从零开始学习 X#

Eric Selje
Salty Dog Solutions, LLC
@EricSelje
www.SaltyDogLLC.Com
eric@saltydogllc.com
翻译：xinjie
QQ:411618689

X# 备受关注，你不知道如何入门？本白皮书将引导您构建自己的第一个 X# 应用程序。我们将一个示例 FoxPro 程序逐步转换为 X#，并演示如何将我们现有的 VFP 技能转移到 X# 的范例中。

您将学习到：

- 如何迈出 X# 的第一步
- 如何在 X# 中访问 DBF 文件
- 如何在 X# 中使用类、表单等

简介

在 2019 年的 Southwest Fox 会议上，我介绍了 X#，涵盖了 X# 的起源、发展以及截止到那时的状态。截止目前，X# 已经是非常稳定和成熟的产品，并且完全能够创建复杂的以数据为中心的 Windows 应用程序，或者基于 Web 的 ASP.NET 应用程序中间件。如果您还没有阅读那届会议的白皮书，我建议您阅读一下，你可以在 http://saltydogllc.com/wp-content/uploads/SELJE-Look-at-X-Sharp.pdf 获得它。完整的阅读它将为您奠定良好的基础。

X# 已经开发很多年了，但是 X# 更新中令人兴奋的无异于它对 Visual FoxPro 方言的支持。这种支持使得从 Visual FoxPro 开发人员的角度来学习 X# 变得很容易，例如，一旦您了解意大利语就可以学习西班牙语（这是一个猜测-我自己都不知道😊）,它们有很多的共同点和相同的认知，因此您应该能够延续您已有的技能到仍然受支持的产品中，并且，您还可以利用 .NET Framework 而不是陈旧的 Win32 类。

在本次会议中，我遇到的最大困难不是语言本身，而是 Visual FoxPro 和 Visual Studio 开发环境之间的差异。如果您有过使用 Visual Studio 的经验，那么您就可以克服这个最大的障碍。当然，如果您有在 Visual Studio 中开发 C# 应用程序的经验，那么您可能会发现 X# 非常容易使用。

此外：如果您想知道“如果我是一位经验丰富的 C# 开发人员，为什么还要学习 X#”问题的答案，我可以告诉您，那是因为 X# 将处理 DBF 的功能内置于该语言中。您可以在 X# 中创建现有 C# 类引用的数据处理类。.

让我们开始吧！在本节中，我们将从已知的内容开始——我自己创建的一个 Visual FoxPro 应用程序。它不是一个可以真正的可以正常运行的应用程序，但这个示例足够小，而且可以将我们在 FoxPro 中使用的许多功能翻译到 X# 。

我们的示例应用程序

原始的 FoxPro 应用程序是一个简单的待办事项管理器 FoxToDos。如果您有熟悉的感觉，那是因为我从 Rick Strahl 的 Vue 演讲中借用了 UI（感谢 Rick！）。我甚至借用了他的 DBF 表，而他的待办事项列表要比我写的酷的多。

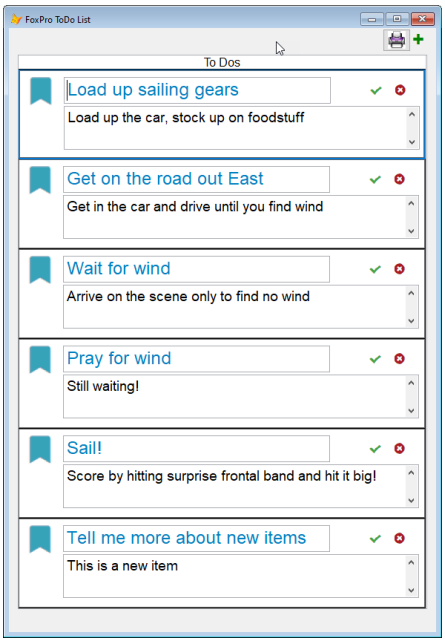


图 1: FoxToDos

您可以在我的 GitHub 账户 https://github.com/eselje/FoxToDos. 获得 FoxToDos 的源码。

在源码中包含此处显示的所有内容。

FoxToDos 没有基于任何的应用程序框架，因此它比任何实际的应用程序都更简单，也更不可靠。它包含以下内容（按顺序）：

- ToDos.dbf，一个包含任务的自由表。

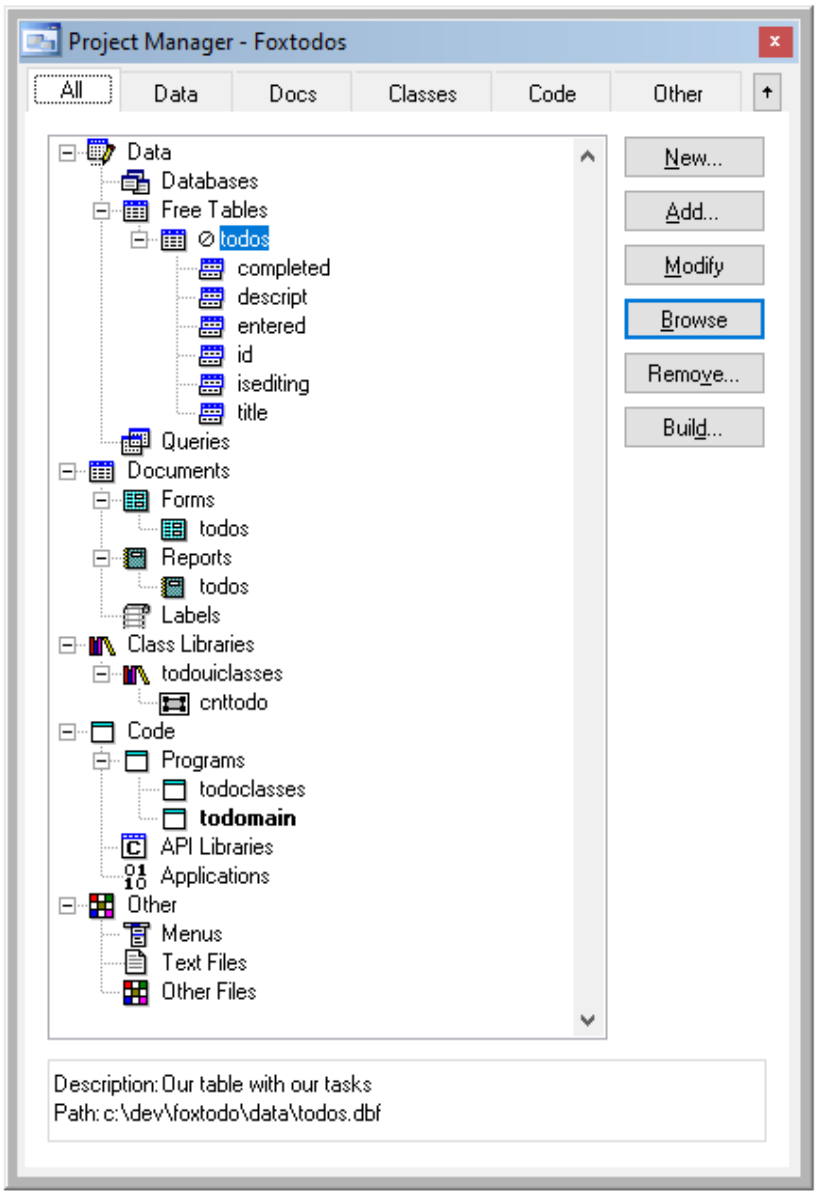


图 2: FoxToDos 项目结构

- ToDos.scx，一个用于用户界面的表单，它包含一个使用自定义控件 cntToDo 的 Grid 。
- ToDos.frx，一个用向导生成的简单报表，用于希望打印任务列表的人。
- ToDoUIClasses.vcx，一个可视化类库，它包含我们用于表单中 Grid 的自定义控件 cntToDo 。
- ToDoClasses.prg，它包含我们需要的非可视业务对象。同时我们有一个用户界面控件 cntToDo，用于将所有任务以一致的方式呈现在表单中。最后，我们有一个使用先导生成的简单报表，将这些转换为 X# 应该足以让我们获得完整的 X# 体验。

- ToDoMain.prg，一个简单的主程序。

转换策略

我们将使用以下步骤将此 Visual FoxPro 应用程序转换为 X#：

1. 在 Visual Studio 中创建一个新的解决方案
2. 将 ToDoClasses.prg 中的类重写为 X# 类
3. 在 Visual Studio 中对业务对象进行单元测试
4. 在 Visual Studio 中创建一个表单，该表单使用这些业务对象与数据库进行交互，并且还包含等效的用户界面符合控件。
5. 创建一个可以设置、运行表单并可以关闭的应用程序。
6. 查看创建报告输出任务列表的可能性。

开发环境

使用 X# 进行开发，你有三种开发环境的选择：

1. 使用任何您喜欢的编辑器（ahem, VI），然后使用命令行编译器进行编译。这是我留给读者自行探索的选项。
2. XIDE，X# 的集成开发环境，它可以和 X# 一起下载。XIDE 是一个完美维护的环境，与 Visual FoxPro 的 IDE 有很多共同点。它是用 X# 编写的，因此它也作为一个生动的例子，说明该语言在正确的地方可以做什么。
3. Visual Studio，专业版（付费版本）或社区版（免费版本）。Visual Studio 的最大有点是，全世界许多开发人员都在使用它，因此它得到了开发人员和社区的大力支持。它具有很多功能，但是以我的经验来说，在性能上还是有点差强人意，这是一种资源浪费。你可以认为它是一只猪或是一条狗——如果您愿意的话，它是一个热狗（如果您看到这个笑话时笑了，请给我发电子邮件，下次我们见面时我会和您喝一杯）。

