在应用程序中增加智能感应

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Web site: http://www.stonefield.com
Web site: http://www.stonefieldquery.com
Blog: http://doughennig.blogspot.com

翻译:xinjie

概述

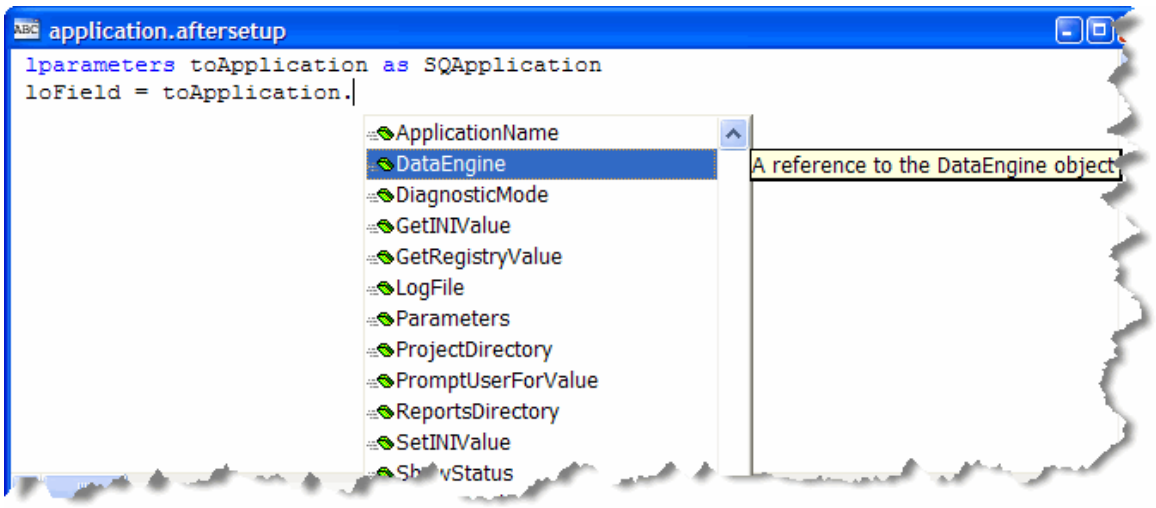
智能感应是 Visual FoxPro 有史以来所增加的最好的特性。与之前所增加的任何特性相比，它为 VFP 开发者提供了强大的生产力。然而，直到 VFP9，它的使用仍然被局限在开发环境中。现在，在运行环境中也可以使用它了！本文描述了为什么说它在一些类型的应用程序中是非常有用的特性，并展示了如何针对你的应用程序来创建一个定制的智能感应环境。

引言

智能感应是 Visual FoxPro 有史以来所增加的最好的特性。与之前所增加的任何特性相比，它为 VFP 开发者提供了强大的生产力。然而，直到 VFP9，它的使用仍然被局限在开发环境中。从 VFP9 开始，智能感应同样在运行环境中受到支持。在我们开始讨论如何做到之前，先来看看为什么要这么做。

最近，我在我的应用程序中提供了一个钩子来允许可以对应用程序进行自定义。例如，在 Stonefield Query 中，用户可以在多处指定代码：在应用程序启动和关闭时、在数据字典被载入后（以便在需要时动态的更改它）、在用户开始工作前、当针对一个查询的数据已经返回但报表还未运行前、等等地方。这样做的目的就是为了灵活性；我不可能针对每个用户的可能需求做出每一种配置，所以，我让用户自己来完成这项工作。显然，这需要用户了解 VFP 语言，但对于开发者、IT 人员和具有天赋的用户而言是没有问题的，尤其是针对一些简单的更改。

在 VFP8 中，使用者可以在适当的代码编辑对话框的键入必要的代码，但是并不能获得智能感应的支持。在 VFP9 中，智能感应不仅仅局限于 VFP 命令和函数，它也可以为 Stonefield Query 对象模型提供支持。**图一**显示了在 VFP9 中输入代码时的截图。注意这个例子中的语法着色和智能感应的成员列表。



***图一** 如果在你的应用程序中允许使用脚本，那么智能感应是很有用的。*

在 FoxPro Advisor 2004 年 9 月的文章 “在 VFP9 应用程序中使用智能感应”一文中，Toni Feltman 讨论了一些方面，它更多的是在运行时刻可智能感应的使用而不是 技术，例如提供类似 Microsoft Office 的自动校正特性以及医疗诊断商业信息的智能感应。

本文就以下两个方面来讨论应用程序中的智能感应：运行时刻智能感应的执行以及针对你的应用程序创建定制的智能感应。

创建运行时刻的智能感应

智能感应是通过运行 _CODESENSE 系统变量指定的程序以及使用 _FOXCODE 系统变量指定的表来运行。默认情况下，在开发环境中，_CODESENSE 指向 VFP 程序目录的 FOXCODE.APP ，_FOXCODE 指向 HOME(7) 中的 FOXCODE.DBF 。在运行环境中，这两个系统变量的默认值都是空。在运行时刻成功提供智能感应的任务之一就是在你的代码中设置这些系统变量并提供一个智能感应程序以及一个表给你的使用者。

尽管我可以将 FOXCODE.APP 打包到我的应用程序中，但是我真的想有一个在本文后面讨论的定制的智能感应程序来支持定制的特性。所以，我考虑如何来创建一个。通过查看 VFP 程序目录下 \XSource\VFPSource\FoxCode 目录(解压 Tools\XSource 目录下的 SXOURCE.ZIP 即可看到)中的 FOXCODE.APP 的源代码，我注意到 FOXCODE.APP 的主程序是 FOXCODE.PRG。我首先想到的是简单的在我的应用程序中包含 FOXCODE.PRG 并设置 _CODESENSE 为 “FOXCODE.PRG”。非常遗憾，我这么做的结果是智能感应并不工作。所以我决定使用一个自定义的 FOXCODE.EXE 来代替它。

为了创建这个 EXE，我首先复制了 Tools\VFPSource\FoxCode 目录下的 FOXCODE.PRG 和 FOXCODE.H（因为我需要对这个 PRG 作出一些更改，我更改复制的副本要比直接将它添加到我的项目中要好的多 ）。接下来，我注释掉了代码中的 DO FORM 语句以及 GetInterface 方法，因为它们会导致项目管理器将我在运行时刻并不需要的大量文件导入到项目中。然后，我创建了一个名为 FOXCODE.PJX 的项目，并将自定义的 FOXCODE.PRG 添加了进去，然后创建了一个简单的 CONFIG.FPW 文件，它只有一行代码：RESOURCE=OFF（当用户无意间运行 FOXCODE.EXE 时可阻止 FOXUSER.DBF 的创建），然后 将 Tools\XSource\VFPSource\FoxCode 目录下的 FOXCODE2.DBF 添加了进来()。最后，我将这个项目编译为 FOXCODE.EXE 。

