第49章 VSTO

本章内容：

- 可以用 VSTO 创建的项目类型，在这些项目中可以包含的功能
- 应用于所有 VSTO 解决方案类型的基础技术
- 使用宿主项和宿主控件
- 构建带自定义 UI 的 VSTO 解决方案

Visual Studio Tools for Office(VSTO)技术可以使用.NET Framework 定制和扩展 Microsoft Office 应用程序和文档。它包含的工具还可以使在 Visual Studio 中简化这个自定义过程，例如，用于 Office Ribbon 控件的可视化设计器。

VSTO 是 Microsoft 发布的一系列产品中的最新产品，可以定制 Office 应用程序。用于访问 Office 应用程序的对象模型随时间逐步演化。如果读者过去曾使用过它，就会熟悉它的某些部分。如果读者以前为 Office 应用程序编写过 VBA 插件，就为本章讨论的技术做好了准备。但 VSTO 提供的、便于与 Office 交互的类已经扩展到 Office 对象模型之外。例如，VSTO 类包括.NET 数据绑定功能。

在 Visual Studio 2008 推出之前，VSTO 一直是一个独立下载的软件包，如果要开发 Office 解决方案，就可以得到它。从 Visual Studio 2008 开始，VSTO 集成到 Visual Studio IDE 中。VSTO 的这个版本也称为 VSTO 3，包含对 Office 2007 的全部支持，还包括许多新功能。例如，可以与 Word 内容控件交互，前面提及的 ribbon 可视化设计器等。

在 Visual Studio 2010 中，VSTO 4 扩展并改进了以前的版本，它更便于部署，且不再需要在客户端 PC 上安装主互操作程序集(Primary Interop Assembly，PIA)，这通过 CLR 4 类型嵌入功能实现。还包含对 Office 2010 的支持。

本章不需要 VSTO 或其以前版本的任何预备知识。

49.1 VSTO 概述

VSTO 包含如下组件：

- 一组项目模板，可用于创建各种类型的 Office 解决方案
- 设计器，支持 ribbons、动作面板和自定义任务面板的可视化布局
- 建立在 Office 对象模型基础之上的类，它们还提供扩展功能

VSTO 支持 Office 2003、2007 和 2010。VSTO 类库有多种形式，分别用于这几种 Office 版本，它们分别使用不同集的程序集。为了简单起见，本章主要介绍 2007 版。

VSTO 解决方案的一般体系结构如图 49-1 所示。

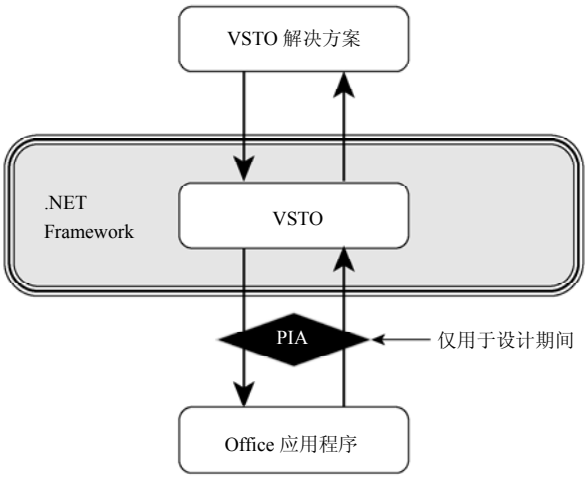

图 49-1

注意，当面向.NET 4 时，在设计期间仅需要如图 49-1 所示的 PIA，当部署到客户端时这用作内嵌的类型。这对于面向.NET 3.5 的VSTO解决方案不成立，此时在开发计算机和目标计算机上都需要PIA。

49.1.1 项目类型

图 49-2 显示了 VS for Office 2007 中的项目模板(Office 2010 中可用的列表与此类似)。本章主要讨论 Office 2007。

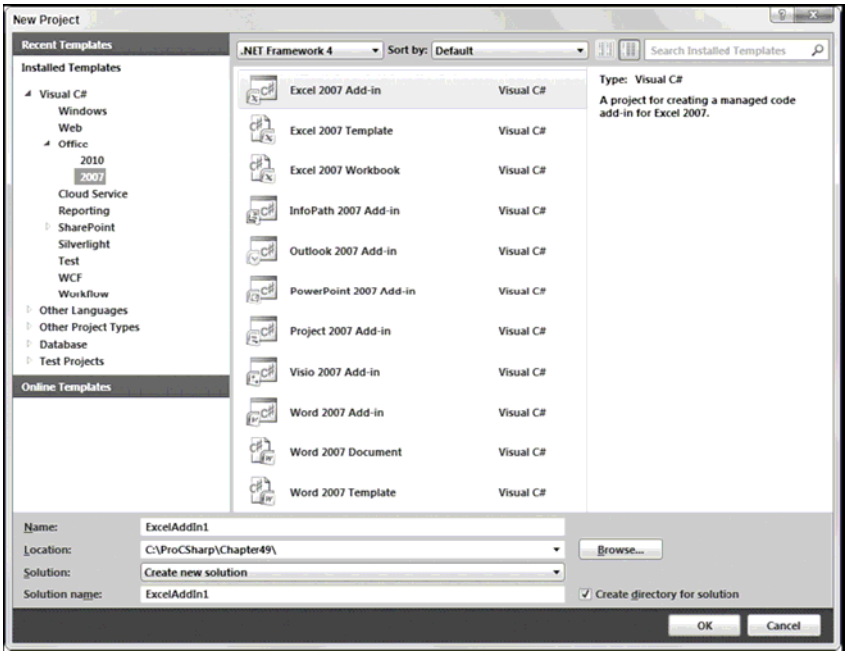

图 49-2

VSTO 项目模板可以分为如下类别：

- 文档级自定义
- 应用程序级的插件
- SharePoint 工作流模板(在图 49-2 中未显示，因为它们位于有关单独的模板类别中)

注意，VSTO 的以前版本包含第 4 类：InfoPath 窗体模板，它在 VSTO 4 中不再可用。

本章主要讨论最常用的项目类型，即文档级自定义和应用程序级的插件。

1. 文档级自定义

创建这种类型的项目时，会生成一个链接到单个文档上的程序集，如 Word 文档、Word 模板或 Excel 工作簿。加载该文档时，关联的 Office 应用程序会检测到该自定义，加载程序集，使 VSTO 定制可用。

这类项目可以给某个特定业务范围的文档提供附加功能，或者在文档模板中添加自定义功能，为整类文档添加附加功能。所包含的代码可以操作文档和文档的内容，包括嵌入对象。还可以提供自定义菜单，包括可以用 Visual Studio Ribbon 设计器创建的功能区菜单。

创建文档级的项目时，可以选择创建新文档，或者复制已有的文档，作为开发的起点。也可以选择要创建的文档类型。例如，对于 Word 文档，就可以选择创建.docx(默认)、.doc 或.docm 文档(.docm 是启用宏的文档)。其对话框如图 49-3 所示。

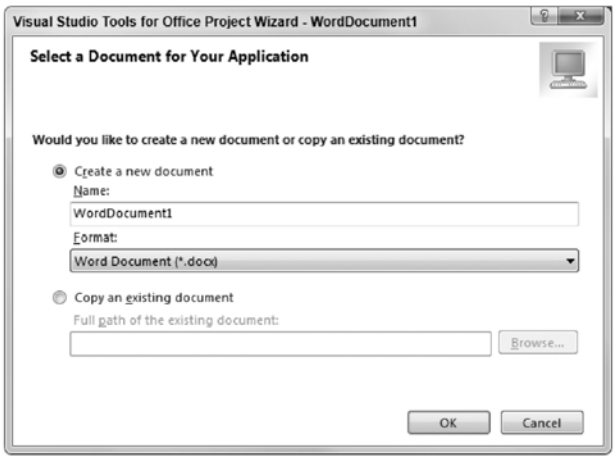

图　49-3

2. 应用程序级的插件

应用程序级的插件不同于文档级的自定义，因为前者可用于整个目标 Office 应用程序。我们可以访问插件代码，而无论加载什么文档，其中都可能包含菜单、文档操作等。

启动某个 Office 应用程序(如 Word)时，它会寻找已在注册表中有注册项的关联插件，并加载它需要的任何程序集。

3. SharePoint 工作流模板

这些项目提供了创建 SharePoint 工作流应用程序的模板。它们用于管理 SharePoint 进程中的文档流。创建这类项目后，就可以在文档的生命周期中，在重要的时刻执行自定义代码。

