一个备注字段的名称。**LineCount** 方法返回在自动换行后输出的行数。

LinesInit 和 NextLine 方法

这两个方法一起工作以一次输出一行。这里是从 **WrapText.PRG** 中拿来的一个示例代码：

```
m.oWrapText.LinesInit("m.TestMemo", 400)
do while m.oWrapText.MoreTodo
    ?SPACE(10), m.oWrapText.NextLine()
EndDo
```

在 **LinesInit** 中的第一个参数是一个内存变量或者一个备注字段的名称。**NextLine** 不需要参数，它返回文本在经过自动换行处理以后的当前行。

其余参数

是传递一个字段的名称、还是传递该字段的内容作为第一个参数，是值得慎重考虑的事情。传递字段的内容的话，我不能保证在运行时一个大的备注字段的多个拷贝在内部不是一样的。而通过传递名称，代码就可以保证不会建立接收到的文本的多种复件。

除了把字段名称作为第一个参数以外，**Reflow** 方法、**LineCount** 方法、**LinesLimit** 方法都有 6 个其它的参数，这些参数的名称与 **Init** 方法接收的 6 个参数的名称是一样的。这个机制使得我们可以很容易的重新设置所有这些参数中的任何一个或全部而不需要去一次又一次的调用 **Init** 方法。不过，由于被忽略的参数会保留它们原来的值，所以，除非你相当确定哪个值应该保留，否则最好小心的明确指定所有的参数。在前面的例子中对 **LinesInit** 的调用把 **pixels** 宽度重新定义为 400。在这里，下面的代码应该会比较安全的：

```
m.oWrapText.LinesInit("m.TestMemo", 400, ;
    "MS Sans Serif", 10, "", .f., .f.)
```

定义换行位置

我们需要知道一个逐步生成的字符串的累计长度。由于不清楚特定不明字体情况下字符之间间隔距离的规则，任何指望把各个字符的宽度加起来就能得出这个长度之和的企图都是过于乐观的。**VFP** 的函数 **TXTWIDTH** 能计算出一个字符串的长度。不过，在一个个的添加字符后再调用这个函数去计算是不现实的。

我用一个二进制的剪切算法来根据字符的数量把字符串剪成两半，调用 **TXTWIDTH**，然后不断地加长或缩短剪切位置后再剪切成两半，一直到换行位置确定下来为止。为了限制迭代的次数，我没有从字符串的最后一个字符开始，而是把 **pixel** 宽度的一半作为起点来开始。

例如，如果在一个段落中有 7500 个字符，而 **pixel** 宽度是 400，那么我要求自动换行点不能超过第 200 个字符。位于 **NextLine** 方法中属于二进制剪切机制的代码段显示如下：

```
local temp, i, j, StringLen, ncut, endi
```

```
* 段落的余下部分
```

```

StringLen = LEN( THIS.OneLine )

* 优化起始位置
endi = MIN(m.StringLen, THIS.HalfPixels)
nCut = m.endi
do while (m.nCut > 0) && 二进制剪切循环
nCut = INT(m.nCut / 2)
DO CASE
    case TXTWIDTH(LEFT(THIS.OneLine, m.endi), ;
        THIS.FontName, ;
        THIS.FontSize, ;
        THIS.FontStyle) < THIS.FoxelWidth
        if m.endi >= m.StringLen
            temp = THIS.OneLine
            THIS.OneLine = ""
            if THIS.MemoPoint < ;
                IIF(THIS.TotLength > 8192, ;
                    THIS.TotLength, ; && 长文本
                    ALEN(THIS.aTextLines)) && 短文本
                THIS.MoreTodo = .t.
            else
                THIS.NoMore && 这是最后一行
            Endif
            return m.temp && 不需要再重算了
        endif
        endi = m.endi + m.nCut && 加长剪切位置
    case m.nCut > 1
        endi = m.endi - m.nCut && 缩短剪切位置
    other && 排除任何超长的部分字符
        nCut = 0
        for i = 2 to m.endi
            endi = m.endi - 1
            if TXTWIDTH(LEFT(THIS.OneLine, m.endi),
                THIS.FontName, ;
                THIS.FontSize, ;
                THIS.FontStyle) < ;