其中有一件我想做但没有做到的事：在 FOXCODE.PJX 或我的应用程序项目中包含 FOXCODE.DBF 。无论是哪种情况，智能感应总是触发一个我找不到原因的错误。所以，你必须将 FOXCODE.DBF 和 FOXCODE.FPT 作为单独的文件。当然，你也可以重命名它们；同时，简单的设置 _FOXCODE 系统变量指向它们。同样，你将不得不在开发环境中提供智能感应表的副本；你可以编辑这个副本以满足你的需要，例如移除一些在运行环境中确定不需要的命令或函数，或者是在备注字段中增加和对象有关的快捷方式的记录 。

现在我有了一个定制的智能感应应用程序和表，我需要做的就是告诉我的应用程序并设置 _CODESENSE 和 _FOXCODE 为合适的值。这里是 SAMPLE.PJX 的主程序 STARTUP.PRG 的代码。注意，它保存并重置了 _CODESENSE、_FOXCODE、路径和系统菜单。这样做的目的在于如果在开发环境中进行调试，你并不会影响它们在开发环境中的设置。

```
* 创建智能感应

local lcCodeSense, ;
  lcFoxCode, ;
  lcCaption, ;
  lcPath
if version(2) = 2
  lcCodeSense = _codesense
  lcFoxCode   = _foxcode
  lcCaption   = _screen.Caption
  lcPath      = set('PATH')
  set path to SOURCE
  push menu _msysmenu
endif version(2) = 2
_codesense = 'FoxCode.EXE'
_foxcode   = 'FoxCode.DBF'

* 这里执行常规的应用程序

_screen.Caption = 'Sample Application'
do SampleMenu.mpr
read events

* 如果在开发环境中，重置智能感应设置、路径设置、标题以及之前退出的菜单

if version(2) = 2
  pop menu _msysmenu
  _codesense      = lcCodeSense
  _foxcode        = lcFoxCode
  _screen.Caption = lcCaption
  set path to &lcPath
endif version(2) = 2
```

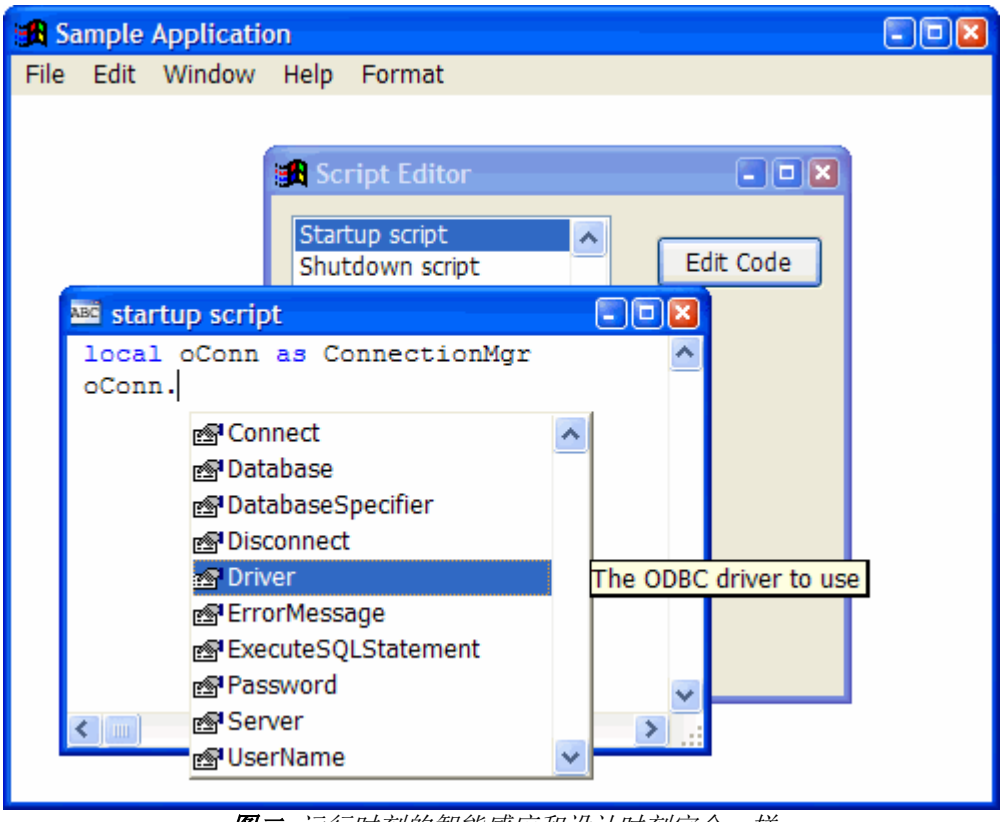
在运行时刻使用智能感应

和 设计时刻一样，在运行时刻，智能感应在两个位置可以起作用：代码(PRG)窗口中和备注字段中。我们已经能够控制这个对象，如果能在编辑框中也支持智能感应就更好了，遗憾的是天不随人愿。需要注意的是，如果想让智能感应在备注字段中工作需要一点儿技巧：你需要确保已关闭了自动换行设置，并打开了语法着色设置，也就是说需要提供保存这些设置的一个自定义 FOXUSER 资源文件。基于此，我更喜欢使用 PRG。即使使用者在一个备注字段中键入了终止符，你仍旧可以将备注字段的内容复制到一个文件中，然后使用 PRG 窗口来编辑它，并将文件内容写回到备注字段。这就是示例应用程序所做的操作。

ScriptEditor.SCX 是示例应用程序中的一个表单，Edit Code 按钮的 Click 事件代码如下：

```
local lcPath, ;
  loForm

* 将 CODE 当前内容写到一个文件中

lcPath = addbs(sys(2023)) + trim(NAME)
strtofile(CODE, lcPath)

* 为 PRG 窗口创建一个典型的表单

loForm = createobject('Form')
with loForm
  .Caption  = trim(NAME)
  .Width    = _screen.Width  - 50
  .Height   = _screen.Height - 50
  .FontName = 'Courier New'
  .FontSize = 10
endwith

* 在 PRG 窗口中编辑代码，然后将结果返回到 CODE

modify command (lcPath) window (loForm.Name)
replace CODE with filetostr(lcPath)
erase (lcPath)
```