49.1.2 项目功能

在各种 VSTO 项目类型中有几个可以使用的功能，如交互面板和控件。我们使用的项目类型决定了可用的功能。表 49-1~49-3 根据项目类型列出了这些功能。

表 49-1

功　能	说　明
动作面板	动作面板是驻留在 Word 或 Excel 的动作面板中的对话框。可以在这里显示任意控件，这是扩展文档和应用程序的一种通用方式
数据缓存	数据的缓存可以在文档外部的缓存数据岛上存储在文档中使用的数据。这些数据岛可以从数据源中更新或手工更新，在数据源脱机或不可用时，允许 Office 文档访问数据
VBA 代码的端点	如前所述，VSTO 支持与 VBA 的交互操作。在文档级的自定义中，可以提供从 VBA 代码中调用的端点方法
宿主控件	宿主控件是 Office 对象模型中已有控件的扩展包装器。可以操作这些对象，并把数据绑定到这些对象上
智能标记	智能标记是嵌入在 Office 文档中、有类型化内容的对象。它们在 Office 文档的内容中自动检测，例如，应用程序检测到相应的文本时，就会自动添加股票报价智能标记。可以创建自己的智能标记类型，定义可以在该类标记上执行的操作
可视化文档设计器	处理文档自定义项目时，要使用 Office 对象模型创建一个可视化设计界面，用于交互式地排放控件。设计器中显示的工具栏和菜单(如本章后面所述)功能齐全

表 49-2 列出了应用程序级的插件功能

表 49-2

功　能	说　明
自定义任务面板	任务面板一般停靠在 Office 应用程序的一个边界上，并提供各种功能。例如，Word 的一个任务面板用于操作样式。与动作面板一样，它们也提供了很大的灵活性
跨应用程序的通信	一旦为某个 Office 应用程序创建了插件，就可以把这个功能提供给其他插件。例如，可以在 Excel 中创建一个财务计算服务，再从 Word 中使用该服务——无需创建一个单独的插件
Outlook 窗体区域	可以创建在 Outlook 中使用的窗体区域

表 49-3 列出了所有项目类型可用的功能

表 49-3

功　能	说　明
ClickOnce 部署	可以通过 ClickOnce 部署方法把自己创建的任意 VSTO 项目发布给最终用户，让用户检测对应用程序集清的变化，拥有文档级和应用程序级解决方案的最新版本
功能区菜单	功能区菜单在所有的 Office 应用程序中使用。VSTO 提供了创建自定义功能区菜单的两种方式。可以使用 XML 定义功能区，也可以使用功能区设计器，后者更容易使用，尽管采用 XML 版本可以保证向后兼容性

49.2　VSTO 项目基础

既然知道了 VSTO 包含的内容，就该查看 VSTO 更实用的一面了，并学习如何构建 VSTO 项目。本节介绍的技术可以应用于所有 VSTO 项目类型。

本节介绍如下内容：

- Office 对象模型
- VSTO 名称空间
- 宿主项和宿主控件
- 基本 VSTO 项目结构
- Globals 类
- 事件处理

49.2.1　Office 对象模型

Office 应用程序的 2007 和 2010 套件通过一个 COM 对象模型提供其功能。可以从 VBA 中直接使用这个对象模型，来控制 Office 功能的任意方面。Office 对象模型在 Office 97 中引入，之后有了许多演变，Office 中的功能也有许多改变。

Office 对象模型有大量类，其中一些类在 Office 应用程序的套件中使用，一些类专门用于个别应用程序。例如，Word 2007 对象模型包含一个 Documents 集合，它表示当前加载的对象，每个对象都用一个 Document 对象表示。在 VBA 代码中，可以根据名称或索引访问文档，调用方法对它们执行操作。例如，下面的 VBA 代码关闭名称为 My Document 的文档，且不保存修改的内容：

```
Documents("My Document").Close SaveChanges:= wdDoNotSaveChanges
```

Office 对象模型包含命名常量(如上述代码中的 wdDoNotSaveChanges)和枚举，更便于使用。

49.2.2　VSTO 名称空间

VSTO 包含一个名称空间集合，该集合包含的类型可用于给 Office 对象模型编写程序。这些名称空间中的许多类型直接映射到 Office 对象模型中的类型上。可以通过 Office PIA 在设计期间访问它们，在部署解决方案时则通过嵌入的类型信息来访问。由于嵌入的类型，因此主要使用用于访问 Office 对象模型的接口。VSTO 还包含不能直接映射的类型，或者与 Office 对象模型无关的类型。例如，有许多类用于 Visual Studio 中支持的设计器。

包装 Office 对象模型中的对象或与它们通信的类型分别放在不同的名称空间中。用于 Office 开发的名称空间如表 49-4 所示。

表　49-2

名 称 空 间	说　　明
Microsoft.Office.Core Microsoft.Office.Interop.*	因为这些名称空间包含 Office 对象模型的接口和瘦包装器，所以提供了处理 Office 类的基本功能。在 Microsoft.Office.Interop 名称空间中有几个嵌套的名称空间，用于每个 Office 产品
Microsoft.Office.Tools	这个名称空间包含的通用类型提供了 VSTO 功能和用于嵌套名称空间中的许多类的基类。例如，这个名称空间包含实现文档级自定义中的动作面板所需的类，以及应用程序级插件的基类

(续表)

名 称 空 间	说　　明
Microsoft.Office.Tools.Excel Microsoft.Office.Tools.Excel.*	这些名称空间包含的类型用于与 Excel 应用程序和 Excel 文档交互
Microsoft.Office.Tools.Outlook	这个名称空间包含的类型用于与 Outlook 应用程序交互
Microsoft.Office.Tools.Ribbon	这个名称空间包含的类型用于处理和创建功能区菜单
Microsoft.Office.Tools.Word Microsoft.Office.Tools.Word.*	这些名称空间包含的类型用于与 Word 应用程序和 Word 文档交互
Microsoft.VisualStudio.Tools.*	这些名称空间提供的 VSTO 基础结构可以在 Visual Studio 中开发 VSTO 解决方案时使用

49.2.3　宿主项和宿主控件

宿主项和宿主控件是经过扩展的接口，使文档级的自定义更容易与 Office 文档交互。这些接口简化了代码，因为它们提供了.NET 样式的事件，且进行了全面地管理。宿主项和宿主控件中的“宿主”表示，这些接口封装和扩展了本地 Office 对象。

在使用宿主项和宿主控件时，也需要使用底层的交互操作类型。例如，如果创建了一个新的 Word 文档，就会接收到对交互操作 Word 文档类型的引用，而不是 Word 文档宿主项。必须注意这一点，并据此编写代码。

Word 和 Excel 文档级自定义都有宿主项和宿主控件。

1. Word

Word 只有一个宿主项 Microsoft.Office.Tools.Word.Document。这表示一个 Word 文档。果然，这个接口有许多方法和属性，可用于与 Word 文档交互。

Word 有 12 个宿主控件，如表 49-5 所示，所有宿主控件都在 Microsoft.Office.Tools.Word 名称空间中。

表　49-5

控　　件	说　　明
Bookmark	这个控件表示 Word 文档中的一个位置，它可以是单个位置，或一个字符范围
XMLNode、XmLNodes	当文档有一个附加的 XML 架构时使用这两个控件，它们允许通过文档内容的 XML 节点位置来引用文档内容。也可以用这两个控件操作文档的 XML 结构
ContentControl	这个接口与本表中剩余 8 个控件有相同的基接口(ContentControlBase)，允许处理 Word 内容控件。内容控件把内容表示为控件，或者启用文档中纯文本没有提供的功能
BuildingBlockGallery-ContentControl	这个控件允许添加和处理文档构建模块，如格式化的表、封面等
ComboBoxContentControl	这个控件表示格式化为组合框的内容
DatePickerContentControl	这个控件表示格式化为日期拾取器的内容
DropDownListContentControl	这个控件表示格式化为下拉列表的内容

(续表)

控　件	说　明
GroupContentControl	这个控件表示的内容是其他内容项的分组集合，包括文本和其他内容控件
PictureContentControl	这个控件表示一幅图像
RickTextContentControl	这个控件表示一块格式文本内容
PlainTextContentControl	这个控件表示一块纯文本内容

2. Excel

Excel 有 3 个宿主项和 4 个宿主控件，它们都包含在 Microsoft.Office.Tools.Excel 名称空间中。Excel 宿主项如表 49-6 所示。

表　49-6

主 机 项	说　明
Workbook	这个宿主项表示整个 Excel 工作簿，它可以包含多个工作表和图表
Worksheet	这个宿主项用于工作簿中的单个工作表
Chartsheet	这个宿主项用于工作簿中的单个图表

Excel 宿主控件如表 49-7 所示。

表　49-7

控　件	说　明
Chart	这个控件表示嵌入到工作表中的图表
ListObject	这个控件表示工作表中的一个列表
NamedRange	这个控件表示工作表中的一个命名区域
XmlMappedRange	当 Excel 电子表格有附加的架构时使用这个控件，它用于处理映射到 XML 架构元素上的范围