Visual Studio Professional 2017 是我在此时使用的开发环境。如果您不熟悉 Visual Studio ，那么在 X# 帮助文件中您可以找到如何在 Visual Studio 中使用 X#。

创建一个新的解决方案

用 Visual Studio 的话说，“解决方案”是应用程序的主要架构。它是主要项目单位的项目集合。您最好将业务对象放在自己的项目中，并将用户界面元素放在另一个独立的项目中，因为这样，您就可以在多个解决方案中将业务对象分离并重新使用（即“引用”）。

要从头创建新的解决方案，请从菜单“文件-新建-项目”开始。该对话框使您指定包含新项目的解决方案的名称，并为您创建解决方案（如果要创建一个新项目并将其作为已有解决方案的一部分，那么您必须打开该解决方案，然后选择菜单“文件-添加-新建项目”。参看图 6）。

因为我们已经安装了 X#（您可以直接从 http://www.xsharp.info 下载安装，这个过程留给读者自己进行练习），因此我们可以在这里使用 Xsharp 模板。

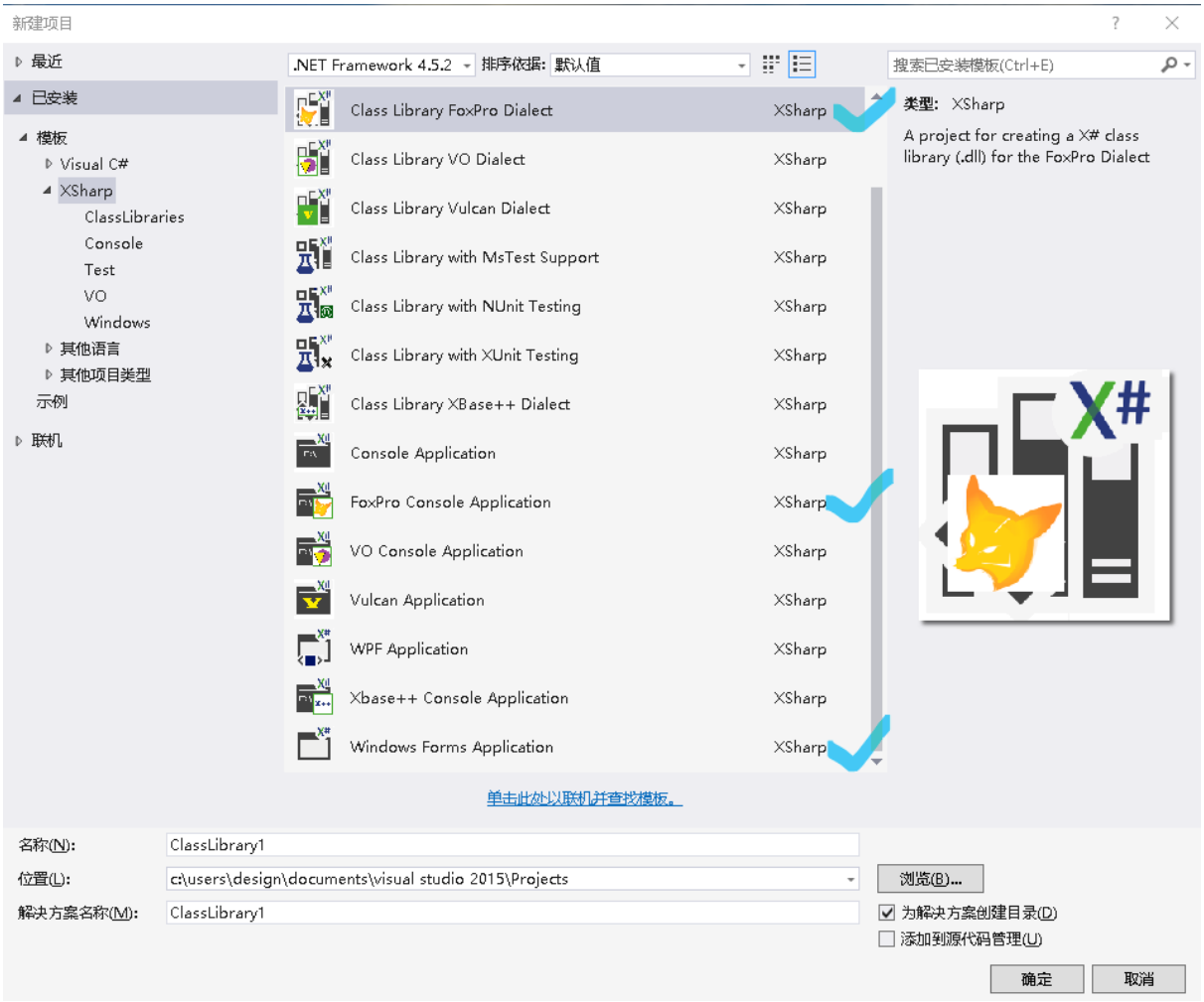


图 3: 新建项目对话框，内置有趣的项目模板

项目模板能为您做什么？

项目模板设置针对项目类型进行了调整的项目的某些属性（参见图 4）。它可能还包括该类型项目通常使用的代码文件和其他资源。例如，请注意基于 **Class Library FoxPro Dialect** 模板创建项目后是如何将方言自动设置为 FoxPro 的。基于此模板的项目还包含一个基准的 PRG 库，以帮助我们入门（图 5），但是，它们也可以包含多达一个完整的应用程序框架。

Visual Studio 允许开发人员创建自己的项目模板，就像 Word 和 Excel 允许您为文档和电子表格创建自己的模板一样。

作为 FoxPro 开发人员，图 3 中显示了三个我们特别感兴趣的项目模板：

- **Class Library FoxPro Dialect**：这种项目将在 General 属性中预先设置 FoxPro 方言（参看图 4），并包括准系统入门类定义。

- **FoxPro Console Application**： 它也将方言设置为 FoxPro，还将“Output Type”设置为“Console Application”。这使我们可以使用正确的退出代码在 FoxPro 方言中创建命令行实用程序，这对于 Visual FoxPro 来说是一件很困难的事（译者注：针对创建控制台应用程序？）。命令行应用程序可以与诸如持续集成工具 Jenkins 之类的开发管道一起很好的工作。

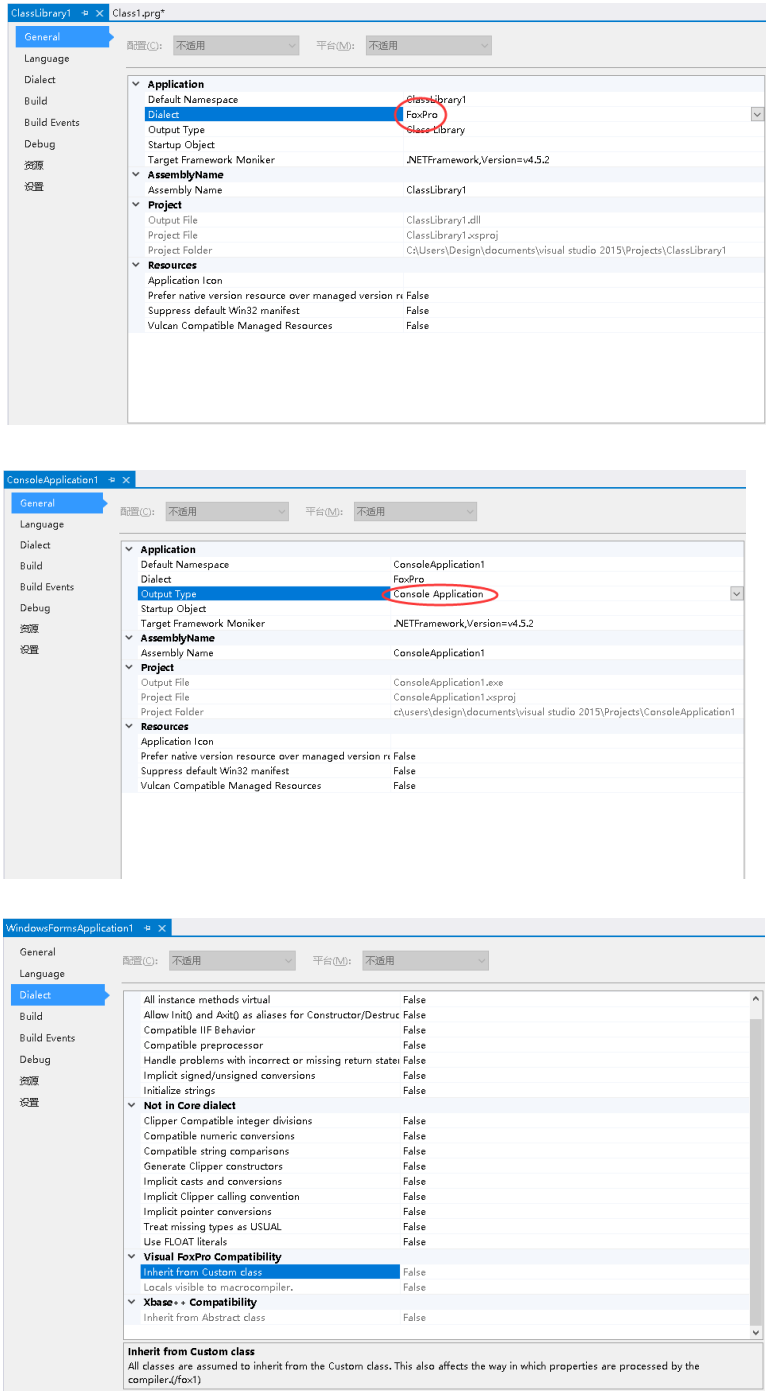


图 4：由项目模板设置的项目属性

- Windows Forms (或 WPF) Application：这些并不是 FoxPro 特有的，但我们将探索使用它们为任务列表创建用户界面

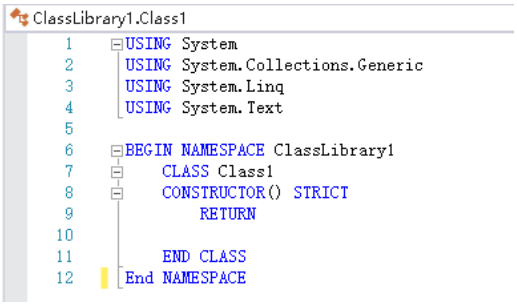


图 5: 项目模板中的开始代码

模板中入门类有趣的地方是，它使用的语法看起来并不像 FoxPro 的。X# 开发团队现在已经对 VFP 语法有了更深入的了解，但是我怀疑这个模板是在正式支持之前编写的。我们稍后会在转换过程中使用更新后的语法。