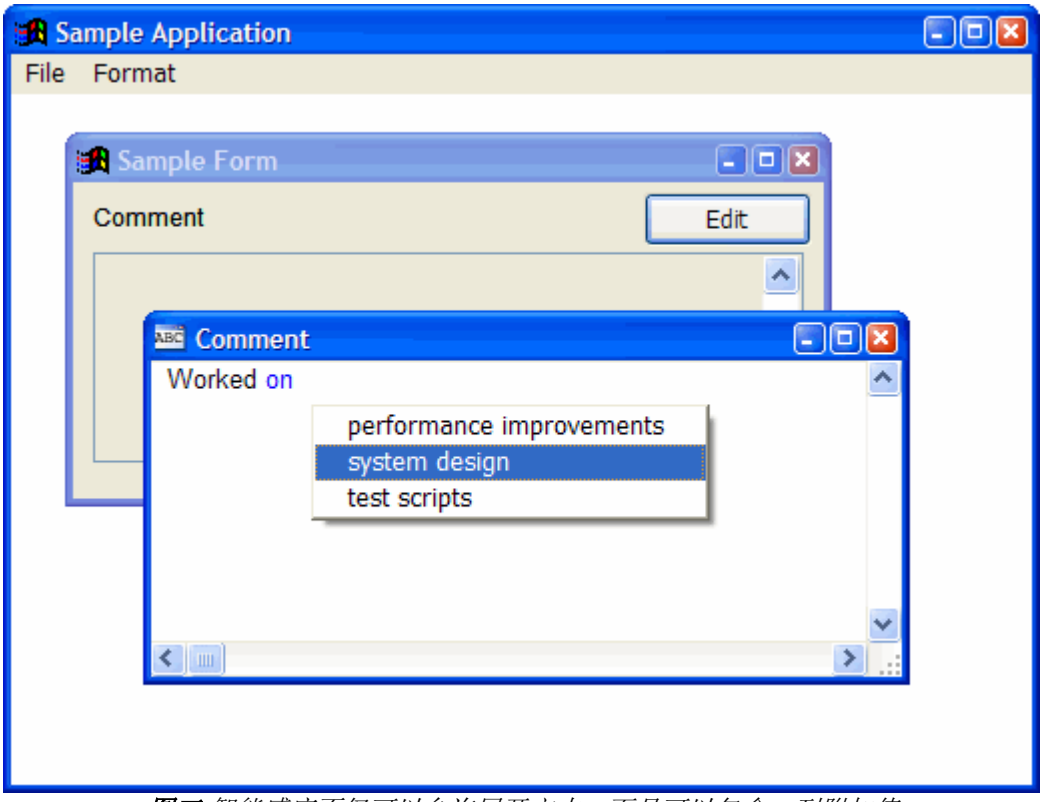
这段代码可以将记录的当前内容写到用户的临时文件目录下的一个 PRG 文件中，通过所创建的代码编辑窗口，我们可以编辑文件，并将 PRG 文件内容写回到表中，然后删除 PRG 文件 。**图二**显示了当运行示例应用程序时智能感应的运行情况（选择一个脚本，然后单击按钮）。



图二 运行时刻的智能感应和设计时刻完全一样

请关注下除字体、窗口大小和位置之外的东西，你对 PRG 窗口不良行为是无法控制的。一个令人讨厌的行为就是当你点击窗口之外的区域时，PRG 窗口会自动关闭（除非你使用 NOWAIT，但这会带来其他的难题）。同时，如果你的应用程序是在一个顶层表单中，也会存在其他的问题。如果你没有使用 MODIFY COMMAND 的 IN WINDOW 子句来指定顶层表单名，并且 _SCREEN = OFF，无论哪种情况，当你使用顶层表单时，代码窗口通常会显的有些不正常。使用 IN WINDOW 可以确保代码窗口作为顶层表单的子窗口，并且不能被移出或调整到超出顶层表单的区域。 所以，我们缺乏对这些窗口的有效控制，我对此有一些 恐惧。

当然，智能感应可以被用于更多情况下的代码编辑；它对于文本表达式同样也是很有用的，就像 Microsoft Word 的自动完成功能那样。例如，在一个银行汇票应用程序中，长长的考勤记录通常包含频繁使用的文字，就像“Worked on”或“Met with”。如果使用者可以键入缩写并自动扩展到所想要的文字，一定是一种很愉快的感受。利用智能感应，这是很容易做到的：创建一条记录，并设置 TYPE = “U”，ABBREV = 所需要的缩写，EXPANDED 为完整的文本。更好的设置是你还可以在 CMD 中指定“{cmdhandler}” 来提供一个可以从弹出菜单选择的列表，以及在 DATA 中指定针对每个列表的值列表。**图三**显示了当用户在 SampleForm.SCX 中编辑代码时，键入”wo”并按空格键之后的状态。