49.2.4　基本的 VSTO 项目结构

第一次创建 VSTO 项目时，系统创建的文件随项目类型的不同而不同，但有一些共同的功能。本节介绍 VSTO 项目的组成。

1. 文档级自定义项目结构

创建文档级自定义项目时，在 Solution Explorer 窗口中有一项表示文档类型。它可以是：

- 表示 Word 文档的.docx 文件
- 表示 Word 模板的.dotx 文件
- 表示 Excel 工作簿的.xlsx 文件
- 表示 Excel 模板的.xltx 文件

每个文档类型都有一个设计器视图和一个代码文件，如果在 Solution Explorer 窗口中展开该项，就会看到它们。Excel 模板还包含子项，它们表示整个工作簿和工作簿中的每个工作表。这个结构可以在每个工作表或工作簿的基础上提供自定义功能。

如果查看上述项目类型的隐藏文件，就会看到几个设计器文件，查看这些设计器文件，还会看到模板生成的代码。每个 Office 文档项都在 VSTO 名称空间中有关联的类，代码文件中的类派生自这些类。这些类定义为部分类，这样自定义代码会与可视化设计器生成的代码分隔开，类似于 Windows 窗体应用程序的结构。

例如，Word 文档模板提供了一个派生自 Microsoft.Office.Tools.Word.Document 宿主项的类。这个类通过 Base 属性提供 Document 宿主项，这个类包含在 ThisDocument.cs 中，如下所示：

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Xml.Linq;
using Microsoft.VisualStudio.Tools.Applications.Runtime;
using Office = Microsoft.Office.Core;
using Word = Microsoft.Office.Interop.Word;

namespace WordDocument1
{
    public partial class ThisDocument
    {
        private void ThisDocument_Startup(object sender, System.EventArgs e)
        {
        }
            private void ThisDocument_Shutdown(object sender, System.EventArgs e)
            {
            }
            #region VSTO Designer generated code
            /// <summary>
            /// Required method for Designer support - do not modify
            /// the contents of this method with the code editor.
            /// </summary>
            private void InternalStartup()
            {
                this.Startup += new System.EventHandler(ThisDocument_Startup);
                this.Shutdown += new System.EventHandler(ThisDocument_Shutdown);
            }
            #endregion
    }
}
```

这些模板生成的代码包含两个主要名称空间的别名，在为 Word 创建文档级自定义时，需要使用这两个名称空间。Microsoft.Office.Core 用于主要的 VSTO Office 类，Microsoft.Office.Interop.Word 用于 Word 专用的类。注意如果要使用 Word 宿主控件，那么还要为 Microsoft.Office.Tools.Word 名称空间添加一条 using 语句。模板生成的代码还定义两个事件处理程序挂钩——ThisDocument_Startup() 和 ThisDocument_Shutdown()，用于在加载或卸载文档时执行代码。

每个文档级自定义项目类型的代码文件(或者，对于 Excel 文件或代码文件)都有类似的结构，还定义了名称空间别名以及 VSTO 类中各个 Startup 和 Shutdown 事件的处理程序。以此为起点，可以添加对话框、动作面板、ribbon 控件、事件处理程序和自定义代码，来定义自定义行为。

在文档级自定义中，还可以通过文档设计器定制文档。根据所创建的解决方案类型，这可能需要给模板添加样板文件，给文档添加交互式内容或其他内容。设计器实际上是 Office 应用程序的宿主版本，使用它们可以像在应用程序中那样输入内容。然而，还可以在文档中添加控件，如宿主控件和 Windows 窗体控件，以及这些控件的代码。

2. 应用程序级插件的项目结构

当创建应用程序级插件时，在 Solution Explorer 窗口中没有文档。而有一项表示创建插件所使用的应用程序，如果展开该项，会看到一个 ThisAddIn.cs 文件。这个文件包含 ThisAddIn 类的一个部分类定义，该类是插件的入口点。这个类派生自 Microsoft.Office.Tools.AddIn Base，它提供了实现插件功能的代码。

例如，Word 插件模板生成的代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;
using Word = Microsoft.Office.Interop.Word;
using Office = Microsoft.Office.Core;
using Microsoft.Office.Tools.Word;
namespace WordAddIn1
{
   public partial class ThisAddIn
   {
      private void ThisAddIn_Startup(object sender, System.EventArgs e)
      {
      }
      private void ThisAddIn_Shutdown(object sender, System.EventArgs e)
      {
      }
      #region VSTO generated code
      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InternalStartup()
      {
         this.Startup += new System.EventHandler(ThisAddIn_Startup);
         this.Shutdown += new System.EventHandler(ThisAddIn_Shutdown);
      }
      #endregion
   }
}
```

可以看出，这个结构非常类似于文档级自定义使用的结构。它包含相同 Microsoft.Office.Core 和 Microsoft.Office.Interop.Word 名称空间的别名，并提供 Startup 和 Shutdown 事件的事件处理程序 (ThisAddIn_Startup()和 ThisAddIn_Shutdown())。这些事件与文档级自定义略有不同，因为它们在加载或卸载插件时引发，而不是在打开或关闭个别文档时引发。

定制应用程序级插件与文档级自定义相同：添加 ribbon 控件、任务面板和其他代码。

49.2.5 Globals 类

所有 VSTO 项目类型都定义一个 Globals 类，它提供了如下内容的全局访问权限：

- 对于文档级自定义，可以访问解决方案中的所有文档。这通过其名称匹配文档类名的成员来提供，如 Globals.ThisWorkbook 和 Globals.Sheet1。
- 对于应用程序级的插件，可以访问插件对象。这通过 Globals.ThisAddIn 提供。
- 对于 Outlook 插件项目，可以访问所有 Outlook 窗体区域。
- 通过 Globals.Ribbons 属性，可以访问解决方案中的所有 ribbon 控件。
- 派生自 Microsoft.Office.Tools.Factory 的接口提供项目类型专用的实用程序方法，通过 Factory 属性可访问这些方法。

在后台，在解决方案中由设计器维护的各种代码文件中通过一系列部分定义来创建 Globals 类。例如，在 Excel 工作簿项目中，默认的 Sheet1 工作表包含设计器生成的如下代码：

```
internal sealed partial class Globals
{
   private static Sheet1 _Sheet1;
   internal static Sheet1 Sheet1
   {
      get
      {
         return _Sheet1;
      }
         set
         {
            if ((_Sheet1 == null))
            {
               _Sheet1 = value;
            }
            else
            {
               throw new System.NotSupportedException();
            }
         }
      }
   }
```

这段代码把 Sheet1 成员添加到 Globals 类中。

Globals 类还允许使用 Globals.Factory.GetVstoObject()方法把交互操作类型转换为 VSTO 类型。例如，这会从 Microsoft.Office.Interop.Excel.Workbook 接口中得到 VSTO Workbook 接口。另外，Globals.Factory.HasVstoObject()方法可用于确定这种转换是否可行。

49.2.6 事件处理

本章前面介绍了宿主项和宿主控件类如何提供我们可以处理的事件。但交互操作类不是这样。我们只能使用几个事件，大多数事件都很难用于创建事件驱动的解决方案。为了响应事件，我们常常要关注宿主项和宿主控件提供的事件。

此处一个明显的问题是应用程序级的插件项目没有宿主项和宿主控件。在使用 VSTO 时，必须面对这个问题。但是，我们在插件中侦听的大多数常用事件都关联到功能区菜单和任务面板的交互操作

中。我们用集成的功能区设计器设计功能区控件，响应功能区控件生成的事件，使控件可以交互操作。任务面板常常作为 Windows 窗体用户控件实现(尽管也可以使用 WPF)，这里可以使用 Windows 窗体事件。这表示，我们不会常常遇到如下情形：需要的功能没有可用的事件。

当需要使用 Office 交互操作对象提供的事件时，通过这些对象上的接口提供这些事件。考虑一个 Word 插件项目。这个项目中的 ThisAddIn 类有一个 Application 属性，利用该属性可以获得对 Office 应用程序的引用。这个属性的类型是 Microsoft.Office.Interop.Word.Application，它通过 Microsoft.Office.Interop.Word.ApplicationEvents4_Event 接口提供事件。这个接口共提供 29 个事件(对于像 Word 这样复杂的应用程序 29 个事件并不多)。例如，我们可以处理 DocumentBeforeClose 事件，来响应 Word 文档的关闭请求。