```

```

        THIS.FoxelWidth
        EXIT
    endif
EndFor
EXIT
ENDCASE
EndDo

if m.endi < 1
    endi = 1
endif

```

智能的回到前面的断词处

如果最后的参数 (**p6force**) 为 **.T.**，那么，当抵达换行位置的时候就立即强制换行，即使尾部挂着一个被拆开的词也是一样。某些人可能需要在某种少见的情况下需要这样的功能，例如，在单行街道地址的情况下，如果街道的名称太长，比如象 “**12746 Avenue of Americas**”，你可能甚至会想要保留最后一个词的碎片。

不过，在大多数正常的情况下，如果换行位置出现在一个词的中间，那么换行位置就应该回头放在前一个断词的地方。有一些特殊的情况得考虑到。如果换行位置放在了任何一个 “[({” 之类的字符位置上，那么，下一行就强制以该字符来开头。回头的动作会停止在第一个空格或者特殊字符（例如任何非字母数字）的位置前。还有要注意的就是防止对象 **-12.345** 这样的数字进行拆分，包括符号。

如果没有找到一个有效的断词位置，回头的动作就不会发生。该行就回到原来的换行位置。

报表输出的边界

当我把经过换行处理后的行们交给报表输出的时候，我注意到有时候会出现整个词从被打印出来的文档中消失了的情况。例如，字符串 **INSPECTED VEHICLE,REMOVED AND REPLACED STABILIZER** 在 8 点的 **Arial** 字体中有 **297 pixel** 那么长。当我把输出的文本框宽度设置为 **300 pixels** 的时候，单词 **STABILIZER** 不见了。而当把文本框扩展到 **322 pixel** 的时候，这个文本才真正打印正确。这里有一个 **25 pixels** 的容差。

通过跟踪和调试，我认识到容差是由字体决定的。需要这么大一个容差的理由我们不清楚。这甚至可能是 **VFP** 内部的一个导致过早换行的错误。不管怎样，如果 **p5margin** 参数被设置为 **.T.**，**WrapText** 会从有效的 **pixel** 宽度中减去宽位字符 “**W**” 在当前字体下宽度的 3 倍。这个强制的边界数防止了单词的丢失，它的确定完全是建立在痛苦的跟踪和调试基础上的。

如果你想使用一个固定数量的 `pixel` 作为自定义的边界容差，你将需要把 `p5margin` 设置为 `.T.`，并指定一个减去自定义容差之后的 `pixel` 宽度参数。

短文本和长文本的拆分策略

我的第一个任务，是从输入的文本中取出一个由硬换行分隔的段落。然后我再把这一长行文本用狭窄的目的宽度包装起来。`VFP` 允许我们可以通过两种途径从一个备注字段或者一个内存变量中取得这样的一个分隔的行。

第一种办法是使用 `ALINES()` 函数，它可以接受一个备注字段的全部内容，然后把其中的每一行抽取出来单独放入一个内存变量。这个方法不受 `MemoWidth` 设置的影响并且速度很快。不过，缺点是：即使是一个非常大的备注字段也必须全部被放在内存中。

第二种办法涉及使用 `MLINE` 函数，它一次取回一行，并且依赖于 `Memowidth` 的设置。`WrapText` 类使用 `Memowidth` 可能用到的最大值 (`8192`) 作为一个魔术数字。大于 `8192` 个字符的长文本被通过使用 `MLINE` 策略一行一行的取得。小于 `8192` 字符的短文本用 `ALINES` 函数来拆分成一个个独立的行。