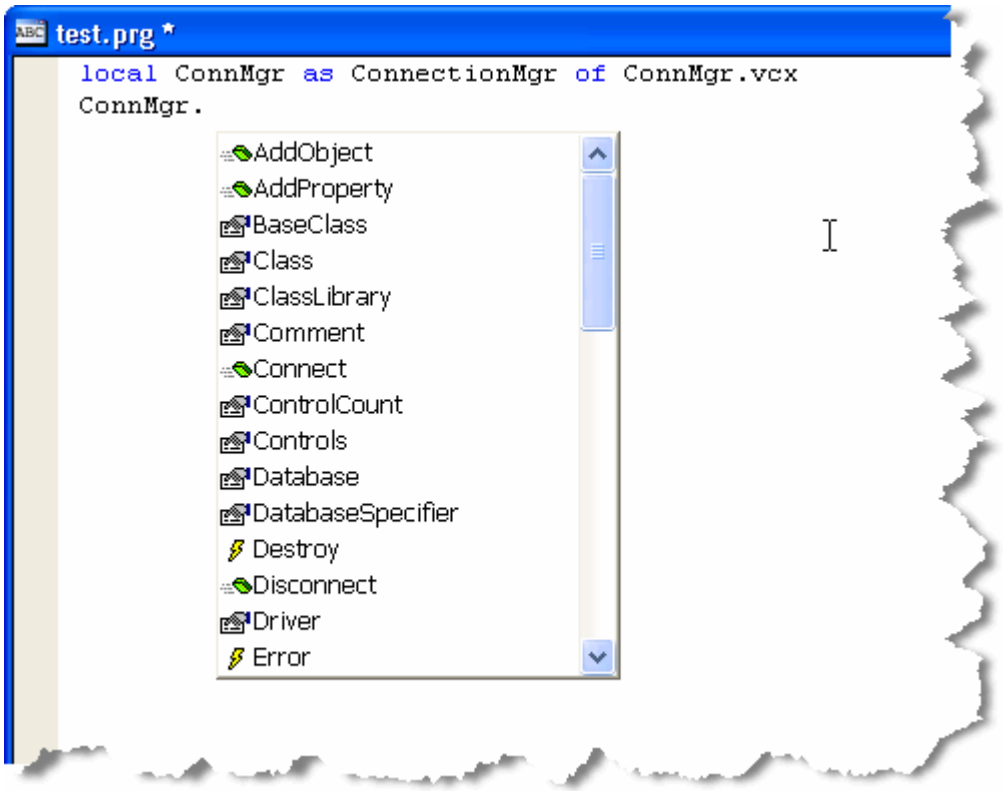


***图三** 智能感应不仅可以允许展开文本，而且可以包含一列附加值*

智能感应的问题

尽管智能感应是伟大的，但是当我在使用一个类时却有一件事时刻的骚扰我，那就是它会显示类的所有成员，而不是我只想看到的有限的几个成员。

例如，**图四**显示了 ConnectionMgr 类的智能感应成员列表。我仅仅对其中几个自定义属性和方法感兴趣，而智能感应却将所有的都显示了出来。如果想达到自己满意，就需要继续努力来，这对于不熟悉类的使用者来说尤其重要。



图四 尽管智能感应允许你从列表中选择成员，但它通常所显示的比我们需要的要多的多。

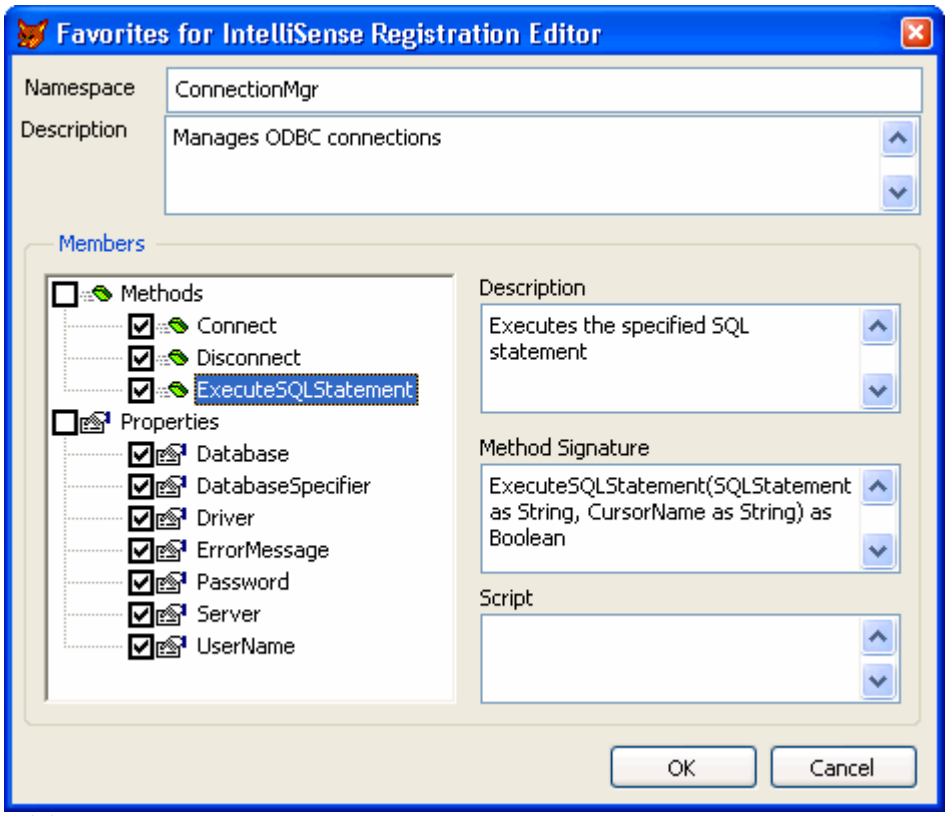
在 VFP9 之前，我们在使用属性窗口时也遇到过类似的问题。在 VFP9 中，微软增加了一个称为“收藏”的标签，在其中可以显示你自己定义的一些成员（一种方式就是右击属性窗口中的成员并选择“增加到收藏”）。所以，现在你不必辛苦的查找你想使用的一个成员，而仅仅在收藏标签下的为数不多的成员中进行查找。

我非常想在智能感应中也使用这样的收藏标签。所以，我决定自己创建一个。也就是 FFI(Favorites tab for IntelliSense)。

受欢迎的智能感应

在我们查看隐藏在引擎下的内容之后，先看看它是如何工作的。我们首先需要做的就是使用 FFI 来注册一个类。打开 ConnMgr.VCX 中的 ConnectionMgr 类（该类库包含在本文档的附带源代码中），然后在命令窗口键入 DO FORM FFIBUILDER 命令。你将看到图五所显示的对话框。这个对话框允许你指定类的“命名空间”。这个命名空间就是当你使用 LOCAL 命令定义变量时 AS 子句后智能感应 所显示的名字。它的默认值为类名，但是你也可以指定为你希望的名字。例如，针对 cApplication 这样的类名，你或许想指定命名空间为 Application。

描述用于针对类（如果这个类被用做另一个类的成员）的智能感应列表中的工具提示信息（例如，示例文件中的 cApplication 类的 User 属性包含 cUser 类的实例，以便类的描述被用于 User 属性）。默认值为通过类菜单的类信息功能所指定的描述或者通过选择项目管理器中类的项目菜单中的编辑描述功能所指定的描述。



图五 FFI 生成器允许你指定针对所选择的类在智能感应起作用时所显示的信息

TreeView 显示了类的自定义属性和方法；如果你想在其中显示类的固有成员，你可以更改 FFIBuilderForm.VCX 中 FFIBuilderForm 类的 LoadTree 方法中的 AMEMBERS() 声明。成员名之前的虚悬则框表示该成员是否显示在智能感应列表中；默认情况下，所有的 自定义成员都将包含。