49.3　构建 VSTO 解决方案

前面几节解释了 VSTO 项目的概念、结构和可以在各种项目类型中使用的功能。本节讨论如何实现 VSTO 解决方案。

图 49-4 列出了文档级自定义解决方案的结构。

对于文档级自定义，至少要与一个宿主项交互操作，该宿主项一般包含多个宿主控件。可以直接使用 Office 对象包装器，但在大多数情况下，应通过宿主项和宿主控件访问 Office 对象模型及其功能。

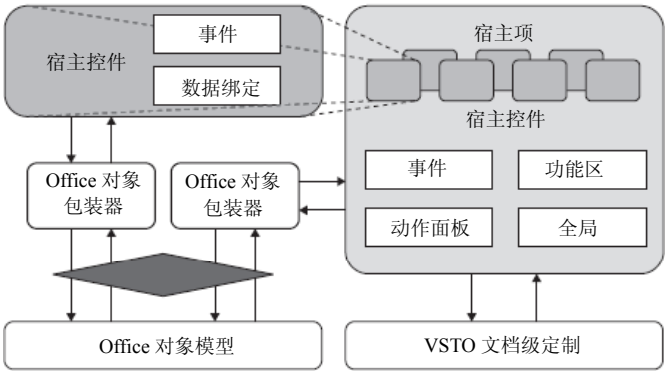

图　49-4

可以在代码中使用宿主项和宿主控件事件、数据绑定、功能区菜单、动作面板和全局对象。

图 49-5 列出了应用程序级插件解决方案的结构。

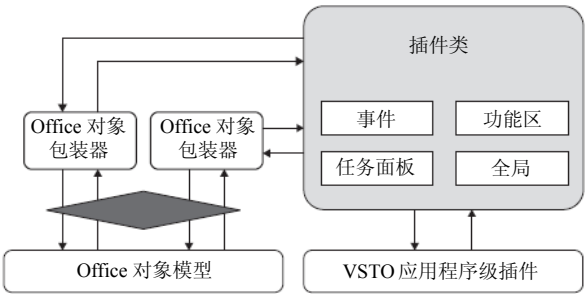

图　49-5

在这个略微简单的模型中，很可能要直接使用 Office 对象的瘦包装器，或者至少通过封装解决方案的插件类来使用。还要在代码中使用插件类提供的事件、功能区菜单、动作面板和全局对象。

本节介绍这两种应用程序类型以及如下主题：

- 管理应用程序级插件
- 与应用程序和文档交互操作
- UI 自定义

49.3.1 管理应用程序级插件

在创建应用程序级插件时，会发现 Visual Studio 执行了通过 Office 应用程序注册插件所需的所有步骤。这表示，添加注册表项，这样在 Office 应用程序启动时，它会自动定位和加载程序集。如果以后要添加或删除插件，就必须导航 Office 应用程序设置，或者手工操作注册表。

例如，在 Word 中，必须打开 Office Button 菜单，单击 Word Options，选择 Add-Ins 选项卡，如图 49-6 所示。

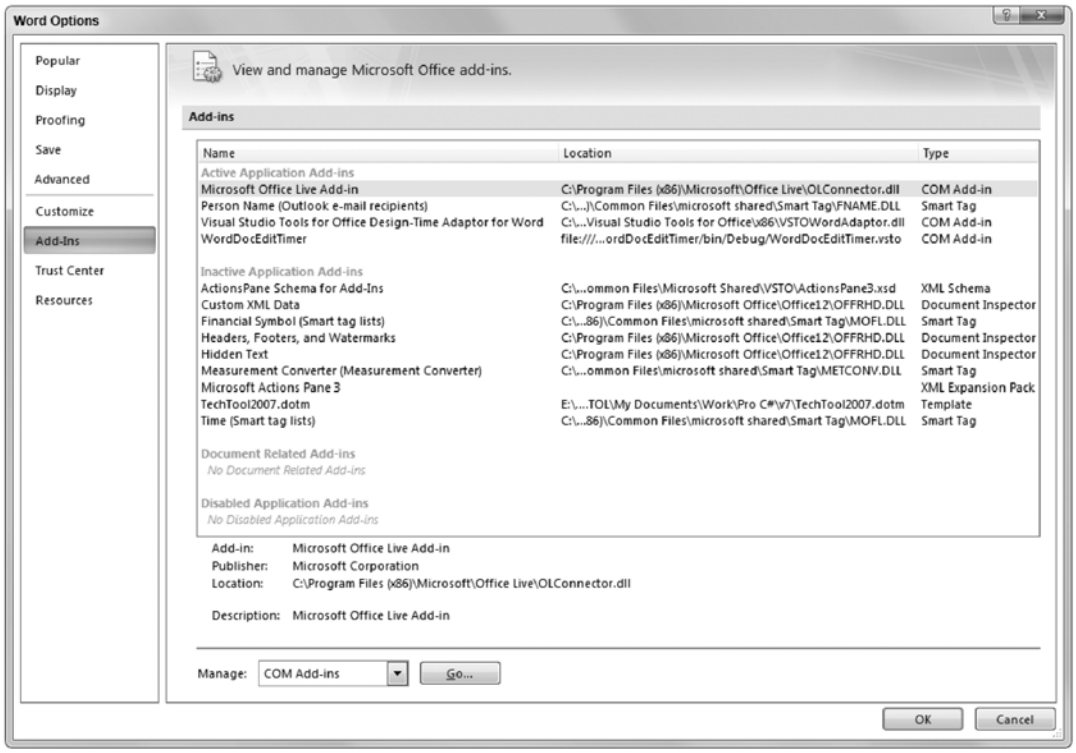

图 49-6

图 49-6 显示了用 VSTO 创建的一个插件：WordDocEditTimer。要添加或删除插件，必须在 Manage 下拉列表中选择 COM Add-Ins(默认选项)，单击 Go 按钮，打开如图 49-7 所示的对话框。

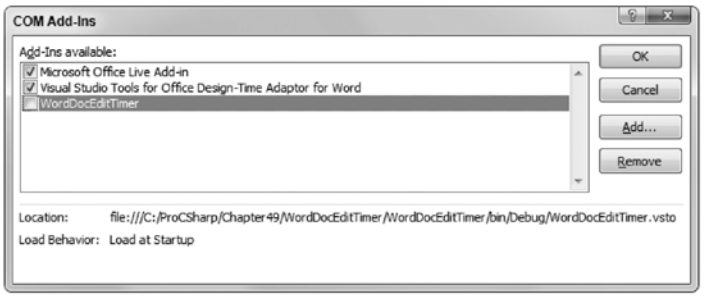

图　49-7

在 COM Add-Ins 对话框中取消选择插件，就会卸载插件，如图 49-7 所示。还可以使用 Add 和 Remove 按钮添加新插件或删除旧插件。

49.3.2　与应用程序和文档交互操作

无论创建什么类型的应用程序，都要与宿主应用程序和/或宿主操作中的文档进行交互。这包括使用下一节介绍的 UI 自定义功能。还可能需要监控应用程序中的文档，这表示必须处理一些 Office 对象模型事件。例如，要监控 Word 中的文档，需要 Microsoft.Office.Interop.Word.ApplicationEvents4_Event 接口中如下事件的事件处理程序：

- DocumentOpen——打开文档时引发
- NewDocument——创建新文档时引发
- DocumentBeforeClose——保存文档时引发

另外，Word 第一次启动时，它会加载一个文档，它可以是空白的新文档，也可以是已加载的旧文档。

本章的可下载代码包含一个 WordDocEditTimer 示例，它维护 Word 文档的一个编辑次数表。这个应用程序的部分功能是监控已加载的文档，其原因在后面解释。因为这个示例也使用自定义任务面板和 ribbon 菜单，所以在介绍完这些主题后，再介绍这个示例。

在 Word 中，通过 ThisAddIn.Application.ActivateDocument 属性可以访问当前活动的文档，通过 ThisAddIn.Application.Documents 属性访问已打开的文档集合。由于有了多文档界面(Multiple Document Interface，MDI)，类似的属性也存在于其他 Office 应用程序中。例如，可以通过 Microsoft.Office.Interop. Word.Document 类的属性操作文档的各个属性。

这里要注意，在开发 VSTO 解决方案时，必须处理的类和类成员的数量相当大。除非已经习惯了，否则很难找到需要的功能。例如，在 Word 中，当前活动选区不是通过活动文档获得的，而是通过应用程序获得的(利用 ThisAddIn.Application.Selection 属性)，其原因并不是很明显。

通过 Range 属性可以把选区应用于插入、读取或替换文本的操作。例如：

```
ThisAddIn.Application.Selection.Range.Text = "Inserted text";
```