需要特别注意的是 Visual FoxPro Compatibility/Inherit from Custom Class 设置。如果您希望代码像在 VFP 中那样工作，那么你就要将其设置为 True。除此之外，这将导致 X# 触发一个 `Init()` 方法（以前 X# 使用 `Constructor()`）并为我们的“属性”创建虚拟的 _access 和 _assign 方法。

为便于阅读，我们将要迁移的类的代码放在了附录 A 中。您可以从 https://github.com/eselje/XToDos 存储库中下载 FoxPro 源代码，并从 https://github.com/eselje/XToDos 下载最终的 X# 解决方案。

FoxPro 类库包含两个类定义：

- ToDo，用于管理单个任务
- ToDos，用于管理 ToDo 对象的集合

尽管我可以用 VCX（可视类库）编写，但是我还是选择直接编写代码，因为这可以更容易的说明如何向 X#过渡，X#中是没有“可视类”这样的概念的。所有的代码均以文本形式完成，这在源代码控制方面是一个巨大的优势，因为不需要在 VFP 中实现序列化二进制文件所需的多种解决方法。正如我们在讨论表单、菜单和自定义控件时所看到的那样，Visual Studio 有一个“视觉”元素，但是源代码本身就是文本。

免责声明：您在本示例代码中发现的任何不完善之处或存疑的设计决策都可能是有目的的包含在其中以说明某些问题。这种做法可能是个错误。谁知道呢？

转换类

ToDo => XtoDo

我们将从 ToDo 类开始，该类将单个任务读写到 DBF 种。

在 FoxPro 种我们这样开始定义类

```
DEFINE CLASS ToDo AS Custom
  Name = "ToDo"
  cId = ""
  oData = .null.
  lNew = .f.
  lSaved = .f.
  lLoaded = .f.
  oException = .null.
```

在 Xsharp 中，我们这样开始定义

```
USING System
USING System.Collections.Generic
USING System.Text

BEGIN NAMESPACE XSharpToDo
   DEFINE CLASS XToDo as Custom
    public id as string
    public title as string
    public descript as string
    public entered as datetime
    public completed as boolean
    private isEditing as boolean
```

Using

最开始的 X# `using` 语句有点儿类似 FoxPro 的 `set classlib to`，因为它们告诉程序：“嘿，我在其他地方存储了一些代码，我可能会在这里使用它们，因此请确保它们对我来说是可用的，但是如果调用了一个您不认识的函数，它可能就在这些类中”。你也许并不使用它们，但是如果您引用它们，那么编译器将包含它们。在我使用的过的其他 .NET 语言中，您可以通过右键快捷菜单选择“删除未使用的 Using”(在我看来，这是个很笨拙的方式)来删除所有不需要的语句，但是这个方法似乎不适用于.PRG 文件，因此，您将需要自行删除不需要的任何东西以缩小代码库的大小。

.Net 具有一个极简主义的概念，类似于“如果您想要它，则必须包含它，否则它将不可用”。`System` 是根命名空间，它包含我们可能在VFP中称为“基类”的内容，但不包括 _Classes.vcx 中包含的可视基类，而是诸如字符串、整型等本地数据类型。如果没有显式的包括 `System` ，那么每次调用它们时，你都必须在系统中添加它到所使用的任何类的头部，因此 `Console.Write()` 就必须写为 `System.Console.Write()`。

您还应该注意的是，USING System 并不包括 System 的每个子命名空间。您必须明确要访问的任何库。有关 .NET 本机类库的完整列表，请参阅 https://docs.microsoft.com/en-us/dotnet/standard/class-library-overview. 上的（优秀）文档。

Begin Namespace

命名空间对于 FoxPro 开发人员来说并不陌生，尽管我们可能没有这样称呼它们。如果您在 Visual FoxPro 中创建 OLEPUBLIC 类，那么创建的 DLL 的名称就是它的命名空间。然后，您就可以从具有 CreateObject("NameSpace.ClassName") 的另一个程序中使用该类。

通过使用 BEGIN NAMESPACE 在代码中指定命名空间，.NET 允许您将类的代码分布在多个文件中，这使得管理源代码更加容易。这样，当您只想对同一个类库中的另一个类进行细微调整时，另一个人正在重构整个类这样的情况就不会产生冲突！

接下来的几行非常相似。.NET 是强类型的，因此，当您指定类属性时，也必须指定类型。此外，在 VFP 中，我们有非常强大的 SCATTER 和 GATHER，允许我们使用一个属性（在我的示例中为 oData 属性）动态存储字段值，而在 X#类中，我们还没有此属性（更新：支持此特性的 2.6 版本在 2020 年 9 月 20 日发布，这对于我这篇文章来说来的太迟了！），因此，我们必须明确字段名称。

属性 v. 字段

.NET 类与 Visual FoxPro 类之间存在根本性的差异。在 VFP 中，当我们在类中添加所谓的“属性”时，我们可以立即为该属性赋值而无需经历任何的麻烦。这是很糟糕的，因为根本不检查输入的内容，任何人都可以读取该值。我们通过在属性中添加 _access 和 _assign 方法来解决此问题。属性（公共、保护、隐藏）的“可见性”影响其他对象是否可以看到属性，但对类自身可见的值没有任何影响。

.NET 类将这些称为“字段”，而不是属性，其可见性取决于它们是“Pbulic”还是“Private”。Public 字段类似于我们的属性，基于前述的原因，我们不建议这样做。

最佳实践

使用 PRIVATE FIELDS 和 PUBLIC PROPERTIES

.NET 类的属性是面向公众的接口，类似于我们的 _access 和 _assign ，它们具有 get() 和 set() 方法，用于过滤对字段的输入或限制输出。

为了模拟 Visual FoxPro 的类行为，X#类有一个“Inherit from Custom Class”（继承自自定义类）选项，在 FoxPro 项目模板中默认设置为 True。在这背后，是 Cutom 类在 .NET 的字段中模拟 FoxPro 属性。

Init() vs Constructor()

FoxPro 的所有类都带有接收参数的 `Init()` 方法。每个类只有一个 Init，并且必须围绕它发送的参数的可能组合进行编码。X#类具有一个 Constructor 方法，并且可以使用不同的“签名”来重载它们：参数可以自由组合，真的很棒。

FoxPro

```
PROCEDURE Init
LPARAMETERS cId
This.cId = cId
IF EMPTY(cId)
    This.New()
ELSE
    This.Load(cId)
ENDIF
    ENDPROC
```

X#

```
public FUNCTION Constructor()
// 没有参数，新任务
    This.New()

public FUNCTION Constructor(cId AS String)
// 具有参数，现有任务
    This.cId = cId
    This.Load(cId)
```

在 X#的 FoxPro 方言中，X#中的类定义确实具有 `init()`方法，您可以像使用 Visual FoxPro 那样使用它。

当您创建任务时，可以向其发送任务的 ID，或者要创建新任务时，将参数置空。FoxPro 的 `init()` 通过检查参数并根据是否传入任何内容使用分支来处理此问题，但是 X#具有重载的构造函数，该构造函数要么采用 ID 作为参数，要么不使用它。我发现，这样的方式更加直观。

`new()` 函数突出显示了 X#与 FoxPro 的相似之处，但是还可以让您获得额外的功能。我们使用.NET 创建 GUID 所需的代码（在这种情况下，可以从我们始终包含的 System 中获得代码，但是我们可以使用整个.NET 库，如果出于某种原因，我们不喜欢使用这个 GUID 库，我们也可以引用其他的库）。

FoxPro

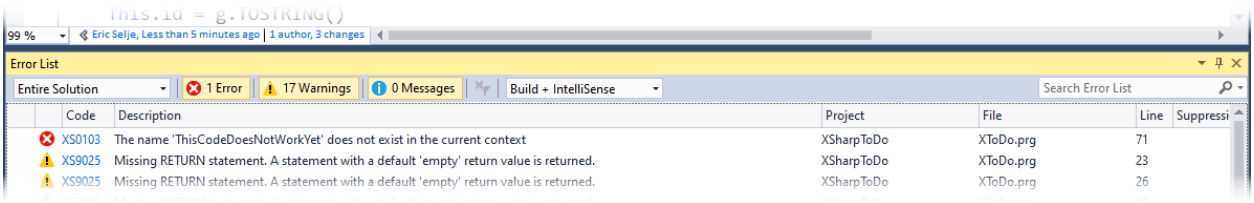
```
PROCEDURE New
LOCAL lUsed, oGUID
lUsed = This.OpenToDos()
oGUID = CreateObject("scriptlet.typelib")
SCATTER BLANK NAME This.oData MEMO
This.oData.Id =Strextract(oGUID.GUID, "{", "}" )
This.oData.Entered = DATETIME()
This.lNew = .t.
RETURN This.oData
```