在使用 `MLINE` 函数的长文本策略下，我们面前有一个限制：如果一个单行段落超过了 `8192` 个字符（大约的数字，需根据字体而定），由于 `MLINE` 依赖于 `MemoWidth`，而后者又不会大于 `8192`，所以这个段落将被拆分成多个段落。这不是一个毫无理由的限制，因为在正常的情况下一个超过 `8192` 个字符的段落并不是我们所希望见到的，即使有时是这样，经过拆分也不会有什么数据丢失或者损坏的情况。而优点则是不会出现要求有无限制的内存，并且我们可以处理非常长的备注字段，即使有点慢。

注意，作为一个规则，两个系统变量 `_MLINE` 和 `MEMOWIDTH` 的值总是在每次使用这个类前被保存下来，然后在使用后再对它们进行恢复。在嵌套的操作中，如果一个更高级别的进程正在使用这两个变量而它们的值又被修改过了，那么混乱就会随之而来。

什么是一个行中断？

我们必须识别出在输入的文本中作为一个段落分隔符的行中断。可以把将文本按目的宽度自动换行的过程看作是软行中断。习惯上，`CHR(13) + CHR(10)` 被称作为 `CRLF` 并被解释做一个段落分隔符。不过，这并不重要。`ALINES` 函数会把单个的 `CR` 和 `LF` 或者它们的任何结合体看作是一个有效的段落终结者。

如果我们统一的把 `CRLF` 放在一起作为硬行中断，要自动换行的时候只插入 `CR` 作为一个软中断，就能让处理过的文本与它原来的状态相比好像没有换行过一样。如果没有这么一种措施，那么换行过的文本就没法回到它的原始状态了。`WrapText` 类在终结一行的时候是插入一个 `CR` 作为软中断的。

结论

新的 **WrapText** 类使得“基于字符集对文本进行精确到 **pixel** 宽度的自动换行控制、然后通过传递过时的 **Memowidth** 设置去段落重排”成为了可能。为了优化内存的利用和性能，使用了两个有区别的策略来分别处理短文本和大于 **8192** 个字符的长文本。一个二进制的剪切算法被用来精确的判定换行位置。可以选择在报表输出中的容差和智能回滚到前面的断词位置。统计行数和一次输出一行而不是整个段落也是可能的。

附件：502ACHARYA.ZIP

在缩放中试试这个

原著: Andy Kramek & Marcia Akins

翻译: Fbilo

这个月，Andy Kramek 和 Marcia Akins 公婆俩研究了一下在一个应用程序环境中与缩放表单有关的一些问题。利用 VFP 9.0 的功能，基本的锚定控件任务已经变得非常简单了（参见 Foxtalk 2.0 2004 年 9 月刊专栏文章《The Kit Box: Anchors Aweight》），但锚并不能解决所有的问题。所以当表单缩放的时候，还是经常会碰到有必要使用自己的代码来实现期望结果的情况。问题是怎样去最好的实现这样的代码。

安迪：我有一个关于缩放表单的问题。为什么用户会认为他们需要能够缩放那些我已经小心的为他们设计好了的表单？

玛西亚：我猜可能有两种情况。我自己碰到过的情况是，大多数情况下原因在于字体太小了，让我看起来不舒服。你知道，工作的时候，我最喜欢的字体是 14 点的 **Comic Sans Serif**。

安迪：我会买一份那个字体的——虽然我也可以带上我的阅读眼镜。第二个原因是什么？

玛西亚：某些表单控件，通常特别是 **Grid**、**Listbox** 和 **editbox** 并不总是能在默认的系统解析率下显示出它们全部的内容。这就意味着用户必须使用滚动条或者缩放表单来看到更多的内容。

安迪：你已经击中问题的要害了！在开始换用 **VFP 9.0** 干活以前，我通常会在我的表单上放一个 **resizer** 对象。现在，我更喜欢用新的 **Anchor** 属性取代它来处理缩放的事情。不过，由于 **VFP 9.0** 只通过改变控件的大小来对付第二个问题，所以当缩放一个使用了许多锚定的控件的表单的时候，结果可能看起来相当的愚蠢。看看图 1 中的控件们身上发生了什么事情。