但是，本章没有足够的篇幅来详细介绍对象库，而读者可以在本章讨论相关的内容时学习对象库。

49.3.3 UI 的自定义

在 VSTO 的最新版本中，最重要的方面是可用于定制用户的自定义 UI 和插件的灵活性。可以给已有的功能区菜单中添加内容，添加全新的功能区菜单，通过添加动作面板定制动作面板，添加全新的动作面板，集成 Windows 窗体、WPF 窗体和控件。

本节介绍这些主题。

1. 功能区菜单

可以在本章介绍的所有 VSTO 项目中添加功能区菜单。添加功能区菜单时，会看到如图 49-8 所示的设计器窗口。

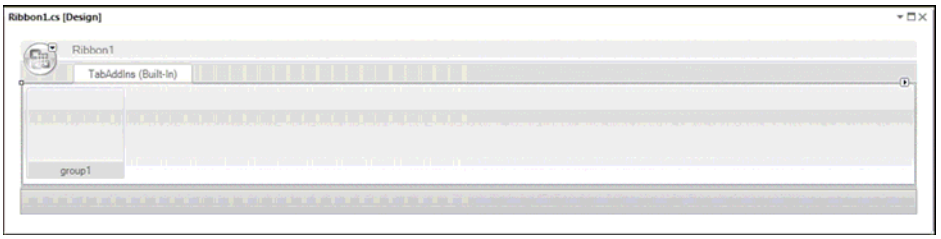

图 49-8

设计器可以给 Office 按钮菜单和功能区菜单上的组添加控件(显示在图 49-8 的左上部)，来定制这个功能区菜单。也可以添加其他组。

功能区中使用的类在 Microsoft.Office.Tools.Ribbon 名称空间中。这包括用于创建功能区的派生 ribbon 类 OfficeRibbon。这个类可以包含 RibbonTab 对象，每个 RibbonTab 对象都包含单个选项卡的内容。选项卡则包含 RibbonGroup 对象，类似图 49-8 中的 group1 组。这些选项卡都可以包含各种控件。

选项卡上的组可以位于目标 Office 应用程序的一个全新选项卡上，或者位于其中一个已有的选项卡上。组在何处显示取决于 RibbonTab.ControlId 属性。这个属性有一个 ControlIdType 属性，它可以设置为 RibbonControlIdType.Custom 或 RibbonControlIdType.Office。如果使用 Custom，那么还必须把 RibbonTab.ControlId.CustomId 设置为 String 值，这是选项卡的标识符。这里可以使用任意标识符。但如果给 ControlIdType 属性使用 Office，就必须把 RibbonTab.ControlId.OfficeId 设置为一个 String 值，该值匹配在当前 Office 产品中使用的其中一个标识符。例如，在 Excel 中，把这个属性设置为 TabHome，可以把组添加到 Home 选项卡中；设置为 TabInsert，可以把组添加到 Insert 选项卡中等。插件的默认属性是 TabAddIns，它由所有插件共享。

许多选项卡可用，尤其在 Outlook 中；可以从 www.microsoft.com/downloads/details.aspx?FamilyID=4329D9E9-4D11-46A5-898D-23E4F331E9AE&displaylang=en 中下载包含完整列表的一系列电子表格。

一旦决定在何处放置 ribbon 组后，就可以添加如表 49-9 所示的控件。

表 49-8

控 件	说 明
RibbonBox	这是一个容器控件，它可用于排放组中的其他控件。可以把 BoxStyle 属性改为 RibbonBoxStyle.Horizontal 或 RibbonBoxStyle.Vertical，在 RibbonBox 中水平或垂直排放控件
RibbonButton	这个控件可用于在组中添加大按钮或小按钮，在按钮的旁边可以有或没有文本标签。把 ControlSize 属性设置为 RibbonControlSize.RibbonControlSizeLarge 或 RibbonControlSize.RibbonControlSizeRegular，就可以控制大小。按钮的 Click 事件处理程序可用于响应交互操作。还可以设置自定义图像或存储在 Office 系统中的其中一幅图像(详见本表后面的内容)
RibbonButtonGroup	这是一个容器控件，它表示一组按钮。它可以包含 RibbonButton、RibbonGallery、RibbonMenu、RibbonSplitButton 和 RibbonToggleButton 控件
RibbonCheckBox	复选框控件，有 Click 事件和 Checked 属性
RibbonComboBox	组合框(合并了文本项和下拉列表项)。列表项使用 Items 属性，输入的文本使用 Text 属性，TextChanged 事件用于响应文本更改
RibbonDropDown	这个容器可以包含 RibbonDropDownItem 和 RibbonButton 列表项，它们分别在 Items 和 Buttons 属性中指定。按钮和列表项格式化到下拉列表中。使用 SelectionChanged 事件响应交互操作
RibbonEditBox	文本框，用户可用于输入或编辑 Text 属性中的文本。这个控件有 TextChanged 事件
RibbonGallery	与 RibbonDropDown 控件相同，这个控件也可以包含 RibbonDropDownItem 和 RibbonButton 项，它们分别在 Items 和 Buttons 属性中指定。这个控件使用 Click 和 ButtonClick 事件，来替代 RibbonDropDown 控件的 SelectionChanged 事件
RibbonLabel	显示简单的文本，用 Label 属性设置
RibbonMenu	弹出菜单，在设计视图中打开它时，可以用其他控件填充该菜单，如 RibbonButton 和嵌套的 RibbonMenu 控件。处理菜单上的菜单项的事件
RibbonSeparator	一个简单的分隔符，用于定制组中的控件布局
RibbonSplitButton	合并了 RibbonButton 或 RibbonToggleButton 和 RibbonMenu 的控件。用 ButtonType 设置按钮的样式，该属性可以是 RibbonButtonType.Button 或 RibbonButtonType.ToggleButton。使用主按钮的 Click 事件或菜单中各按钮的 Click 事件来响应交互操作
RibbonToggleButton	一个按钮，可以处于选中或未选中状态，用 Checked 属性指定。这个控件也有 Click 事件

也可以设置组的 DialogBoxLauncher 属性，以便把一个图标显示在组的右下部。顾名思义，使用这个属性可以显示一个对话框，或者打开一个任务面板，或者执行任何其他操作。通过 GroupView Tasks 菜单可以添加或删除这个图标，如图 49-9 所示，该图还显示了表 49-8 中的其他一些控件，因为它们在设计视图中显示在功能区上。

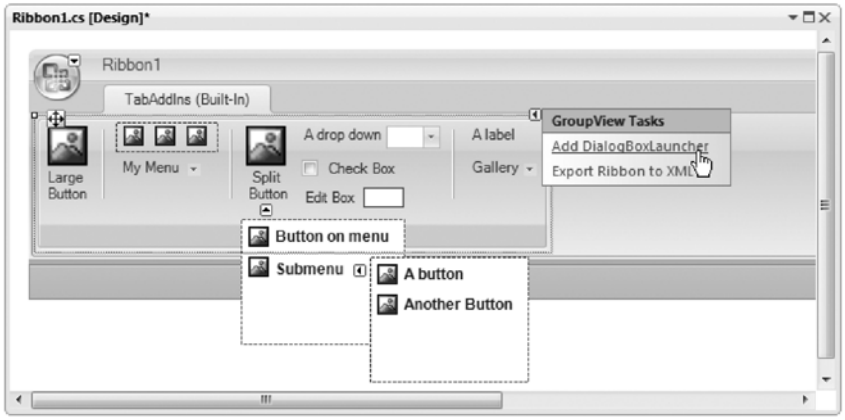

图 49-9

要给控件设置图像，如给 RibbonButton 控件设置图像，就可以把 Image 属性设置为自定义图像，把 ImageName 设置为图像名(以便在 OfficeRibbon.LoadImage 事件处理程序中优化图像的加载)，也可以使用内置的 Office 图像。为此，应把 OfficeImageId 属性设置为图像的 ID。

可以使用许多图像；还可以从 www.microsoft.com/downloads/details.aspx?familyid=12b99325-93e8-4ed4-8385-74d0f7661318&displaylang=en 上下载列出这些图像的电子表格。图 49-10 显示了一个示例。