X#

```
PROCEDURE New
LOCAL lUsed
This.Clear()
VAR g = GUID.NEWGUID()
This.id = g.TOSTRING()
this.entered = DateTime.Now
this.isEditing = true
this.isNew = .t.
RETURN This.oData
```

测试我们的代码

现在，我们已经为 X#类编写了一些方法，我们可以检查它在那个悠久的传统中是否有错误：它可以编译吗？按 Ctrl+Shift+B 构建解决方案，“输出”窗口将显示发现的所有问题。双击出现问题的行，将带您直接进入编辑器中的代码，或者，单击错误代码将带您到可以为您提供有关该错误更多信息的网页。



一旦编译完成，没有任何错误，我们希望找到一种方法来确保它确实有效。

命令窗口？

FoxPro 开发者喜欢打开命令窗口，实例化类并手动调用方法来“测试”我们的代码。如果它们没有达到我们的预期，我们将设置一个断点并以调试模式浏览代码。

Visual Studio 并没有命令窗口。如果您安装了 XIDE 环境，您将得到一个类似命令窗口的，被称为 XSI 的窗口——X#解释器（去年白皮书中有关于 XSI 的更多信息）。由于我们在此演示中使用 Visual Studio，因此我们将创建一个快速的控制台应用程序来“测试”我们的代码。

要创建控制台应用程序，请右键单击解决方案，选择“添加-新的项目”（参看图 6），将其基于 **FoxPro Console Project** 模板（请参考图 3）并为其命名，然后添加对要测试的库的引用（图 7）。将新项目设置为 Startup 项目，并更改代码以编写基本测试：

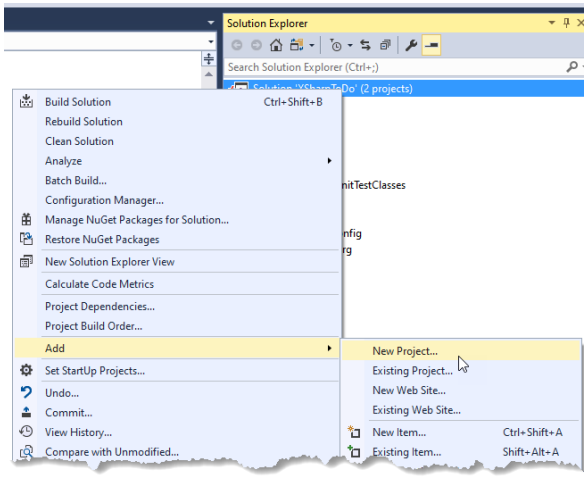


图 6: 将项目增加到已有解决方案

```
USING System
USING XSharpToDo

FUNCTION Start() AS VOID STRICT
    LOCAL oToDo AS XToDo, cTestId AS String, cDescript As String
    cTestId = "EDF53AEF-5C29-4DC4-A"
    oToDo = createObject("XToDo")
    IF oToDo.openToDos()
        SET DELETE ON
        SCAN
            Console.WriteLine("{0:00}: ID: {1}, {2}",  RECNO(), ToDos.id, ToDos.descript)
//            ? RECNO()  ToDos.id ToDos.descript
            IF ToDos.Id = cTestId
                cDescript = ToDos.descript
            ENDIF
        ENDSCAN
        ? cTestId +": " + cDescript
        oToDo.Load(cTestId)
        oToDo.closeToDos()
    ELSE
```

```
        ? "Could not open ToDos.dbf"
        ? "Default folder is " + SET("DEFAULT")
ENDIF
WAIT
 RETURN
```

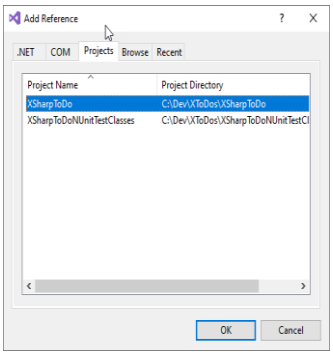
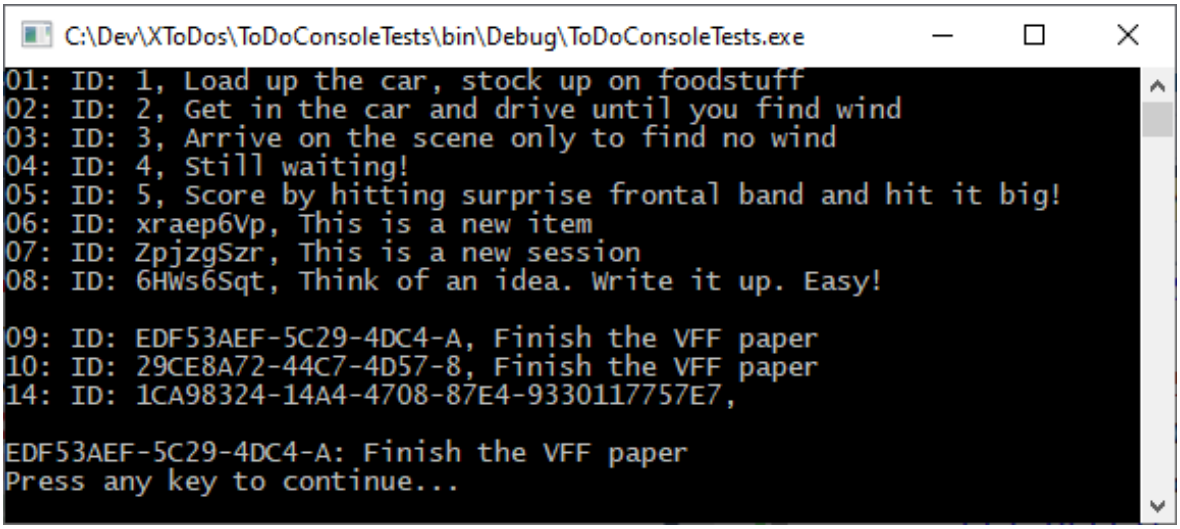


图 7：增加一个引用

这是 X# 代码，但是看起来应该很熟悉。除了我使用了免费的 Console.WriteLine 来展示 X#的一些额外功能，除此之外，这就是 VFP 代码。

经过几轮调试（作为经验丰富的 FoxPro 开发人员，您可以毫无疑问的使用 Visual Stuido 的调试器），您应该在“输出”窗口获得预期的结果：



“稍等”，您说，“当我运行此应用程序时，我会看到一个命令窗口，而您说 Visual Studio 没有命令窗口！”

您不用太兴奋。此“命令窗口”仅在您正在积极调试代码时可用，甚至它不能理解 FoxPro 的所有，因此它并不是特别有用（暂时来说）。

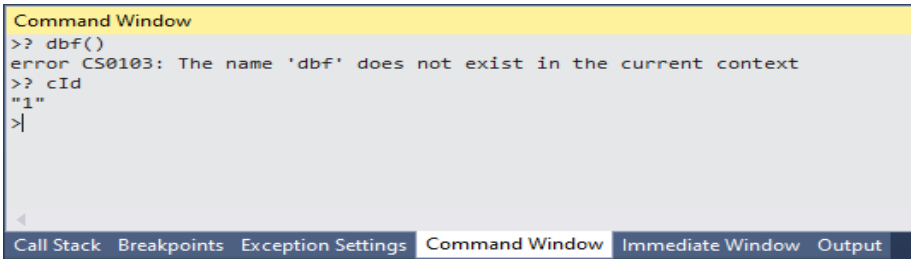


图 8: 命令窗口

单元测试

测试代码更好的方式是编写单元测试，实际上，真正的“测试驱动”开发会指导我们甚至在开始编码之前就编写这些代码。但这不是真正的 TDD，因为：a)这些不是“真正的”单元测试（它们与真实的数据库进行交互）；b)我们现在才开始编写它们。重要的是要承认我们何时知道正确的做法是什么，而我们仍然没有这样做。

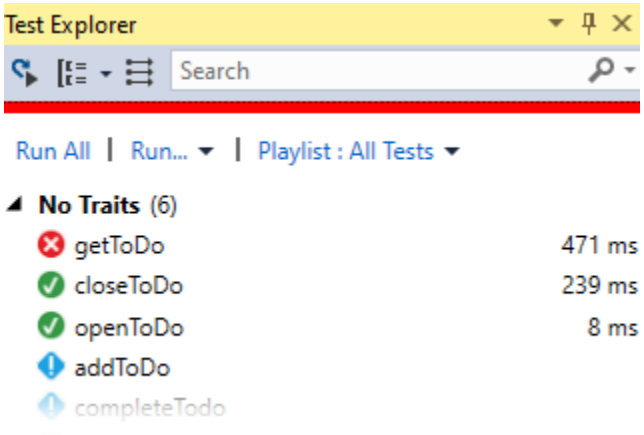
FoxPro 提供了一种流行的单元测试工具 FoxUnit。这是一个单独安装的程序（理想情况下是通过 Thor 进行的），并没有继承到 IDE 中。

而 Visual Studio 将其完全内置在 IDE 中，并且您可以选择多种测试框架。我选择 Nunit 进行演示是因为它和 FoxUnit 非常相似。为了创建测试，你将基于 **Class Library with Nunit Testing** 的新项目添加到解决方案中（参见图 3）。