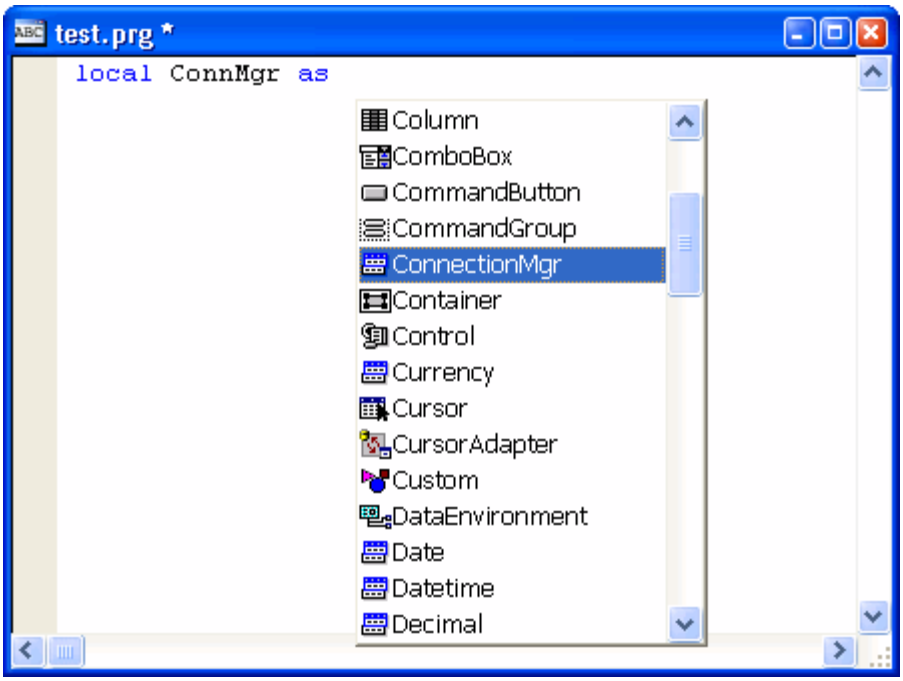
成员描述被用于智能感应列表中成员的描述，其默认值为当你创建这个成员时所键入的描述。方法签名是当你在方法名后键入一个括号或一个句点时针对参数的方法提示。它的默认值为方法名及在方法中所包含的任何 LPARAMETERS 声明，但是你可以编辑它为你所想显示的内容，包括返回值的数据类型。

脚本编辑框允许你输入一些可执行的脚本，这些脚本在你选择这个成员并键入一个括号、句点或等号（为属性赋值）时被执行。后面，我们会看到一个例子。

确信你已经在对话框中完成了所有的更改后，单击 OK 增加和所操作的类对应的记录，它会将已注册的每个成员都增加到 FFI 表中。如果记录增加成功，它也会在智能感应表中增加两条记录：一个是和类对应，一个是和 FFI 的智能感应脚本对应。本文后面会谈论到其中更多的细节。

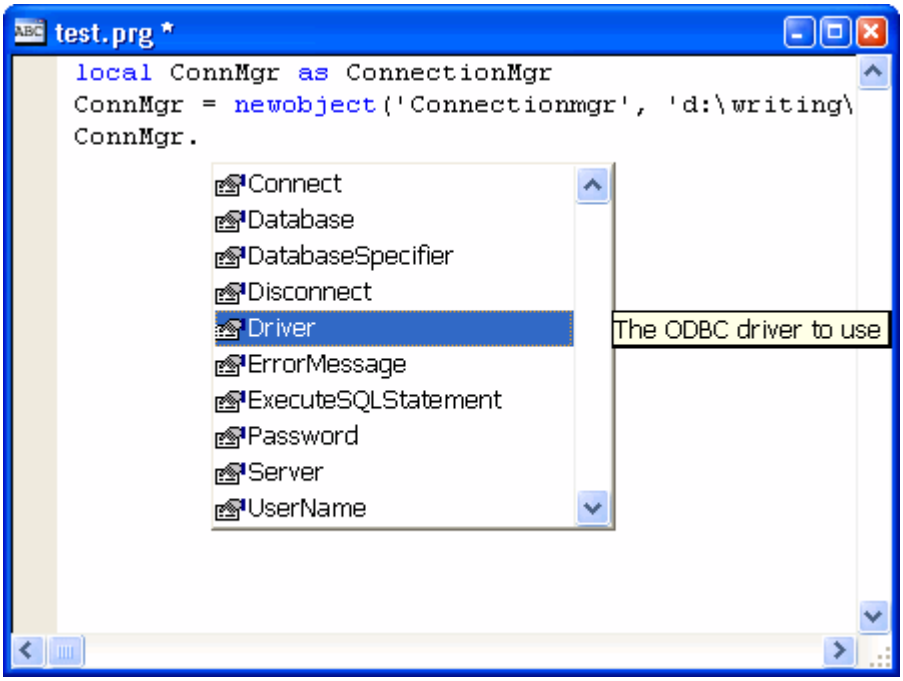
要查看 FFI 针对类是如何工作的，你需要创建一个 PFG 并键入 LOCAL ConnMgr AS。当你在 AS 后键入空格后，你将会看到**图六**所显示的智能感应列表。当你选择 ConnectionMgr 并键入回车时，你将看到下面的代码被写入到 PRG 文件中（Path 是 VCX 所在路径）：

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



图六 任何使用 FFI 注册的类都显示为智能感应中的一个类型

现在，在 ConnMgr 后面键入一个句点。如**图七**所示，智能感应仅显示注册的类成员（这就是 FFI）。如果你选择方法，例如 ExcuteSQLStatement，当你键入括号时，你将看到作为工具提示的方法签名，这可以让你很容易的看到需要向方法传递什么样的参数。



图七 *智能感应仅显示已注册的指定类。请将此图与图四进行对比。*

FFI 是如何工作的

FFI 的秘密其实只有两件事：智能感应如何操作在智能感应表中定义为 “types” 的记录，以及智能感应脚本是如何运行的。Types 通常用于数据类型（例如整数或字符）或者基类（例如 Checkbox 和 Form）。然而，你也可以定义为其他类型，无论是手工，还是通过工具菜单的智能感应管理器来添加 TYPE 被设置为”T” 的记录。

其他类型的记录通常是自定义类或者 COM 对象，这可以允许你得到他们的成员列表。我也是在使用它，但是我们将通过一个脚本和一个智能感应控制类来规定智能感应如何工作。

如果你在使用 FFIBUILDER 表单注册一个类之后查看你的智能感应表 (USE (_FOXCODE) AGAIN 并 BROWSE)，你将在表末尾看到两条新的记录。一个是针对类的类型记录，它其中包含了你所指定的命名空间以及类在智能感应工作时所使用的脚本，CMD 字段指定为 “{HandleFFI}”；另一个是脚本记录，它的 Type 字段值为 “S”，ABBREV 字段值为 “HandleFFI”。