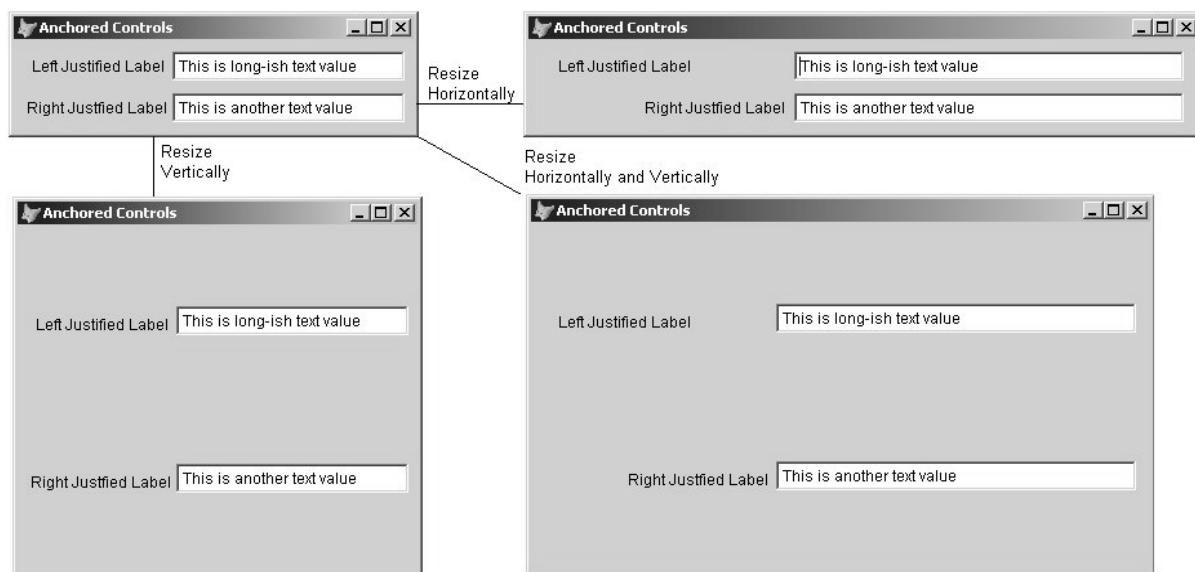


图 1、缩放一个带有一些被锚定了的控件的表单（1）

玛西亚：我同意，那看起来是很蠢。那么你为这些控件使用的锚定设置到底是什么呢？

安迪：事实上，那是一些我们在我们的关于锚的专栏文章中为 **Textbox** 和 **Label** 类推荐使用的设置。我用的值是 **672**，翻译过来就是“垂直固定[512]” + “左边相对[32]” + “右边相对[128]”。

玛西亚：哦，好吧，至少我们不用对付垂直缩放的问题了。既然字体没有改变，那么情况可能会看起来更糟（见图 2）。

安迪：可你看清楚我的麻烦没有？这种做法对于一个 **editbox** 或者 **grid** 来说没问题，可对于 **textbox** 或者 **label** 来说简直就是一场灾难！

玛西亚：这种情况的话，恐怕我们得回到老办法里自己写缩放代码了。

安迪：打住。为什么呀？既然我们可以使用 **Anchor** 属性来处理缩放的问题，我们只要需要在需要的时候去调整字体就可以了。

玛西雅：但我们还是需要去判定尺寸的改动给当前控件造成了什么样的影响的。事实上，我们仍然需要去检测表单是否被缩放过了。既然如此，那么在这种情况下的 **anchor** 还有什么用处呢？

安迪：那么我们就得回头使用在表单上放一个缩放对象的老办法喽？做是可以做到的啦，可让人有点头疼的是：必须等到表单上所有的控件已经被初始化了以后，在表单的 **Init()** 中完成它。

玛西亚：是的，而且我总觉得这种做法根本就是错的。想想这个：我们所拥有的，是一个缩放对象，这个对象会侵入到其所在环境中所有其他对象的私有空间中去操作它们的属性，而后者们又对此一无所知。对我来说，这样一个对象的存在似乎不合于面对对象的原则。