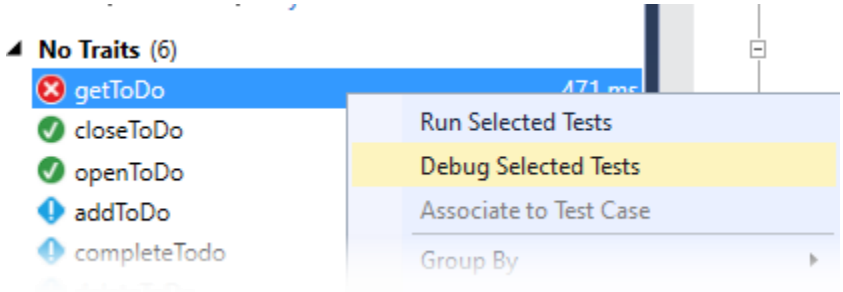
接下来，添加对要测试的库 XSharpToDo 的引用，就像我们使用基本控制台测试应用程序时所做的那样，并编写一个测试：

```
[Test];
   METHOD getToDo AS VOID STRICT
   VAR oToDos := XSharpToDo.XToDos{}
   VAR oTodo := oToDos.getToDo("EDF53AEF-5C29-4DC4-A")
   VAR cExpected := "EDF53AEF-5C29-4DC4-A"
   Assert.AreEqual(cExpected, oTodo.id, "Did not get the right TODO")
   RETURN
```

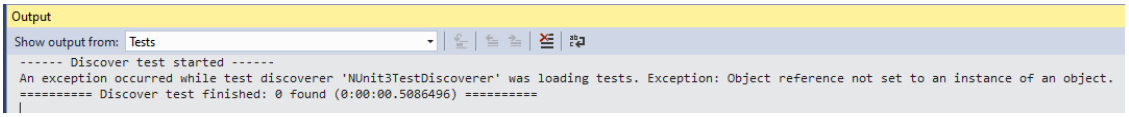
该测试现在显示在“测试资源管理器”中。当您单独或完全（Ctrl+R, A）运行测试时，会清除的表明测试是通过、失败还是尚未实施。它还显示测试运行了多长时间，这是方法性能的早期指标。



如果您难以从测试中获得预期结果，请选择“调试选定的测试”，以便在设置的断点处停止。也就是当您可以使用“命令窗口”时，就像您在 Visual FoxPro 中使用局部变量和监视窗口时那样。



注意： 我遇到一个非常令人沮丧的 Nunit 问题，它不是 X#的问题，时我在尝试运行测试时一直收到这个消息：：



我并不是唯一遇到此问题的人，唯一可靠的解决方案似乎是清除 Nunit 缓存（删除文件夹），地址为

```
%appdata%\..\Local\Temp\VisualStudioTestExplorerExtensions\<nUnitVersion>
```

然后重启Visual Studio。我将来会考虑使用 Xunit 或 MSTest ！

最佳实践

将每个类放在自己的代码文件中

现在，我们知道了如何向 ToDo 类添加测试方法，并且我们可以继续添加其余的方法。该代码很简单。有关该测试类的完成的 X#代码，请参阅附录 B，当然，这可能过时了，所以请转到存储库以获得当前的测试代码。

向我们的命名空间添加其他的类

在 FoxPro 中，我们可以并且通常是将一个类库的所有类置于一个 VCX 或 PRG 文件中。FoxPro 项目的名称决定了我们类的命名空间。

在 .NET 中不建议保留这种习惯。X#中的每个类都应位于自己的代码文件（PRG）中，并且库中所有类均位于一个项目中。这种组织结构使源代码嵌入时很容易确定修改哪个源代码变得更加容易，而且可以减少文件冲突。

要将新类添加到现有项目中，请右键单击该项目，然后选择“添加-新项目”。您将看到“增加新项目”对话框（图 9）.这里显示了项目模板的列表，该列表类似于前面提到的项目模板，区别就在于这里您仅可以创建一个文件而不是整个项目。

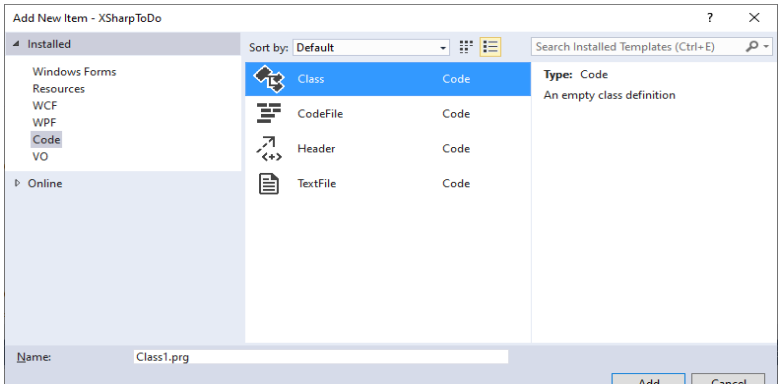


图 9: 新项目模板

约定是用要创建的类名命名 new.PRG。命名空间默认为项目名称，这可能是您希望的名称，但是您可以根据需要更改它。一个项目中可以有多个命名空间。

启动项目

前面创建控制台项目以测试软件时，我们将其设置为“启动项目”。当您单击“开始”时，一个应用程序必须知道该怎么做。因此在“解决方案”中的某个位置必须至少有一个“启动项目”，并且该启动项目必须具有一个名为 `Start()` 的类，以使您的工作顺利进行。应用它可以设置环境和全局变量、打开表、传入参数。要为您的解决方案设置启动项目，请右键单击项目，然后选择**设为启动项目**。

可以通过右键单击解决方案选择设置启动项目来设置多个启动项目。例如，如果要启动 Windows Form 应用程序时启动 .ASP NET 网站，这就很有用。

如果解决方案是 XAML 项目，那就是个例外，我们在下一节中进行讨论。

用户界面

至此，我们的业务逻辑已转换并经过测试，但是此应用程序上没有恰当的用户界面（尽管有一个控制台项目）。我们可以为我们的业务类别选择任意数量的用户界面，例如 Angular 网站或手机应用程序，但是 Windows 窗体是最接近 VFP 应用程序的用户界面。

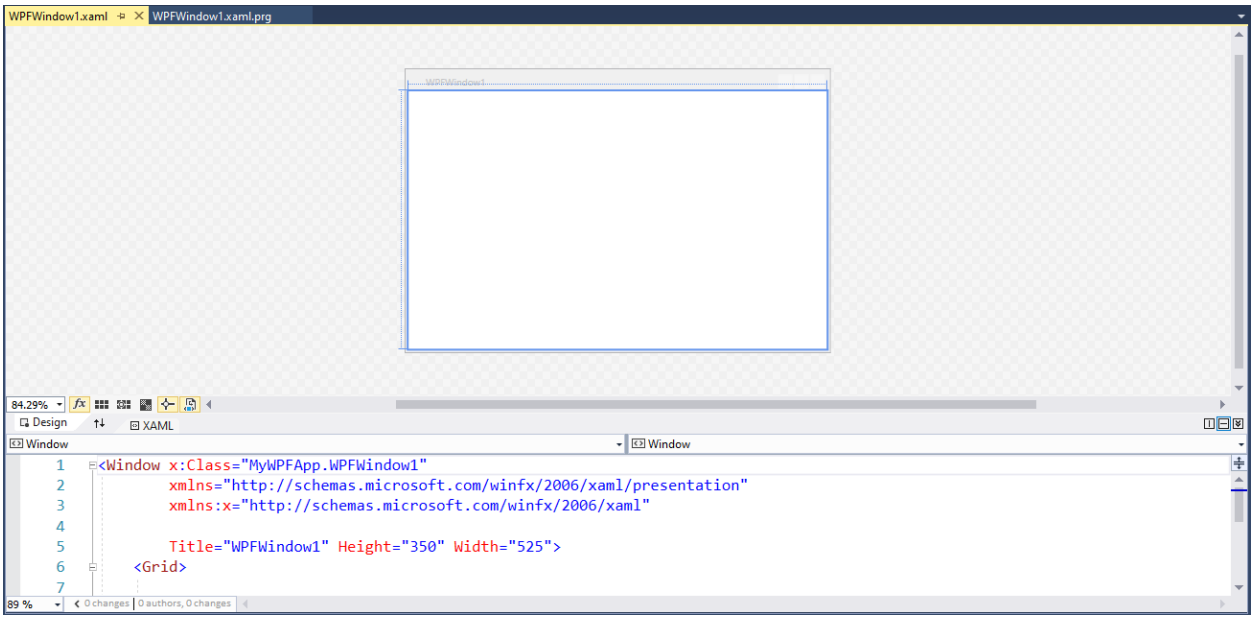
Windows 窗体有两种形式。猛一看，原始的 WinForms 似乎与 Visual FoxPro 表单有很多共同点。设计外观看起来很相似，并且有一个工具箱，其中包含许多熟悉的空间，例如 TextBox 和 CheckBox。但是，FoxPro 表单具有看似强大的功能，它使您可以包含构成和继承一层又一层的控件，而 WinForms 根本无法与之匹敌。微软确实曾说过，WinForms 不会向.NET Core 迈进。从那之后，他们就退缩了，但是 WinForms 的未来比替代方案更加的不确定。

然后是更新、更复杂、功能更强大的 WPF(Windows Presentation Format)表单。在背后，它使用 XAML 的 XML 方言来布局接口，但是命令是使用 C#或 X#编写的。这里的想法是，您团队中的 UI/UX 设计人员可以创建表单，而编码人员可以处理实际的逻辑。当然，我们 FoxPro 开发人员通常同时扮演这两个角色。

可以用整本的书描述创建 WPF 表单，因此我们只能在此处涉及最浅显的部分，这足以模拟我们的 VFP 表单。我想说的是，在撰写本文时，要完成所有工作，创建表单，尤其是数据绑定，难度相当的大。在这点，您不必对 X#进行抨击，因为这不是它的错。事实上，X#开发人员正在开发一种实用程序，可以将 FoxPro 表单转换为 WPF 或 WinForms，但截止目前此项工作还未完成。一旦完成，它对我们顺利度过难关是非常宝贵的。