下面的代码是 DATA 备注字段的脚本（代码中 Path1 指放置 FFI.VCX 的目录，Path2 指 FoxCode.EXE 的目录）：

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

define class FoxCodeLoader as FoxCodeScript
  cProxyClass    = 'FFIFoxCode'
  cProxyClasslib = 'Path1\FFI.vcx'
  cProxyEXE      = 'Path2\FoxCode.EXE'

  procedure Main
    local loFoxCode, ;
      luReturn
    if version(2) = 2
      loFoxCode = newobject(This.cProxyClass, This.cProxyClasslib)
    else
      loFoxCode = newobject(This.cProxyClass, ;
        juststem(This.cProxyClasslib), This.cProxyEXE)
    endif version(2) = 2
    if vartype(loFoxCode) = 'O'
      luReturn = loFoxCode.Main(This.oFoxCode)
    else
      luReturn = ''
    endif vartype(loFoxCode) = 'O'
    return luReturn
  endproc
enddefine
```

这个代码中定义了 FoxCodeScript 类的一个子类，FoxCodeScript 类是在由 _CODESENSE 系统变量指定的智能感应程序中指定的。这个子类覆盖了 FFI.VCX 中的 Main 方法（它由智能感应来调用）来实例化 FFIFoxCode 类并调用其 Main 方法。调用传递一个对象引用给智能感应数据对象，它包含了如何使用类型和其他智能感应设置的信息。注意 FFIFoxCode 是在 FoxCode.EXE 或者 VCX 中被实例化；我们并不想在开发环境中使用 FoxCode.EXE，因为它包含不久之后我们需要讨论的 FFI.DBF，在这种情况下，我并不想复制这个表，因为它可能包含过时的记录。

HandleFFI 脚本用于所有使用 FFI 注册的类，所以在表中，它只有一条记录。

作为这个脚本的结果， FFIFoxCode.Main 被所有的已注册的命名空间的智能感应任务所调用，例如，当你键入 “Local 变量名 AS ”并从智能感应列表中选择一个命名空间时，或者当你在包含表示已被实例化的类的变量的语句中键入一个“触发”字符（例如一个句点、一个括号或一个等号）时。

FFIFoxCode

FFIFoxCode 类用来完成针对 FFI 的定制的智能感应，所以我们需要看一下这个类的细节。
类的 Init 方法只做两件事：

- 打开调试（为了更容易的对代码进行调试）
- 通过调用 OpenFFITable 来打开 FFI.DBF (这个表包含所注册的类和它的成员信息)。如果不能打开表，则显示一个错误信息并返回 .F.，阻止类的实例化。

```
* 打开调试

sys(2030, 1)

* 打开 FFI (Favorites for IntelliSense) 表

local llReturn
llReturn = This.OpenFFITable()
if not llReturn
  messagebox('不能打开 FFI.DBF 。', 64, '智能感应控制器')
endif not llReturn
return llReturn
```

从智能感应脚本调用的 Main 方法控制着 FFI 的所有智能感应。正如前面看到的，脚本传递一个 FoxCode 对象到 Main 方法。对象的 Data 属性包含 FoxCode 表中恰当记录的 Data 备注字段内容，该记录对应所注册的类，在这种情况下，Data 包含类所在的命名空间。Main 方法通过调用 GetFFIMember 方法来确定你所输入的类的成员（也可以是类自身），然后从 FFI 表中的适当记录返回 一个 SCATTER NAME 对象。

如果 FoxCode 对象的 MenuItem 属性包含命名空间（如果智能感应通过在智能感应菜单选择一个项目被触发，那么 MenuItem 包含使用者所选择的项目），那么当我们键入了“LOCAL 变量名 AS”语句时，Main 调用 HandleLOCAL 方法来处理这个问题。否则的话，Main 只做有限的几件事。如果 FoxCode 的 MenuItem 属性包含一个值，它是用户从列表中选择的要赋予属性的值，那么 Main 调用 HandlePropertyAssignment 方法以便在代码中插入所选的值（后面我们将看到如何使用它）。如果触发智能感应的字符是句点，那么我们需要显示出已注册的类的成员，所以调用 DisplayMembers 来显示它。如果触发智能感应的是一个括号、等号或者一个逗号，并且在 FFI 表的 SCRIPT 备注字段中保存有代码，那么这个代码将被执行。这样的脚本可以是这样的：为一个属性提供一系列可选的枚举值，或者是方法的参数列表（后面我们将看到一个这样的例子）。

最后，如果 FFI 记录的 TIP 备注字段被填充，Main 就使用它作为智能感应的工具提示。这对于显示方法的签名是很有用的（例如：“Login(UserName as String, Password as String) as Boolean”）。

```
* 这是 FFI 类的智能感应脚本调用的主程序

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

* 从 FFI 表中获取命名空间和一个对象
* 同时，得到触发智能感应的字符

  lcNameSpace = .Data
  loData      = This.GetFFIMember(.UserTyped, lcNameSpace)
  lcReturn    = ''
```

```
  lcTrigger   = right(.FullLine, 1)
  do case

* 使用者从列表中选择了一个值赋予一个属性

    case vartype(loData) <> 'O' and not empty(.MenuItem)
      lcReturn = This.HandlePropertyAssignment(toFoxCode)

* 不能从 FFI 表中找到成员，所以，什么都不做

    case vartype(loData) <> 'O'

* 如果我们使用的是 LOCAL 语句，存储要返回并插入的文本

    case atc(lcNameSpace, .MenuItem) > 0
      lcReturn = This.HandleLOCAL(toFoxCode, lcNameSpace, ;
        trim(loData.Class), trim(loData.Library))

* 如果是通过一个句号触发，显示一个成员列表

    case lcTrigger = '.'
      This.DisplayMembers(toFoxCode, loData)

* 如果是通过"(" 触发(来显示方法参数列表)，或者 "="
* (针对属性)，或者 "," (来键入一个新的参数)，那么我们就执行一个脚本

    case inlist(lcTrigger, '=', '(', ',') and not empty(loData.Script)
      lcReturn = execscript(loData.Script, toFoxCode, loData)