安迪：这明显的破坏了封装的原则，可我们还是需要这么一个对象来管理这个过程呀！

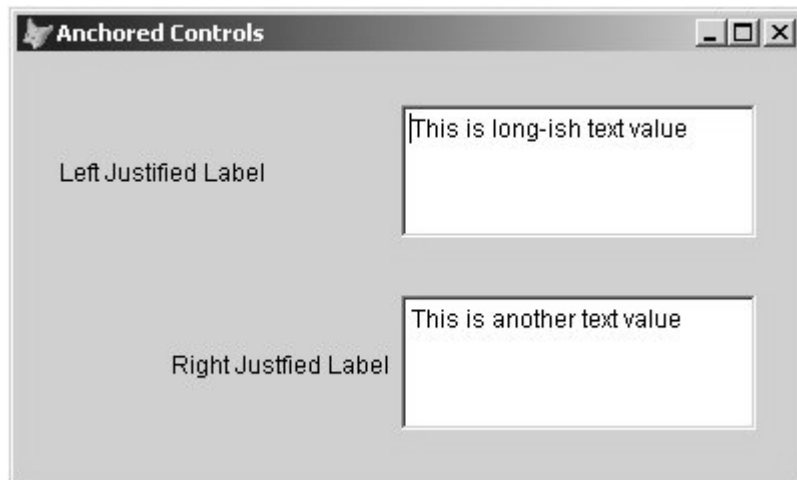


图 2、缩放一个带有锚定了的控件的表单

玛西亚：为什么？如果我们正在缩放一个表单，表单自己当然知道自己正在被缩放。为什么不让它去把这个事实通知给所有包含在它里面的对象们、由这些对象们自己去排列呢？

安迪：嗯，我不清楚这么做行不行。不过这样的话，你不是得在每个基类中写入上“吨”的代码了吗？毕竟，对于所有控件来说，缩放的基本机制本质上都是一样的——所以，在这些基类中的缩放代码将会存在着大量重复的东西，而用前面的一个缩放对象的办法的话，所有的代码都集中在一个类里。

玛西亚：在某些情况下是这样的。例如文本框、标签和命令按钮等等，它们将使用同样的代码，因为我们希望它们展现出同样的行为，而且它们将使用同样的属性来控制它们的大小和位置。然而，当处理编辑框、列表框、**Grid** 的时候，我们需要它们有着各自不同的行为。更糟糕的是，它们各自都有着一套不同的属性必须要被操作。例如，列表框和组合框们有一个 **ColumnWidths** 属性，当 **ColumnCount** 大于 1 的时候，这个属性跟缩放是有关系的。而 **Grid** 则有着一些有自己的宽度的列，如此等等。

安迪：这么做看起来还是有太多的活要干。

玛西亚：建立这么一个体系要干的活的确要比简单的使用一个缩放对象类多。不过，我们只需要在基类中做一次就可以了。多写一点代码来确保我们的类们能被正确的封装难道不值得吗？

安迪：对此我保留意见。那么我们从哪里开始呢？

玛西亚：让我们先从表单类开始吧。正是它对缩放作出反应，并将消息广播给所有包含在它里面的控件的。

安迪：好吧，那么，我们必须知道的第一件事情是表单是否已经被缩放得更大、或者更小了，以及缩放了多少。

玛西亚：请记住，缩放一个表单会直接改动它原来的 **Height** 和 **Width** 属性。因此，准确的说，这两个属性是“当前”的高度和宽度。而我们需要的是表单的初始高度和宽度，以便我们计算出表单的当前面积和初始面积的比数。

安迪：嗯，多谢提醒。

玛西亚：我们需要两个比数：一个，用来告诉控件们表单当前的宽度与初始的宽度有多大的差异，以便它们用同样的比率来调整自己的宽度；而另一个则用来处理高度。

安迪：那么，要做的第一件事情是在表单类上建立必要的属性、并在表单被初始化的时候来生成这些属性的值喽？

玛西亚：没错，或者也可以使用 **AddProperty()** 方法来动态建立这些属性。我给表单基类添加了一个 **Setup** 方法，从 **Init()** 中调用这个方法来完成这个任务。