我们希望将用户界面和业务类放在一个单独的项目中，但是它可以是同一解决方案的一部分。要创建这个新项目，请在解决方案上右击鼠标，然后选择“添加-新项目”（参见图 3）这次我们选择“Xsharp-Windows-WPF Application”作为项目模板。项目名称将会是文件在磁盘存储位置下的文件夹名称。

WPF 应用程序的项目模板包括 WPF 表单的初始文件（WPFWindow1.xaml），其中包括“隐藏代码”文件（WPFWindow1.xaml.prg）。这是一个 PRG，因为我们选择了 Xsharp 项目模板，因此它实用了 X#中的 Core 方言的语法。如果要实用 VFP 语法，则需要修改项目属性（参见图 4）。



它还包括一个启动文件 App.xaml 。

```
<Application x:Class="MyWPFApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="WPFWindow1.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

以及一个 App.xaml.prg，但是它其中没有代码。如果将这个新的 XAML 项目设置为启动项目，编译完成后，它实际上可以工作的（图 10）！但是，它没有 Start() 方法，应用程序是如何知道它应在启动时启动该表单？在 XAML 中，指示起始格式的另一种方法是在 XAML 的 Application 标记的 StartupURI 属性中。您可以在我们的代码清单中看到它指向 WPFWindow1.xaml。

图 10: 我们第一个XAML应用程序

该应用程序并不强大。实际上，它什么都没有做，但是请注意，我们不必执行 READ EVENTS 或 DO FORM 或任何其他操作即可获得基本功能。这是因为该应用程序是.NET 中 Application 类的实例，除非您使用第三方框架，这是 VFP 从未有过的。

在表单中添加控件

我们的 VFP 示例在表单的右上角有两个按钮用于打印和添加任务。在 WPF 中，我们不能简单的向表单添加按钮。

首先，我们必须先在窗口中添加某种类型的布局容器。这些类似于 FoxPro 的容器对象，但是它们是具有不同的行为不同类型容器，并且允许我们放弃惯用的绝对定位和锚定。这种布局非常灵活，可用于各种屏幕分辨率和各种设备。下面是受欢迎的特性，但实际上还有更多：

- Canvas：最类似于 FoxPro 的容器，在其中您必须将对象手动放置在可用空间中。
- Grid：您添加的每个控件都排列在棋盘的行或列中。
- StackPanel：根据方向，您添加的每个控件都位于上一个对象的旁边或下方。
- WrapPanel：添加的每个控件都位于上一个控件的旁边，并在到达水平边缘时环绕。
- DockPanel：与 VFP 可停靠容器非常像，这些容器位于其父容器的边缘或中心。
- ToolBarPanel：StackPanel 的子类，如果控件不适合，该子类将控件放入“溢出”区域。

您也可以混合并嵌套这些容器。这是将两个按钮添加到 StackPanel 内部的 ToolBarPanel 中的代码：

```
  <StackPanel Orientation="Vertical">
    <ToolBarPanel Height="60" Orientation="Horizontal" HorizontalAlignment="Right">
      <Button Background="Transparent" BorderThickness="0">
        <Image Name="imgNew"  Source="Images\Ribbon.png" Height="24px" Width="24px"
HorizontalAlignment="Right" Margin="10" ToolTip="Add Task"></Image>
```

```
      </Button>
      <Button Background="Transparent" BorderThickness="0">
        <Image Name="imgPrint" Source="Images\Print.png" Height="24px" Width="24px"
HorizontalAlignment="Right" Margin="10" ToolTip="Print To-Do List"></Image>
      </Button>
    </ToolBarPanel>
  </StackPanel>
```

猛一看代码似乎有些冗长，但实际上非常简单。Visual Studio 内部的 Intellisense 使得向每个控件添加更多属性非常容易，而且您还有一个非常熟悉得“属性”窗格（图 11）。

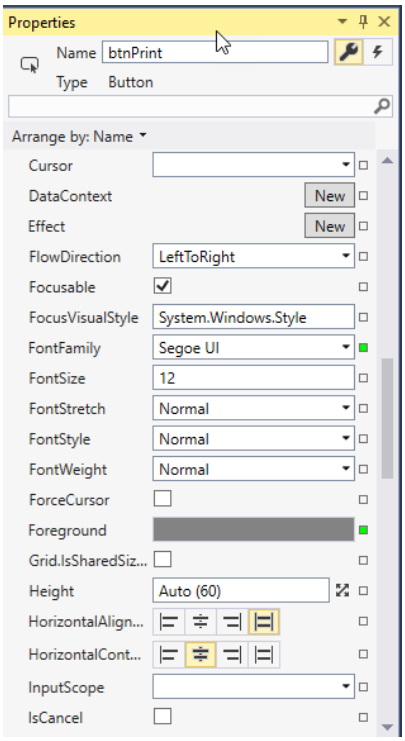


图 11: 控件属性面板

将事件绑定到控件

现在，我们的“添加”和“打印”按钮什么也做不了，因此，我们需要继续。由于添加和打印可能也是你想从窗口菜单调用的事件，并且我们不想重复劳动，因此我们可以在窗口中创建“CommandBinding”。这为我们提供了一个路由的中心位置，并可以确定事件是否在任何给定时间都可行（例如，除非剪贴板中有内容，否则无法粘贴；除非有任务要执行，否则无法打印任务列表。）。将此代码添加到<Window>元素下面：

```
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.New"
   Executed="NewCommand_Executed"
   CanExecute="NewCommand_CanExecute" />
  <CommandBinding Command="ApplicationCommands.Print"
    Executed="PrintCommand_Executed"
    CanExecute="PrintCommand_CanExecute" />
</Window.CommandBindings>
```

这个命令绑定集合为我们的每个命令命名，告诉我们事件触发时该怎么做，以及事件是否可以被触发。在这背后，我们添加了这些方法（我在 WPF 项目中使用 C#，但我可以选择使用 X#，因为这里的代码量很小，因此无关紧要）：

```
private void NewCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
       { e.CanExecute = true; }
private void NewCommand_Executed(object sender, ExecutedRoutedEventArgs e)
      { MessageBox.Show("New Task"); }
private void PrintCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
       { e.CanExecute = true; }
```

```
private void PrintCommand_Executed(object sender, ExecutedRoutedEventArgs e)
        { MessageBox.Show("Print"); }
```

现在，我们仅需要向每个按钮添加一个属性即可将它们链接到命令：

Command="ApplicationCommands.New"
Command="ApplicationCommands.Print"

现在，当我们启动应用程序时，我们将看到我们的窗口，其两个按钮都位于 tackPanel 内部的 ToolBarPanel 中，我们得到的结果如图 12 所示：

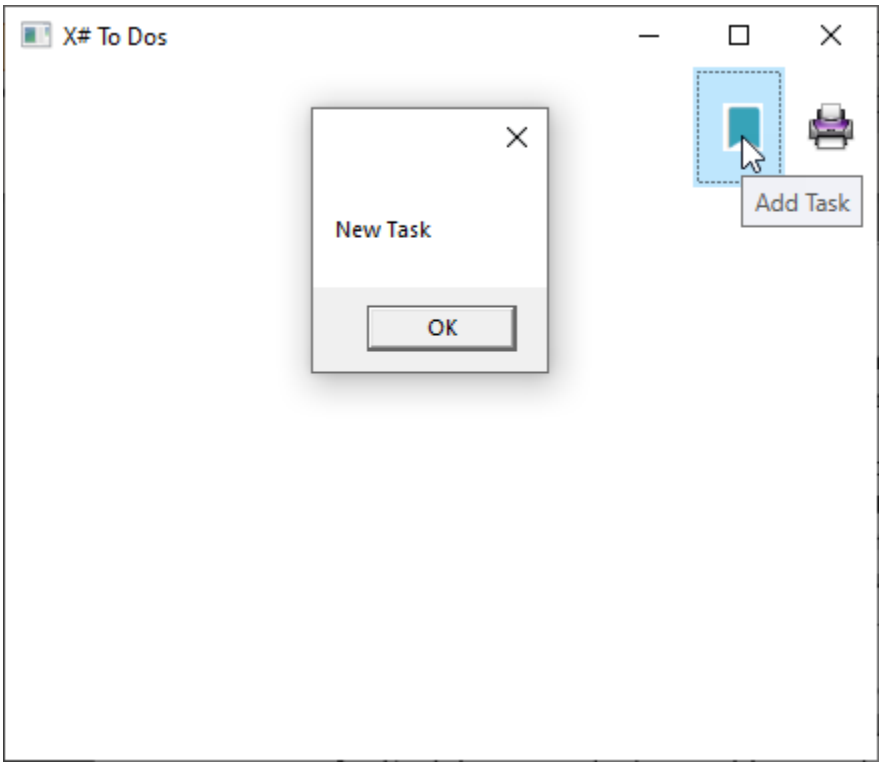


图 12: 我们的应用程序，已得到改善

将我们的业务对象绑定到表单

我们花了很长的时间创建和测试我们的业务对象，是时候使用它们了。

1. 添加 `using XsharpToDo` 语句，以使窗口可以找到我们的类

2. 在类中添加一个属性：

```
XToDos oTasks = new XToDos();
```

3. 实例化窗口时，将数据加载到我们的对象中：

```
this.oTasks.Load();
```

4. 更改我们的按钮以调用我们的业务对象中的方法：

this.oTasks.New(""); // 在 NewCommand_Executed 按钮中
this.oTasks.Print(); // 在 PrintCommand_Executed 按钮中

我已经在清单 1 中突出显示了额外的代码。现在单击“新任务”按钮，通过业务对象将空白记录插入 DBF!