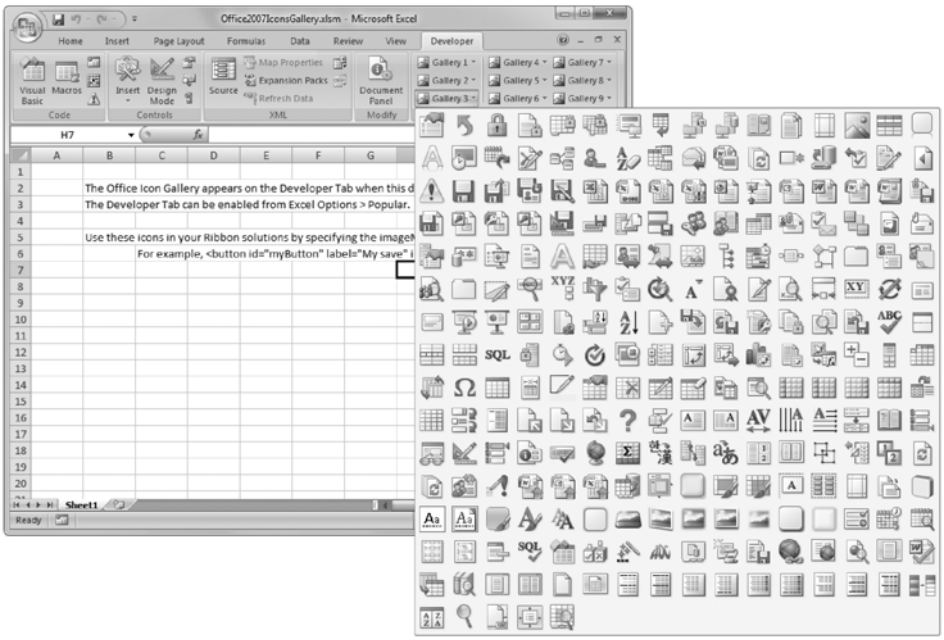

图 49-10

图 49-10 显示了 Developer 功能区选项卡，通过 Popular 选项卡上的 Excel Options 对话框中的 Office 按钮可以启用它。

单击一幅图像，就会打开一个对话框，指出该图像的 ID 是什么，如图 49-11 所示。

图　49-11

功能区设计器非常灵活，还可以提供希望出现在 Office 功能区上的许多额外功能。但是，如果要进一步定制 UI，就要使用动作面板和任务面板，因为可以通过它们创建任意 UI 和功能。

2. 动作面板和自定义任务面板

使用动作面板和任务面板可以显示停靠在 Office 应用程序界面的任务面板区域中的内容。任务面板在应用程序级的插件中使用，动作面板在文档级自定义中使用。任务面板和动作面板都必须继承自 UserControl 对象，这表示应使用 Windows 窗体创建一个 UI。如果把 WPF 窗体驻留在 UserControl 的 ElementHost 控件上，那么还可以使用 WPF UI。

要把动作面板添加到文档级自定义中的一个文档中，应把动作面板类的一个实例添加到文档的 ActionsPane 属性的 Controls 集合中。例如：

```
public partial class ThisWorkbook
{
   private UserControl1 actionsPane;
   private void ThisWorkbook_Startup(object sender, System.EventArgs e)
   {
      Workbook wb = Globals.Factory.GetVstoObject(this.Application.ActiveWorkbook);
       wb.AcceptAllChanges();
       actionsPane = new UserControl1();
       this.ActionsPane.Controls.Add(actionsPane);
   }
   ...
}
```

这段代码在加载文档(这里是 Excel 工作簿)时添加了动作面板。也可以在 ribbon 按钮事件处理程序中添加动作面板。

在应用程序级的插件项目中，自定义任务面板通过 ThisAddIns.CustomTaskPanes.Add()方法属性添加。这个方法也允许命名任务窗口，例如：

```
public partial class ThisAddIn
{
   Microsoft.Office.Tools.CustomTaskPane taskPane;
   private void ThisAddIn_Startup(object sender, System.EventArgs e)
   {
       taskPane = this.CustomTaskPanes.Add(new UserControl1(), "My Task Pane");
       taskPane.Visible = true;
   }
   ...
}
```

注意 Add()方法返回一个 Microsoft.Office.Tools.CustomTaskPane 类型的对象。可以通过这个对象的 Control 属性访问用户控件本身。还可以使用这个类型的其他属性，例如，上面代码中的 Visible 属性，来控制任务面板。

此时，应注意 Office 应用程序的一个不太寻常的功能，尤其是 Word 和 Excel 之间的区别。由于历史原因，尽管 Word 和 Excel 都是 MDI 应用程序，但这两个应用程序驻留文档的方式不同。在 Word 中，每个文档都有一个唯一的父窗口。而在 Excel 中，每个文档都共享同一个父窗口。

在调用 CustomTaskPanes.Add()方法时，默认行为是把任务面板添加到当前的活动窗口中。在 Excel 中，这表示每个文档都显示该任务面板，因为它们都使用同一个父窗口。而在 Word 中，情况有所不同。如果希望任务面板显示给每个文档，就必须把它添加到包含文档的每个窗口中。

要把任务面板添加到特定的文档中，应给 Add()方法传递 Microsoft.Office.Interop.Word.Windows 类的一个实例，作为第 3 个参数。通过 Microsoft.Office.Interop.Word.Document.ActiveWindow 属性可以获得与文档关联的窗口。

下一节介绍如何完成这个操作。

49.4 示例应用程序

如前所述，本章的示例代码包含一个 WordDocEditTimer 应用程序，它维护 Word 文档的一个编辑次数列表。本节将详细解释这个应用程序的代码，因为该应用程序揭示了前面介绍的所有内容，还包含一些有用的提示。

这个应用程序的一般操作是只要创建或加载文档，就启动一个链接到文档名上的计时器。如果关闭文档，该文档的计时器就暂停。如果打开以前计时的文档，计时器就恢复。另外，如果使用 Save As 把文档另存为另一个文件名，计时器就更新为使用新文件名。

这个应用程序是一个 Word 应用程序级的插件，使用一个自定义任务面板和一个功能区菜单。功能区菜单包含一个按钮和一个复选框，按钮用于打开和关闭任务面板，复选框用于暂停当前的活动文档的计时器。包含这些控件的组追加到 Home 功能区选项卡的最后。任务面板显示一个活动计时器列表。

这个用户界面如图 49-12 所示。

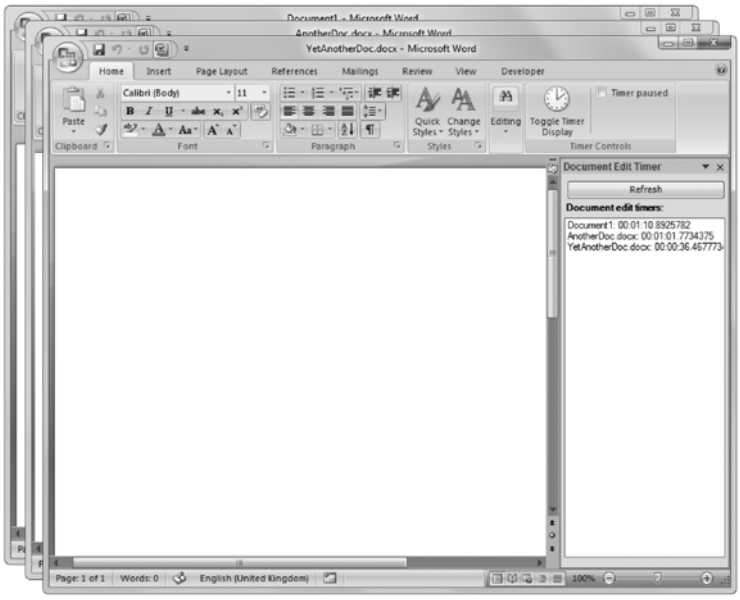

图 49-12

计时器通过 DocumentTimer 类来维护：


```
public class DocumentTimer
{
   public Word.Document Document { get; set; }
   public DateTime LastActive { get; set; }
   public bool IsActive { get; set; }
   public TimeSpan EditTime { get; set; }
}
```

代码段 DocumentTimer.cs

这段代码保存了对 Microsoft.Office.Interop.Word.Document 接口的一个引用、总编辑时间、计时器是否激活，以及它上一次激活的时间。ThisAddIn 类维护这些对象的一个集合，这些对象与文档名关联起来：


```
public partial class ThisAddIn
{
   private Dictionary<string, DocumentTimer> documentEditTimes;
```

代码段 DocumentTimer.cs

因此，每个计时器都可以通过文档引用或文档名来定位。这是必要的，因为文档引用可以跟踪文档名的变化(这里没有可用于监控文档名变化的事件)，文档名允许跟踪已关闭和再次打开的文档。

ThisAddIn 类还维护一个 CustomTaskPane 对象列表(如前所述，Word 中的每个窗口都需要一个 CustomTaskPane 对象)：

```
private List<Tools.CustomTaskPane> timerDisplayPanes;
```

插件启动时，ThisAddIn_Startup()方法执行几个任务。首先它初始化两个集合：

```
private void ThisAddIn_Startup(object sender, System.EventArgs e)
{
   // Initialize timers and display panels
   documentEditTimes = new Dictionary<string, DocumentTimer>();
   timerDisplayPanes = new List<Microsoft.Office.Tools.CustomTaskPane>();
```