WITH This

IF .BorderStyle = 3 && 允许缩放

*** 记录初始尺寸

.AddProperty([nOriginalheight], .Height)

.AddProperty([nOriginalwidth], .Width)

.AddProperty([nWidthRatio], 1)

.AddProperty([nHeightRatio], 1)

*** 设置一个最小宽度和高度来防止出错

IF .MinWidth = -1

.MinWidth = .Width / 2

ENDIF

IF .MinHeight = -1

```
.MinHeight = .Height / 2
ENDIF
ENDIF
ENDWITH
```

安迪：我注意到你在这里强制指定了一个最小宽度和高度。为什么？

玛西亚：某些特定的控件——例如 **pageframe**——有一个问题，当它们被缩得太小的时候会出现一个运行时错误。

安迪：所以你设置了一个强制的约束来防止用户们中镖？

玛西亚：没错。出于根据表单所显示的信息来缩放表单的考虑，我选择了表单初始化时大小的一半。不管怎么说，当你把表单缩小到一半的时候，表单上的东西至少还可以辨认得出来。

安迪：好的。那么现在我们需要分别去处理那些控件了。这些控件也需要记录下它们得初始值，以便当需要在表单上成比例的缩放它们的时候，它们能知道从什么值开始缩放。

玛西亚：为了保持接口的统一性，我给所有的可视化控件也添加了一个 **Setup** 方法，并从 **Init()** 中直接调用这个方法。该方法会记录控件的初始大小和位置。基于性能的考虑，我就直接把重要的属性添加到这些类里了。

安迪：好，假定表单上有大量的控件，每个控件都会调用 **AddProperty()** 至少四次（以添加初始的 **Height**、**Width**、**Top** 和 **Left** 属性），这样的话，表单初始化的速度显然会受到影响。所以你这么做（译者注：指在设计时就把属性添加到类里，而不是在运行时调用 **AddProperty()** 来添加）是有意义的。那么，代码会与表单的不同、并且根据不同的类也会出现差别吗？

玛西亚：是的，当然了。我们先来看看一个文本框的代码（毕竟正是这个控件让我们开始考虑本文的内容的）：

```
WITH This
*** 记录尺寸、位置以及字体
.nOriginalheight = .Height
.nOriginalLeft = .Left
.nOriginalTop = .Top
.nOriginalwidth = .Width
.nOriginalFontSize = .FontSize
```

安迪：我猜给标签、命令按钮、复选框的代码将是一样的，因为我们想让它们表现出同样的行为。

玛西亚：是的。被缩放后将会显示更多或更少数据的控件们不需要记录初始的 **FontSize**，因为缩放过程将不会去修改这个属性。所以，在一个列表框中，我们不需要去记录这个属性，而需要记录 **ColumnWidths**。另一方面，一个组合框就两个属性都需要记录下来了。所以，在这个方案里，它们的代码是不同的。

安迪：**Okay**，这样一来，为了表现出正确的行为而记录下一套数据就是每个控件自己的责任了。现在，表单应该怎样将一个控件需要对自己进行缩放的信息告诉该控件呢？

玛西亚：这才是真正“酷”的地方！我给每个类增加了一个 **IResize** 属性，这个属性将可能去对该类本身进行缩放。这个属性有一个相关联的 **assign** 方法（名为 **Iresize_assign**），代码就是放在这个方法中的。现在，表单所需要做的，就是在它自己的 **Resize()** 方法代码中计算出高度和宽度改动的比率：

```
This.nHeightratio = this.Height / this.noriginalheight
This.nwidthratio = this.Width / this.noriginalwidth
```

然后象下面这样用 **SetAll()** 方法来将表单已经被缩放过了的事实进行广播：

```
ThisForm.SetAll( 'IResize', .T. )
```