```
using XSharpToDo;
using XSharp.VFP;

namespace ToDoInterface
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        XToDos oTasks = new XToDos();

        public MainWindow()
        {
            InitializeComponent();
            this.DataContext = this;
            this.oTasks.Load();
        }

        private void NewCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }
        private void NewCommand_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            // MessageBox.Show("New Task");
            this.oTasks.New("");
        }
        private void PrintCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
        {
            e.CanExecute = true;
        }
        private void PrintCommand_Executed(object sender, ExecutedRoutedEventArgs e)
        {
            // MessageBox.Show("Print");
            oTasks.Print();
        }
    }
}
```

清单 1: 窗口背后的代码

将数据绑定到表单

我建议您在 X#中查找 Robert van der Hulst 的有关数据绑定的会议资料。

报表

我不知道有什么工具可以将 FRX 文件转换为等效的.NET，尽管 X#开发人员确实在愿望清单中列出了该工具。有一种名为 ReportPro3 的付费产品，其报表设计器与 FoxPro 的外观非常相似。它具有合并、分组、汇总等功能。虽然它确实支持 DBF 文件，但它还不能使用 FRX 文件。

如果您的数据存储于 SQL Server、MySQL 之类的数据库中，或者实际上存储于非 DBF 的任何其他数据库中，那么有很多可用的报表工具可供您选择，例如 Telerik Reports 或开源的 FastReport（请注意：我并不会为它们进行任何明示或暗示的背书）。

对于输出 DBF 数据，我确实在 X#论坛上看到有关使用 FRX 报告的其他尝试的闲聊。一种意见是将 FoxPro 编写的 FoxyPreviewer 用 X#重写。另一个意见是从 X#内部调用 FoxPro 的本机运行时。无论哪种方式，都需要保留 VFP 来设计或修改报表。事实证明 FoxPro 功能强大到难以模仿！

其他的开发考虑

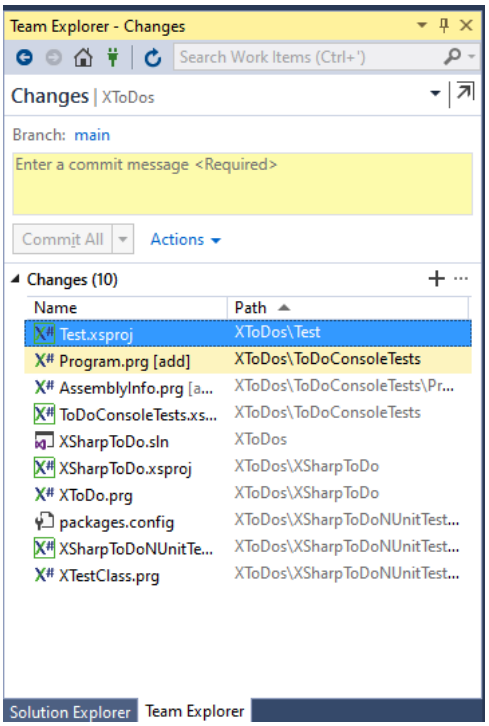
数据库

如图所示，除了熟悉的类似于 FoxPro 的语法外，选择 X#作为开发工具的另一个令人信服的原因是它具有使用 DBF 的能力。但是 X#可以使用从 SQLite 到 Oracle 的许多其他数据库。根据我对 X#论坛的了解，一个非常受欢迎的后端是 Sybase 的 SQLAnywhere。.NET 可以通过其 System.Data 库访问它，该库非常强大。

框架？

在 FoxPro 中，我们有稳定的应用程序框架，从头开始创建程序时，我们可以选择其中的一种。经常使用的诸如 Mere Mortals、ProMatrix 和 CodeBook 之类。目前还没有基于 FoxPro 方言的 X#应用程序框架，但是许多 C#应用程序框架（例如 Oak Leaf 的 MM.NET）可以轻松的与 X#业务对象相结合，使您可以走的更远。

源代码控制



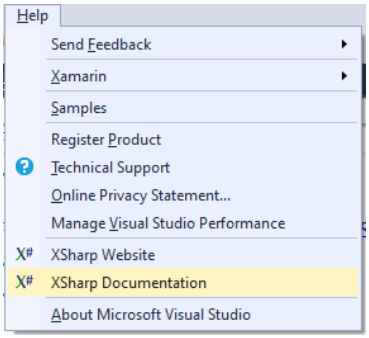
使用 Visual Studio 这样的现代 IDE 时，最好的功能之一就是其集成的源代码控制许多开发人员进行了大量工作来创建可以正确序列化/反序列化 FoxPro 二进制文件的实用程序，然后将这些工具添加到 Thor 中，以尽可能减少管理源代码的麻烦，但是，没有什么比 Team Explorer 窗口更流畅的了。在这里，您可以添加、登入、提交到本地存储库、以及从远程存储库推送和拉出（请参阅 Rick Borup 关于 Git 的白皮书）。由于代码没有二进制文件，因此无需序列化任何内容。

外部库和IDE附加组件

如果您和其他 FoxPro 开发人员一样喜欢 Thor，那么您会喜欢 Visual Studio 的同类产品。Thor 是一个可视化管理器，通过 GoFish5、Project Manager、PEMEditor 和 FoxUnit 等实用程序扩展了 FoxPro 的 IDE。Visual Stuido 也具有自己的市场（https://marketplace.visualstudio.com/），该市场具有数百个 Visual Studio 扩展。

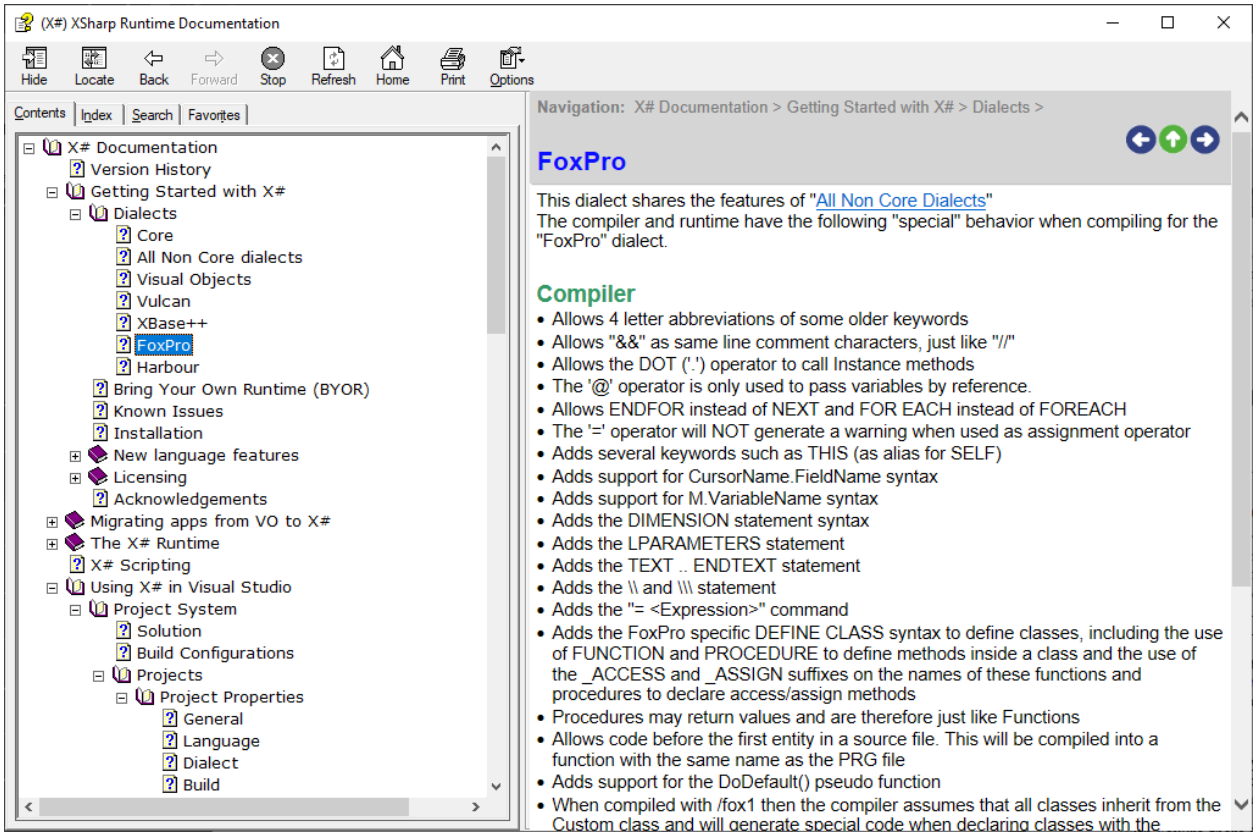
VFPX 是 FoxPro 的许多 IDE 实用程序和其他类库（例如 FoxCharts 和 Log4VFP）的中央源代码存储库，您可以使用它们来美化应用程序。Nuget 是 Visual Studio 的程序包管理器，它具有成千上万个您可以利用的类库，而且大多数是免费的。它包含从网关支付到实体框架到整个应用程序框架的所有内容。

获取帮助



如果您在学习 X#时需要帮助，最好的起点就是 www.xsharp.info。通过 Visual Studio 菜单“帮助-XSharp 网站”，你可以轻松直达。那里有活跃的用户论坛，还有一个专门用于 FoxPro 迁移者的论坛。核心开发人员可以迅速介入回答您的问题。

X#下载中随附的帮助文件也很好。它将使您想起 Visual FoxPro 帮助文件。您可以在“开始”菜单中找到它，也可以在 Visual Studio 中选择菜单“帮助-Xsharp 文档”



总结

这次讨论几乎涵盖了 Visual Studio 的所有强大功能以及在 X#和.NET Framework 中的编码。我希望将一个简单的 FoxPro 应用程序转换为 X#的练习对我们有所启发。以我有限的经验看，我认为 X#为理解.NET 开发提供了一个很好的入口。我认为重要的还包括，尽管 X#使您可以使用类似于 FoxPro 的语法和概念，从而使您可以延续自己的技术生命，但您并不应该受此限制——事实上，整个.NET 框架您都可以使用。

X# 永远无法使您的 FoxPro 代码不加任何修改就可以在其中运行并可以进行编译——这需要您自己的努力。但是，这并不是一件困难的事，它将为您提供重新访问和重构代码的机会，并通过单元测试和集成的版本控制使原有的代码更强壮。

自从我 2019 年在会议上发表白皮书以来，FoxPro 的兼容性取得了令人瞩目的进步。X#是开源的，在我们社区的其他开发人员在做出贡献的同时，核心开发人员聚焦于付费会员用户的愿望清单。嘿，他们必须要付账单！如果您希望开发集中在 FoxPro 兼容性上，那么我希望您以会员身份支持他们。

鸣谢与目录

Twitter 图标来自于 Smashicons 在 www.flaticon.com 所设计的图标

https://docs.microsoft.com/en-us/windows/apps/desktop/visual-studio-templates
https://fox.wikis.com/wc.dll?Wiki~GUIDGenerationCode~VB
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/fields
https://www.youtube.com/watch?v=CniIPEFZ1Oo (数据绑定)
https://www.xsharp.info/itm-help/foxpro-compatibility-list
https://www.wpf-tutorial.com

一个有关基于 Fox 的开发的总体的有趣的哲学讨论：
https://support.west-wind.com/Thread5U70W2EQW.wwt

附录A：原始的 FoxPro 类

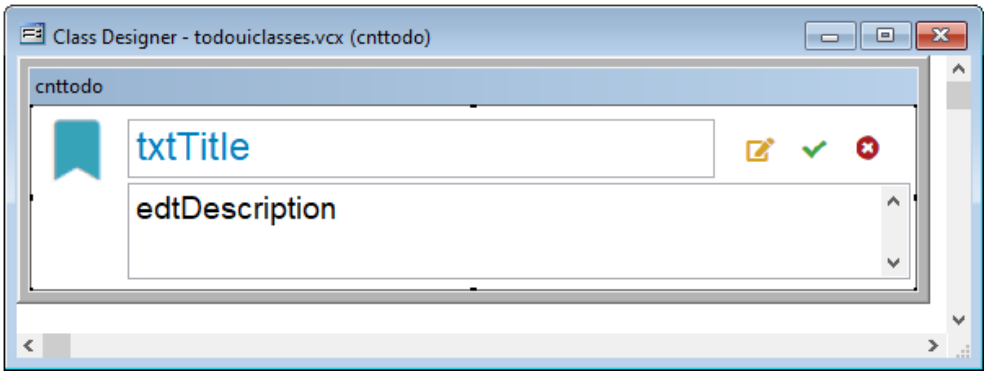
```
* Collection of ToDos
DEFINE CLASS Todos AS Custom
  DIMENSION aToDos[1]    && Array of ToDo
Objects
    nTodos = 0
    cTableName = "data\ToDos"