接着通过 ApplicationEvents4_Event 接口添加几个事件处理程序：

```
   // Add event handlers
   Word.ApplicationEvents4_Event eventInterface = this.Application;
   eventInterface.DocumentOpen += new Microsoft.Office.Interop.Word
      .ApplicationEvents4_DocumentOpenEventHandler(
         eventInterface_DocumentOpen);
   eventInterface.NewDocument += new Microsoft.Office.Interop.Word
      .ApplicationEvents4_NewDocumentEventHandler(
         eventInterface_NewDocument);
   eventInterface.DocumentBeforeClose += new Microsoft.Office.Interop.Word
      .ApplicationEvents4_DocumentBeforeCloseEventHandler(
         eventInterface_DocumentBeforeClose);
   eventInterface.WindowActivate += new Microsoft.Office.Interop.Word
      .ApplicationEvents4_WindowActivateEventHandler(
```

```
            eventInterface_WindowActivate);
```

这些事件处理程序用于监控文档的打开、创建和关闭，并确保功能区上的 Pause 复选框保持最新状态。后一个功能使用 WindowsActivate 事件跟踪窗口的激活状态来实现。

在这个事件处理程序中，最后一个任务是开始监控当前文档，把自定义任务面板添加到包含文档的窗口中：

```
  // Start monitoring active document
  MonitorDocument(this.Application.ActiveDocument);
  AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);
}
```

MonitorDocument()实用程序方法为文档添加一个计时器：

```
internal void MonitorDocument(Word.Document Doc)
{
   // Monitor doc
   documentEditTimes.Add(Doc.Name, new DocumentTimer
   {
      Document = Doc,
      EditTime = new TimeSpan(0),
      IsActive = true,
      LastActive = DateTime.Now
   });
}
```

这个方法仅为文档创建了一个新的 DocumentTimer 对象。该 DocumentTimer 引用文档，其编辑次数是 0，处于激活状态，且在当前时间激活。接着把这个计时器添加到 documentEditTimes 集合中，并把它关联到文档名上。

AddTaskPaneToWindow()方法把自定义任务面板添加到窗口中。这个方法首先检查已有的任务面板，确保窗口中还没有一个任务面板。此外，Word 中的另一个奇怪的功能是如果在加载应用程序后，立即打开一个旧文档，默认的 Document1 文档就会消失，且不引发关闭事件。因为在文档中访问包含任务面板的文档窗口时，这可能导致引发异常，所以该方法还检查表示该异常的 ArgumentNullException：

```
private void AddTaskPaneToWindow(Word.Window Wn)
{
   // Check for task pane in window
   Tools.CustomTaskPane docPane = null;
   Tools.CustomTaskPane paneToRemove = null;
   foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
   {
      try
      {
         if (pane.Window == Wn)
         {
            docPane = pane;
            break;
         }
      }
      catch (ArgumentNullException)
      {
         // pane.Window is null, so document1 has been unloaded.
```

```
            paneToRemove = pane;
        }
    }
```

如果抛出一个异常，就从集合中删除冲突的任务面板：

```
// Remove pane if necessary
timerDisplayPanes.Remove(paneToRemove);
```

如果窗口中没有任务面板，这个方法最终就添加一个任务面板：

```
// Add task pane to doc
if (docPane == null)
{
    Tools.CustomTaskPane pane = this.CustomTaskPanes.Add(
        new TimerDisplayPane(documentEditTimes),
        "Document Edit Timer",
        Wn);
    timerDisplayPanes.Add(pane);
    pane.VisibleChanged +=
        new EventHandler(timerDisplayPane_VisibleChanged);
  }
}
```

添加的任务面板是 TimerDisplayPane 类的一个实例。稍后介绍这个类。添加它时使用的名称是 Document Edit Timer。另外，在调用 CustomTaskPanes.Add()方法后，还为得到的 CustomTaskPane 的 VisibleChanged 事件添加了一个事件处理程序。这样在第一次显示任务面板时，可以刷新显示内容：

```
private void timerDisplayPane_VisibleChanged(object sender, EventArgs e)
{
    // Get task pane and toggle visibility
    Tools.CustomTaskPane taskPane = (Tools.CustomTaskPane)sender;
    if (taskPane.Visible)
    {
        TimerDisplayPane timerControl = (TimerDisplayPane)taskPane.Control;
        timerControl.RefreshDisplay();
    }
}
```

TimerDisplayPane 类提供一个 RefreshDisplay()方法，它在上面的代码中调用。顾名思义，这个方法刷新 timerControl 对象的显示内容。

接着，代码确保监控所有文档。首先，创建新文档时，调用 eventInterface_NewDocument()事件处理程序，调用 MonitorDocument()和前面介绍过的 AddTaskPaneToWindow()方法监控该文档。

```
private void eventInterface_NewDocument(Word.Document Doc)
{
    // Monitor new doc
    MonitorDocument(Doc);
    AddTaskPaneToWindow(Doc.ActiveWindow);
```

新文档在计时器运行时启动，此时这个方法还清除了功能区菜单中的 Pause 复选框。这通过一个实用程序方法 SetPauseStatus()实现，该方法在功能区中定义：

```
    // Set checkbox
    Globals.Ribbons.TimerRibbon.SetPauseStatus(false);
}
```

在关闭文档之前，调用 eventInterface_DocumentBeforeClose()事件处理程序。这个方法冻结文档的计时器，更新总编辑时间，清除 Document 引用，并删除文档窗口中的任务面板(使用稍后介绍的 RemoveTaskPaneFromWindow()方法)，之后关闭窗口。

```
private void eventInterface_DocumentBeforeClose(Word.Document Doc,
    ref bool Cancel)
{
    // Freeze timer
    documentEditTimes[Doc.Name].EditTime += DateTime.Now
        - documentEditTimes[Doc.Name].LastActive;
    documentEditTimes[Doc.Name].IsActive = false;
    documentEditTimes[Doc.Name].Document = null;
    // Remove task pane
    RemoveTaskPaneFromWindow(Doc.ActiveWindow);
}
```

打开文档时，调用 eventInterface_DocumentOpen()方法。此处该方法要完成更多工作，因为在监控文档之前，这个方法必须查看计时器的名称，确定文档是否已有计时器：

```
private void eventInterface_DocumentOpen(Word.Document Doc)
{
    if (documentEditTimes.ContainsKey(Doc.Name))
    {
        // Monitor old doc
        documentEditTimes[Doc.Name].LastActive = DateTime.Now;
        documentEditTimes[Doc.Name].IsActive = true;
        documentEditTimes[Doc.Name].Document = Doc;
        AddTaskPaneToWindow(Doc.ActiveWindow);
    }
```

如果目前还没有监控文档，就为新文档配置一个新监控器：

```
    else
    {
        // Monitor new doc
        MonitorDocument(Doc);
        AddTaskPaneToWindow(Doc.ActiveWindow);
    }
}
```

RemoveTaskPaneFromWindow()方法用于从窗口中删除任务面板。其代码首先检查特定的窗口中是否有任务面板：

```
private void RemoveTaskPaneFromWindow(Word.Window Wn)
{
    // Check for task pane in window
    Tools.CustomTaskPane docPane = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        if (pane.Window == Wn)
```

```
            {
                docPane = pane;
                break;
            }
        }
```

如果找到一个任务面板，就调用 CustomTaskPanes.Remove()方法删除它。还要从任务面板引用的本地集合中删除它。

```
    // Remove document task pane
    if (docPane != null)
    {
        this.CustomTaskPanes.Remove(docPane);
        timerDisplayPanes.Remove(docPane);
    }
}
```

这个类中的最后一个事件处理程序是 eventInterface_WindowActivate()，在激活窗口时调用它。这个方法获得活动文档的计时器(先检查文档是否有计时器，因为在调用这个方法时，新文档可能还没有添加计时器)，选中功能区菜单上的复选框，以便更新文档的复选框：

```
private void eventInterface_WindowActivate(Word.Document Doc,
    Word.Window Wn)
{
    if (documentEditTimes.ContainsKey(this.Application.ActiveDocument.Name))
    {
        // Ensure pause checkbox in ribbon is accurate, start by getting timer
        DocumentTimer documentTimer =
            documentEditTimes[this.Application.ActiveDocument.Name];
        // Set checkbox
        Globals.Ribbons.TimerRibbon.SetPauseStatus(!documentTimer.IsActive);
    }
}
```

ThisAddIn 类的代码还包含两个实用程序方法。第一个方法——ToggleTaskPaneDisplay()用于设置 CustomTaskPanes.Visible 属性，为当前的活动文档显示或隐藏任务面板。

```
internal void ToggleTaskPaneDisplay()
{
    // Ensure window has task window
    AddTaskPaneToWindow(this.Application.ActiveDocument.ActiveWindow);
    // toggle document task pane
    Tools.CustomTaskPane docPane = null;
    foreach (Tools.CustomTaskPane pane in timerDisplayPanes)
    {
        if (pane.Window == this.Application.ActiveDocument.ActiveWindow)
        {
            docPane = pane;
            break;
        }
    }
    docPane.Visible = !docPane.Visible;
}
```