* 如果智能感应通过 "(" 或 "," 触发，显示方法的工具提示（通常是方法签名）

    case inlist(lcTrigger, '(', ',') and not empty(loData.Tip)
      .ValueTip  = loData.Tip
      .ValueType = 'T'
  endcase
endwith
return lcReturn
```

这里我们没有看到 HandleLOCAL 的代码；你可以随时去查看它的代码。当你键入“LOCAL 变量名 AS”时， Main 调用它，并从类型列表中选择一个已注册的命名空间。然后生成一个你必须要有的 NEWOBJECT() 语句。这个方法唯一的问题是判断如何生成这样的语句。

(我并没有将 “newobject” 硬编码到代码中，因为我在使用 VFP 关键字时总是使用小写字母，但是在这方面，其他的开发者或许和我有着不一样的爱好。这个问题可以通过访问智能感应表中 TYPE=”F”(针对”函数”)和 ABBREV=”NEWO”(NEWOBJECT 的缩写)，并更改 CASE 字段值来解决。)

从 Main 中调用的 GetFFIMember ，用来在 FFI 表中查找类或成员。它使用的是 FoxCode 对象的 UserTyped 属性（所传递的第一个参数），它包含你键入的相关的命名空间。例如，当你键入 “llStatus = ConnMgr.ExecuteSQLStatement(”时， UserTyped 包含 “ConnMgr.ExecuteSQLStatement”。

GetFFIMember 先在 FFI 表中找到和指定类匹配的记录。如果在 UserTyped 中包含句点，那么它会查找匹配的成员记录。如果可以找到，它会从记录中返回一个 SCATTER NAME 对象。

```
* 确定用户键入的命名空间的成员并从 FFI 表中的匹配记录中返回一个 SCATTER NAME 对象

lparameters tcUserTyped, ;
  tcNameSpace
local loReturn, ;
  lcUserTyped, ;
  llFound, ;
  lnPos, ;
  lcMember, ;
  lnSelect

* 获取所使用的类型。如果以括号开始，则去掉括号

loReturn    = .NULL.
```

```
lcUserTyped = alltrim(tcUserTyped)
if right(lcUserTyped, 1) = '('
  lcUserTyped = substr(lcUserTyped, len(lcUserTyped) - 1)
endif right(lcUserTyped, 1) = '('

* 在 FFI 表中查找类记录。如果在键入的文本中包含句点，尝试查找成员记录

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

* 如果找到匹配的记录，则创建一个 SCATTER NAME 对象。

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

从 Main 调用的 DisplayMembers 是用来当你在行末键入一个句点时显示一个已注册的类成员 列表。DisplayMembers 调用 GetMembers 来查找指定类的成员集合（这里我们并不关注方法）。然后，它使用成员名及其描述来填充 FoxCode 对象的项目数组，并设置对象的 ValueType 属性为“L”，以这种方式告诉智能感应来显示一个包含项目数组的列表框。

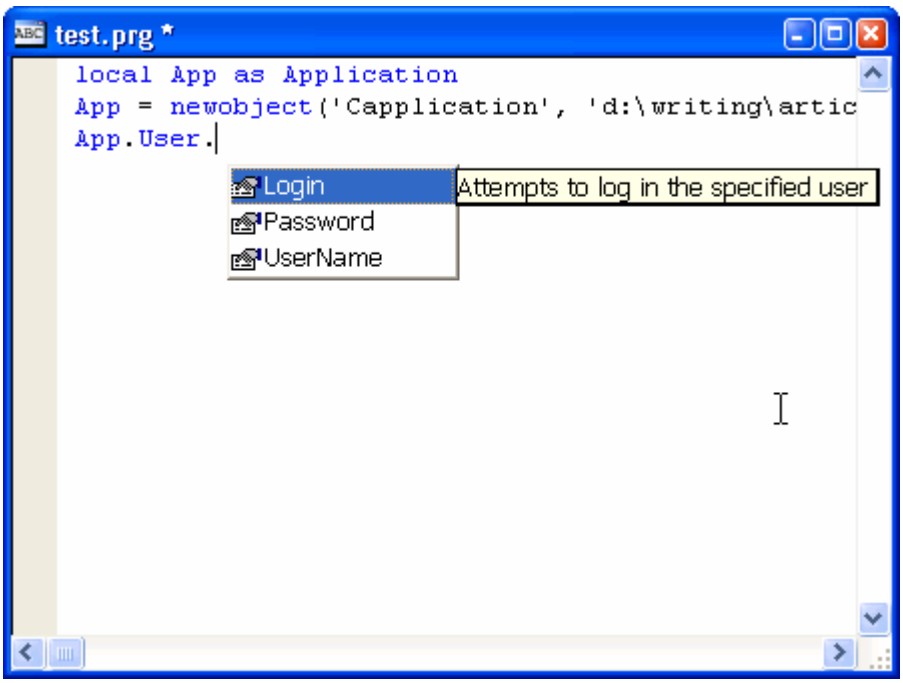
这个代码显示了智能感应的一个设计缺陷：FoxCode 对象具有一个单一的 Icon 属性，它包含要显示在列表框中的图像的名称。也就是说我们所获得的项目数组需要额外的一列，因为在这种情况下，我们想针对方法和属性显示不同的图像。但是很遗憾的是，我们仅仅只能为所有的项目显示单一的图像。

```
* 建立智能感应要显示的 成员列表

lparameters toFoxCode, ;
  toData
local loMembers, ;
  lcPath, ;
  lnI, ;
  loMember
with toFoxCode

* 获取当前类的成员集合

  loMembers = This.GetMembers(alltrim(toData.Member))
  if loMembers.Count > 0

* 把每一个成员增加到 FoxCode 对象的项目数组中

    dimension .Items[loMembers.Count, 2]
    lcPath = iif(file('propty.bmp'), '', home() + 'FFC\Graphics\')
    for lnI = 1 to loMembers.Count
      loMember       = loMembers.Item(lnI)
      .Items[lnI, 1] = loMember.Name
      .Items[lnI, 2] = loMember.Description
      if loMember.Type = 'P'
        .Icon = lcPath + 'propty.bmp'
      else
        .Icon = lcPath + 'method.bmp'
      endif loMember.Type = 'P'
    next loMember

* 设置 FoxCode 对象的 ValueType 属性为 "L"，这意味着显示一个列表框，在项目数组中包含被定义的项目
```

```
    .ValueType = 'L'
  endif loMembers.Count > 0
endwith
```

一些高级的使用方法

有一些令人感兴趣的事可以让你在使用 FFI 时更容易的使用。一个就是类可以包含等级化的对象，通过注册所包含的类为另一个类的成员来使用 FFI，你可以得到很清爽的智能感应。例如，在运行时刻，cApplication 类的 User 属性可以包含 cUser 类的一个实例。通常情况下，你在 User 属性并不能获得智能感应，因为在设计时刻时它并不包含对象。然而，如果你打开 cUser 类并指定它的命名空间为“Application.User”，你就可以在命名空间 Application 中的 User 成员上获得智能感应，正如在**图八**中所显示的那样。