    PROCEDURE Init
    SET EXCLUSIVE OFF

    PROCEDURE OpenTodos
    IF NOT USED(This.cTableName)
      USE (This.cTableName)
    ENDIF
    RETURN USED(This.cTableName)

    PROCEDURE Load
    LOCAL n
    This.OpenTodos()
    SET DELETED ON
    COUNT TO This.nToDos
    DIMENSION This.aToDos[This.nToDos]
    n = 1
    SCAN
      This.aToDos[n]=CREATEOBJECT("Todo",
id)
      n = n + 1
    ENDSCAN
    This.CloseTodos()
    RETURN This.nToDos

    PROCEDURE New
    This.nTodos = This.nTodos + 1
    DIMENSION This.aToDos[This.nToDos]
    This.aTodos[This.nTodos] =
CREATEOBJECT("Todo")
    This.aTodos[This.nTodos].Save()
    RETURN This.nToDos

    PROCEDURE CloseToDos
    LPARAMETERS lLeaveOpen
    IF NOT lLeaveOpen
       USE IN SELECT("ToDos")
    ENDIF

    PROCEDURE Complete
    oToDo =CREATEOBJECT("ToDo", ToDos.id)
    oToDo.oData.Completed=.t.
    RETURN oToDo.Save()

    PROCEDURE Delete
    oToDo =CREATEOBJECT("ToDo", ToDos.id)
    RETURN oToDo.Delete()
ENDDEFINE
```

```
* Individual ToDo
DEFINE CLASS ToDo AS Custom
    Name = "ToDo"
    cId = ""
    oData = .null.
    lNew = .f.
    lSaved = .f.
    lLoaded = .f.
    oException = .null.


    PROCEDURE Init
    LPARAMETERS cId
    This.cId = cId
    IF EMPTY(cId)
       This.New()
    ELSE
       This.Load(This.cId)
    ENDIF
    ENDPROC


    PROCEDURE New
    lUsed = This.OpenToDos()
    SCATTER BLANK NAME This.oData MEMO
    This.lNew = .t.
    This.CloseTodos(lUsed)
    RETURN This.oData


    PROCEDURE Load
    LPARAMETERS cId
    LOCAL lUsed
    cId = EVL(cId,This.cId)
    IF NOT EMPTY(cId)
       TRY
          lUsed = This.OpenToDos()
          LOCATE FOR id = cId
          IF FOUND()
             SCATTER NAME This.oData
MEMO
             This.cId = cId
             This.lLoaded = .t.
             This.lNew = .f.
          ENDIF
       CATCH TO oEx
          This.oException = oEx
       FINALLY
          This.CloseTodos(lUsed)
       ENDTRY
    ENDIF
    RETURN This.lLoaded


    PROCEDURE Save
```

```
    LOCAL lUsed
    This.lSaved = .F.
    IF This.lLoaded OR This.lNew
      lUsed = This.OpenToDos()
      TRY
        IF This.lNew
* There are many ways to create a GUID,
including calls to CoCreateGUID in
Ole32.dll, but this is easy. From
https://fox.wikis.com/wc.dll?Wiki~GUIDGen
erationCode~VB
           LOCAL oGUID
           oGUID =
CreateObject("scriptlet.typelib")
           This.oData.Id =
Strextract(oGUID.GUID, "{", "}" )
           This.oData.Entered = DATETIME()
           INSERT INTO ToDos FROM NAME
This.oData
           This.cId = This.oData.Id
        ELSE
           LOCATE FOR id = This.cId
           GATHER NAME This.oData MEMO
        ENDIF
        This.lSaved = .t.
        This.lNew = .f.
    CATCH TO oEx
        This.oException = oEx
    FINALLY
        This.CloseTodos(lUsed)
    ENDTRY
    ENDIF
    RETURN This.lSaved

    PROCEDURE Delete
    LOCAL lUsed, lReturn
    IF NOT EMPTY(This.cId)
        lUsed = This.OpenToDos()
        LOCATE FOR id = This.cId
        lReturn = FOUND()
        IF lReturn
            DELETE
        ENDIF
        This.CloseTodos(lUsed)
    ENDIF
    RETURN lReturn

    PROCEDURE OpenTodos
    LOCAL lUsed
    lUsed = USED("ToDos")
    IF NOT lUsed
        USE data\ToDos IN 0
    ENDIF
    SELECT ToDos
    RETURN lUsed

    PROCEDURE CloseToDos
    LPARAMETERS lLeaveOpen
    IF NOT lLeaveOpen
        USE IN SELECT("ToDos")
    ENDIF

ENDDEFINE && ToDo
```

cntToDo Visual Class – in Code

```
**********************************
*-- Class:          cntToDo
*-- ParentClass:    container
*-- BaseClass:      container
*
DEFINE CLASS cntToDo AS container


   Width = 544
   Height = 114
   BackColor = RGB(255,255,255)
   Name = "cnttodo"


ADD OBJECT imgTask AS image WITH ;
      Picture =
"..\images\ribbon.png", ;
      Height = 43, ;
      Left = 10, ;
      Top = 6, ;
      Width = 37, ;
      Name = "imgTask"


ADD OBJECT txtTitle AS textbox
WITH ;
      FontSize = 18, ;
      Height = 36, ;
      Left = 60, ;
      Top = 9, ;
      Width = 360, ;
      ForeColor = RGB(0,128,192), ;
      Name = "txtTitle"


ADD OBJECT imgCompleted AS image
WITH ;
      Picture =
"..\images\checkmark.png", ;
      Height = 28, ;
      Left = 469, ;
      Top = 12, ;
      Width = 23, ;
      Name = "imgCompleted"


ADD OBJECT imgDelete AS image WITH;
      Picture =
"..\images\delete.png",;
      Height = 20, ;
      Left = 507, ;
      Top = 16, ;
      Width = 14, ;
      Name = "imgDelete"


ADD OBJECT imgEdit AS image WITH ;
      Picture =
"..\images\edit.png", ;
      Height = 20, ;
      Left = 435, ;
      Top = 16, ;
      Visible = .F., ;
      Width = 25, ;
      Name = "imgEdit"


ADD OBJECT edtDescription AS
editbox WITH ;
      FontSize = 14, ;
      Height = 59, ;
      Left = 60, ;
      Top = 48, ;
      Width = 480, ;
      ControlSource = "descript", ;
      NullDisplay = "", ;
      Name = "edtDescription"


ENDDEFINE
*
*-- EndDefine: cnttodo
**********************************
```



附录B：X#类