安迪：干得漂亮！使用 **SetAll()**最大的好处是：**VFP** 会去干检查哪个对象拥有指定属性的工作。更棒的是，它甚至会深入到所有子容器内部去，不像 **Refresh()**，当前不激活的 **page** 就不会刷新。

玛西亚：是的，这样就省略了常见的那种深入到容器内部去所需的递归代码。它还省略了用于根据各个对象的基类来缩放对象的大量 **CASE** 语句。各个控件唯一需要做的就是从表单上取得改动的比率，然后用这些比率来相应地调整它们自己的值。这里是文本框的代码：

```
*** 取得表单的高度和宽度改动比率

lnWidthRatio = Thisform.nWidthRatio
lnHeightRatio = Thisform.nHeightRatio

*** 相应的缩放控件

WITH this
```

```

.Width = .nOriginalWidth * InWidthRatio
.Height = .nOriginalHeight * InHeightRatio
.Top = .nOriginalTop * InHeightRatio
.Left = .nOriginalLeft * InWidthRatio

*** 让表单来选择最佳的字体

Thisform.GetBestFit( This, ;
LEN(ALLTRIM( This.Text )), 6 )

.IResize = m.vNewVal

ENDWITH

```

安迪：看起来相当的直观，除了一件事情——调用表单的 `GetBestFit()` 是怎么回事？你还从来没有提到这个过。

玛西亚：那是我们搞清楚使用什么字体的地方。该方法接收一个对控件的引用、控件中要被显示的文本（可以是控件的 `Caption`、`Value` 或者 `DisplayValue`——根据控件的类型而定）的长度、以及一个可选的偏移量。

安迪：这个偏移量是干嘛用的？

玛西亚：它是用来告诉该方法：当计算字体的大小的时候，允许该控件的 `border` 是多少 `pixel`。现在我来跟你讲清楚这个方法是怎么工作的。检查过了输入的参数以后，我们首先计算在原来使用的字体下一个字符的平均宽度：

```

InOrigCharWidth = FONTMETRIC( 6, toControl.FontName, ;
toControl.nOriginalFontSize )

```

安迪：这是假定控件原来的大小和字体对于控件中将要被显示的字体来说是适当的，我觉得这样应该没问题。可要是控件中是空的、没有显示任何数据呢？

玛西亚：在那种情况下，我们会假定控件的初始大小正好等于将要被填充控件的字符个数。我们根据控件的初始宽度和我们刚刚计算好的字符平均大小来获得这个字符个数。

```

IF EMPTY( tnNumChars )
tnNumChars = (toControl.nOriginalWidth/InOrigCharWidth)
ENDIF

```

安迪：我明白接下去将要干什么了。你是要计算出用于生成被缩放控件的字符平均宽度，然后根据目标字符宽度对初始字符宽度的比率来调整字体。

```
InTgtCharWidth = INT( toControl.width / tnNumChars )  
InNewFontSize = INT( toControl.nOriginalFontSize ;  
* (InTgtCharWidth/InOrigCharWidth) )
```

玛西亚：如果那么简单就好了，不幸得是：问题有点儿复杂。例如，如果你试图将字体设置为小于 4 pixel 或者大于 72 pixel 时将会出现一个错误，所以我们必须对此进行检查：

```
toControl.FontSize = ICASE( InNewFontSize < 4, 4, ;  
InNewFontSize > 72, 72, ;  
InNewFontSize )
```