上述代码中的 ToggleTaskPaneDisplay()方法由功能区控件上的事件处理程序调用，如后面所述。最后，该类有另一个从功能区菜单中调用的方法，它允许功能区控件暂停或恢复文档的计时器：

```
internal void PauseOrResumeTimer(bool pause)
{
   // Get timer
   DocumentTimer documentTimer =
      documentEditTimes[this.Application.ActiveDocument.Name];
   if (pause && documentTimer.IsActive)
   {
      // Freeze timer
      documentTimer.EditTime += DateTime.Now - documentTimer.LastActive;
      documentTimer.IsActive = false;
   }
   else if (!pause && !documentTimer.IsActive)
   {
      // Resume timer
      documentTimer.IsActive = true;
      documentTimer.LastActive = DateTime.Now;
   }
  }
}
```

这个类定义中的其他代码是 Shutdown 事件的空事件处理程序，以及 VSTO 为关联 Startup 和 Shutdown 事件处理程序而生成的代码。

接着布置项目中的功能区，即 TimerRibbon，如图 49-13 所示。

图 49-13

这个功能区包含一个 RibbonButton、一个 RibbonSeparator、一个 RibbonCheckBox 和一个 DialogBoxLauncher。按钮使用大显示样式，其 OfficeImageId 设置为 StartAfterPrevious，显示如图 49-13 所示的钟面(这些图像在设计期间不可见)。功能区使用 TabHome 选项卡类型，从而其内容追加到 Home 选项卡中。

功能区有 3 个事件处理程序，每个处理程序都调用前面介绍的 ThisAddIn 类的一个实用程序方法：


```
private void group1_DialogLauncherClick(object sender,
   RibbonControlEventArgs e)
{
   // Show or hide task pane
   Globals.ThisAddIn.ToggleTaskPaneDisplay();
}
private void pauseCheckBox_Click(object sender, RibbonControlEventArgs e)
{
   // Pause timer
   Globals.ThisAddIn.PauseOrResumeTimer(pauseCheckBox.Checked);
}
private void toggleDisplayButton_Click(object sender,
   RibbonControlEventArgs e)
{
   // Show or hide task pane
   Globals.ThisAddIn.ToggleTaskPaneDisplay();
}
```

代码段 TimerRibbon.cs

ribbon 还包含自己的实用程序方法 SetPauseStatus()，如前所述，该方法由 ThisAddIn 类中的代码调用，以选中复选框或清除复选框。

```
internal void SetPauseStatus(bool isPaused)
  {
    // Ensure checkbox is accurate
    pauseCheckBox.Checked = isPaused;
}
```

这个解决方案中的另一个组件是在任务面板中使用的 TimerDisplayPane 用户控件。这个控件的布局如图 49-14 所示。

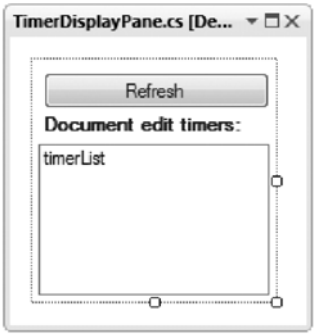

图　49-14

这个控件包含一个按钮、一个标签和一个列表框——这些都是很普通的显示控件，也可以用更漂亮的 WPF 控件替代它们也很简单。

该控件的代码保存对文档计时器的一个本地引用，该引用在构造函数中设置：


```
public partial class TimerDisplayPane : UserControl
{
   private Dictionary<string, DocumentTimer> documentEditTimes;
   public TimerDisplayPane()
   {
      InitializeComponent();
   }
   public TimerDisplayPane(Dictionary<string, DocumentTimer>
      documentEditTimes): this()
   {
      // Store reference to edit times
      this.documentEditTimes = documentEditTimes;
   }
```

代码段 TimerDisplayPane.cs

按钮事件处理程序调用 RefreshDisplay()方法刷新计时器的显示内容：

```
private void refreshButton_Click(object sender, EventArgs e)
{
   RefreshDisplay();
}
```

RefreshDisplay()方法也从 ThisAddIn 类中调用，如前所述。考虑到该方法的任务，这是一个相当复杂的方法。它还参照已加载文档列表，检查被监控的文档列表，并修正任何问题。这种代码在 VSTO 应用程序中常常是必不可少的，因为 COM Office 对象模型的接口偶尔不能像期望的那样工作。这里的经验规则是防御式编码。

该方法首先清除 timerList 列表框中的当前计时器列表：

```
internal void RefreshDisplay()
{
   // Clear existing list
   this.timerList.Items.Clear();
```

接着，检查监控器。这个方法迭代 Globals.ThisAddIn.Application.Documents 集合中的每个文档，确定文档是被监控、未被监控、或被监控了但从上次刷新以后改变了文件名。

要找出被监控的文档，只需根据键的 documentEditTimes 集合中的文档名，检查文档名：

```
// Ensure all docs are monitored
foreach (Word.Document doc in Globals.ThisAddIn.Application.Documents)
{
   bool isMonitored = false;
   bool requiresNameChange = false;
   DocumentTimer oldNameTimer = null;
   string oldName = null;
foreach (string documentName in documentEditTimes.Keys)
{
   if (doc.Name == documentName)
   {
      isMonitored = true;
      break;
   }
```

如果文档名不匹配，就比较文档引用，这样可以检测对文档名的更改，如下面的代码所示：

```
  else
  {
     if (documentEditTimes[documentName].Document == doc)
     {
        // Monitored, but name changed!
        oldName = documentName;
        oldNameTimer = documentEditTimes[documentName];
        isMonitored = true;
        requiresNameChange = true;
        break;
     }
  }
}
```

对于未监控的文档，需要创建一个新的监控器：

```
// Add monitor if not monitored
if (!isMonitored)
{
   Globals.ThisAddIn.MonitorDocument(doc);
}
```

而名称改变的文档需要通过用于旧命名文档的监控器重新关联起来：

```
  // Rename if necessary
  if (requiresNameChange)
  {
     documentEditTimes.Remove(oldName);
     documentEditTimes.Add(doc.Name, oldNameTimer);
  }
}
```

调整文档编辑计时器后，生成一个列表。代码还会检测引用的文档是否依然加载，对于没有依然加载的文档，把 IsActive 属性设置为 false，暂停该文档的计时器。这也是防御性编程方式：

```
// Create new list
foreach (string documentName in documentEditTimes.Keys)
{
   // Check to see if doc is still loaded
   bool isLoaded = false;
   foreach (Word.Document doc in
      Globals.ThisAddIn.Application.Documents)
   {
      if (doc.Name == documentName)
      {
         isLoaded = true;
         break;
      }
      }
      if (!isLoaded)
      {
         documentEditTimes[documentName].IsActive = false;
         documentEditTimes[documentName].Document = null;
      }
```

对于每个监控器，把一个列表项添加到列表框中，其列表框包含文档名和总编辑时间：

```
      // Add item
      this.timerList.Items.Add(string.Format("{0}: {1}", documentName,
         documentEditTimes[documentName].EditTime +
         (documentEditTimes[documentName].IsActive ?
         (DateTime.Now — documentEditTimes[documentName].LastActive):
         new TimeSpan(0))));
    }
  }
}
```

这就完成了这个例子中的代码。这个例子说明了如何使用功能区和任务面板控件，如何维护多个 Word 文档中的任务面板。它还揭示了本章前面介绍的许多技术。

49.5 小结

本章学习了如何使用 VSTO 为 Office 产品创建托管解决方案。

本章的第一部分介绍了 VSTO 项目的一般结构和可以创建的项目类型，还探讨了可用于简化 VSTO 编程的功能。

接下来详细论述了 VSTO 解决方案中可用的一些功能，讨论了如何实现与 Office 对象模型通信，介绍了 VSTO 中可用的名称空间和类型，学习了如何使用这些类型实现各种功能。之后，探讨了 VSTO 项目的一些编码功能，以及如何使用这些功能获得希望的结果。

然后，进行了一些实践。我们学习了如何在 Office 应用程序中管理插件，如何与 Office 对象模型交互操作，如何用功能区菜单、任务面板和动作面板定制应用程序的 UI。

最后，探讨了一个示例应用程序，它揭示了前面学习的 UI 和交互操作技术。这个示例不仅包含许多代码，还包含有用的技巧，包括如何在多个 Word 文档窗口中管理任务面板。