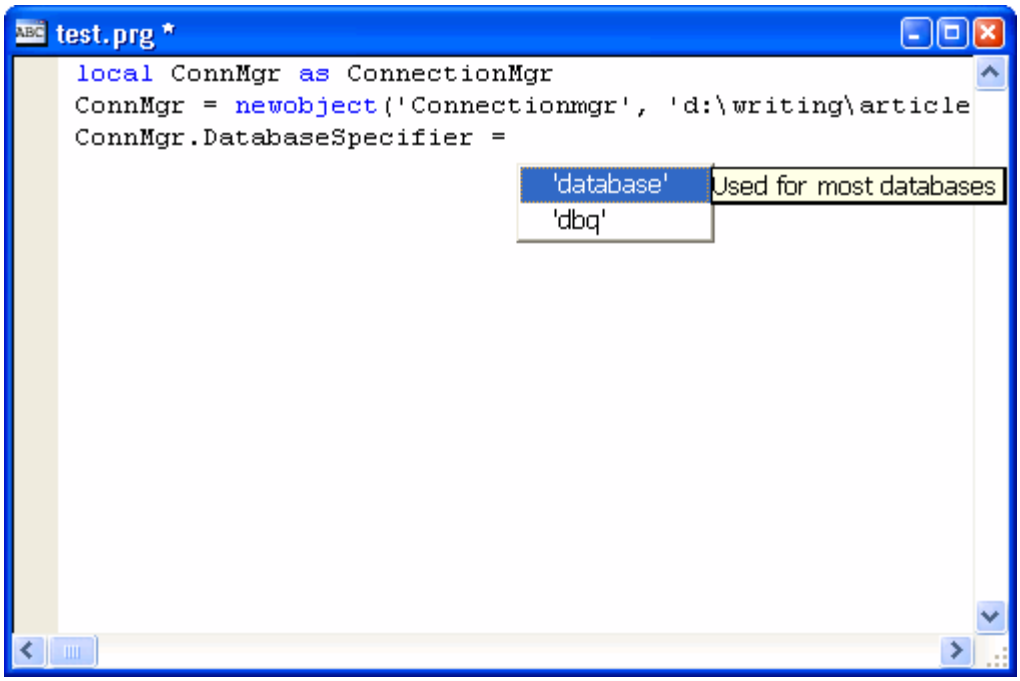


图八 *如果类的命名空间被注册为主命名空间的子命名空间，你甚至可以像运行时刻那样在要实例化的对象上获得智能感应。*

另一个很有用的就是智能感应如何运行在 FFI 生成器表单中的编辑框中键入的和成员有关的脚本。例如，ConnectionMgr 的 DatabaseSpecifier 属性指出了用什么样的 ODBC 字符串连接数据库。对于很你多数据库，例如 ACCESS 和 Dbase，你需要使用 “dbq=database path”来指定数据库，使用 “database=database name”来指定数据库名。

与其告诉开发人员 DatabaseSpecifier 属性的值为 “database” 和 “dbq”，不如让智能感应显示一个可用值的列表来让用户选择（如**图九**所示）。通过填充 FoxCode 对象的项目数组并设置它的 ValueType 属性为”L”，针对 DatabaseSpecifier 属性的脚本就可以执行。

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



图九 *通过使用一个脚本，你可以针对属性赋值显示一个自定义值列表*

注意，所有的脚本要传递一个引用到 FoxCode 对象和 FFI 记录的 SCATTER NAME 对象。同时，这个代码将设置 ItemScript 为 “HandleFFI”；这会告诉智能感应去调用 HandleFFI 脚本，在用户选择一个值后它会调用 FFIFoxCode.Main 。正如前面我们看到的那样，当 FoxCode 对象的 MenuItem 属性包含一个值时，Main 调用 HandlePropertyAssignment，也就是当用户从我们提供的列表中选择一个值时。如果它不是数值，这种方法将增加有引号的值，如果不这样，这个值将不会正常工作。最后需要注意的是，脚本的返回值会传递到智能感应，所以，除非不想在代码中插入值，否则应返回一个空字符串并设置 ValueType 为不是”V” 的值。

下面的代码显示了一个方法中和参数相关的上下文工具提示以及它们的值的列表：

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

更新智能感应程序

为了支持 FFI，你需要将 FFI.VCX 和 FFI.DBF 增加到 FOXCODE.PJX 中。同样，我将用于成员列表的图像文件(PROPTY.BMP 和 METHOD.BMP)也包含了进去。这里发现一个问题，只有将图像文件包含在实际的应用程序项目中(这里是 SAMPLE.PJX)或者作为单独的文件时，在智能感应列表中的成员名字前才会显示对应的图像。

遗留问题

当测试这项技术时我看到了智能感应中的一些 bug，或者说是设计缺陷：

- 在运行时刻，当你键入 “LOCAL 变量名 AS” 时并不会像在设计环境中那样显示一个类型列表。
- 在运行时刻，括号匹配并不会工作。
- 如果你有一个对象是另一个对象的成员（例如这个示例中 cApplication 对象的 User 成员，它包含另一个 cUser 对象的引用），智能感应显示的是父对象的成员。例如，如果你键入 LOCAL loUser as Application.User（cApplication 类在注册时使用的命名空间为 Application），当你键入 “LoUser” 和一个句点时，智能感应列表显示的是 cApplication 类的成员，而不是 cUser 类。

总结

智能感应是我们 VFP 开发者一旦开始使用就会非常喜欢的一个特性。现在，你可以在你的应用程序中为客户提供同样的便利。很显然，这种技术并不是针对所有人，但是如果在你的应用程序中提供了这个特性，我保证你的客户一定也会非常喜欢。在这里要感谢 Randy Brown，他曾经是 Visual FoxPro 的项目经理，他指导我如何使智能感应仅显示所希望显示的成员。

作者传记

Doug Hennig 是 Stonefield Systems Group Inc 的合作伙伴。在 Stonefield Software Inc，他是 Stonefield Database Toolkit(SDT) 的作者。 Stonefield Query 以及 VFP 中的成员数据编辑器、锚定生成器、CA 和数据环境生成器的作者。Doug 是“What’s New in Visual FoxPro” 系列图书的合著者之一（最新的一本是“What’s New in Nine”）以及 “The Hacker’s Guide to Visual FoxPro 7.0” 一书的合著者之一。同时是 “The Hacker’s Guide to Visual FoxPro 6.0” 和 “The Fundamentals” 的技术编辑。这些图书都是由 Hentzenwerke 出版发行 (http://www.hentzenwerke.com)。 Doug 以前为 FoxTalk 的 “Reusable Tools”栏目每月提供文章。1997 年之后的每界微软 FoxPro 开发者会议（DevCon）及世界用户和开发会议都发表过演讲。他还是 VFPX 的领导者之一 (http://www.codeplex.com/Wiki/View/aspx?ProjectName=VFPX)。 从 1996 年至今，他一直是微软的 MVP。