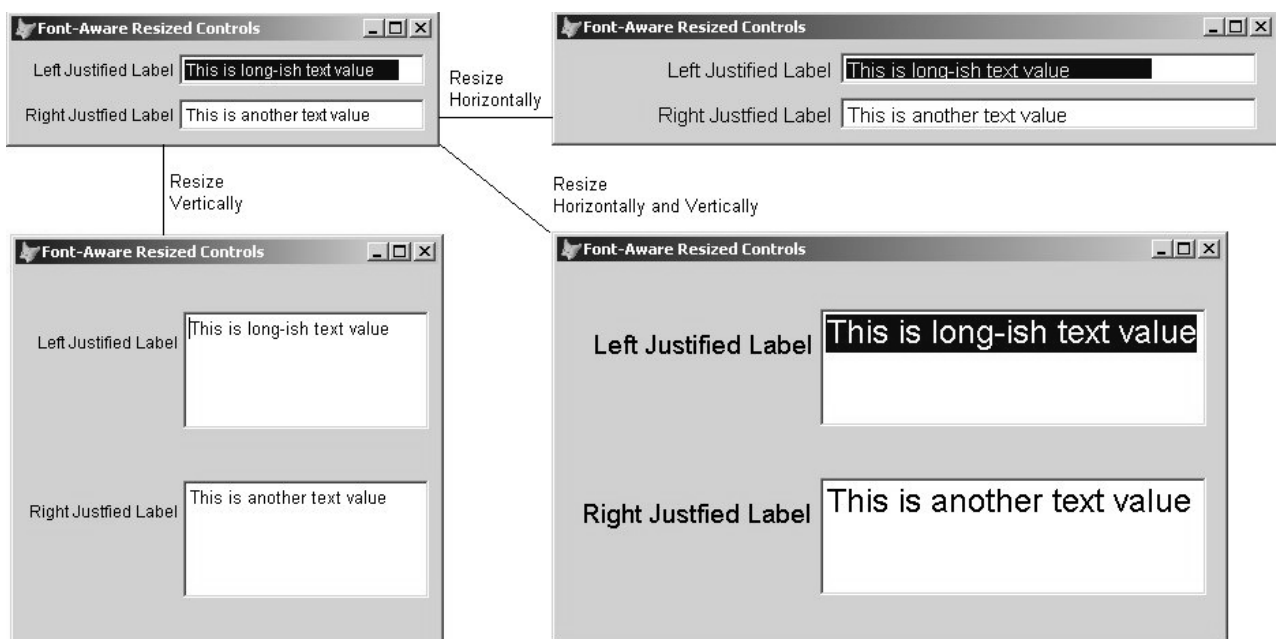


图 3、对字体敏感的缩放

安迪：啊哈！我在这里看到 VFP 9.0 专用的代码啦！在以前的版本中这就无效了。

玛西亚：这是唯一用到版本专用代码的地方，所以，通过把 ICASE() 换成嵌套的 IIF() 语句以让它可以运行在以前的版本中是很轻松的。下一件事情是去调整新的 FontSize 以确保它对于一个控件来说不会太高了。基于这个原因，我们需要再次调用 FontMetric() 函数：

```
WITH toControl  
InMaxFontHeight = ( .Height - tnOffset )
```



```
InSetFontHeight = FONTMETRIC( 1, .FontName, .FontSize )
```

*** I 如果字体太大，就一直减小它到适当的程度

```
DO WHILE InSetFontHeight > InMaxFontHeight
```

```
    IF .FontSize = 4
```

```
        EXIT
```

```
    ENDIF
```

```
    .FontSize = .FontSize - 1
```

```
    InSetFontHeight = FONTMETRIC( 1, .FontName, ;
```

```
    .FontSize )
```

```
ENDDO
```

```
ENDWITH
```

安迪：就这样吗？

玛西亚：Yep。现在这是表单基类的一部分了，所以你总是可以在需要的时候找到它。

安迪：不过它真的总是会比简单的使用锚来说表现得更好吗？

玛西亚：这得由你自己来判断了。图 3 显示了结果。

安迪：嗯，不管是水平的还是全面的缩放，看起来都比使用锚定控件的办法好多了。不过，在只垂直缩放表单的情况下，它看起来还是有点蠢。

玛西亚：我想你可以增加代码来检查是否控件只向单个方向进行缩放。这样的情况下，你可以只调整控件的位置而不对它进行缩放，让我们把这个任务作为练习留给读者们吧！这个专栏的可下载附件中包含有我们已经讨论过的一个表单的示例以及所有的代码。

附件：502KITBOX.ZIP