# 1 VS 2008扩展方式一览

　　扩展VS有三种主要的方式：Macro、Add-In和VsPackage。事实上，还有更多的选择，这里先简单列一下：

　　Macro

　　Add-In

　　VsPackage

　　VS Shell

　　Domain Specific Language Tool

　　Visualizer

　　Code Snippet

　　Project/Item Template

　　MSBuild

　 甚至External Tools（菜单Tools->External Tools）也可看作是一种扩展方式，我们可以将外部工具添加到VS菜单中，这样使用起来会更方便一点。

# 2 Add-In介绍

外接程序是在 Visual Studio 集成开发环境 (IDE) 中运行的已编译的 DLL。编译可以保护知识产权和提高性能。虽然可以手动创建外接程序，但您可能会发现使用“外接程序向导”更为简便。此向导创建一个功能全面但却很基本的外接程序，创建完该程序后可立即运行它。“外接程序向导”生成基本的空程序后，可向其添加代码并对其进行自定义。

## 2.1 使用“外接程序向导”创建外接程序

1. 创建一个新的 Visual Studio 外接程序项目。在“文件”菜单上指向“新建”，然后单击“项目”。定位到“其他项目类型”节点中的“扩展性”。当您创建项目时，“外接程序向导”将启动。
2. 在“选择编程语言”页上，选择要用于编写外接程序的编程语言。
3. 在“选择应用程序主机”页上选择一个或多个应用程序，比如 Visual Studio。

这样，您将能够选择需要的应用程序以便创建外接程序后可在其中运行该外接程序；例如，Visual Studio 或 Visual Studio“宏 IDE”。

1. 在“输入名称和说明”页中键入外接程序的名称和说明。创建了外接程序后，其名称和说明将显示在“外接程序管理器”对话框的“可用外接程序”列表中。向外接程序的说明中添加足够的详细信息，以便用户能够了解外接程序的功能、工作方式等信息。
2. 在“选择外接程序选项”页可以指定：
   1. 是否希望外接程序显示在“工具”菜单上。
   2. 希望启动外接程序的时间。
   3. 外接程序是否使用模式用户界面 (UI)。如果不使用，则可以将外接程序安全地用于命令行生成。
3. 在“选择‘帮助’中的‘关于’信息”页中，指定是否要将有关外接程序的信息显示在 Visual Studio 帮助中的“关于”窗口中。如果要显示，请添加想要显示的信息。

可以添加到 Visual Studio 帮助中的“关于”窗口的信息包括版本号、支持详细信息和授权数据等类似信息。

7完成步骤 1 - 6 后，选定的选项会显示在“摘要”页中以供查看。如果感到满意，请单击“完成”以创建外接程序。如果要更改某些内容，请单击“上一步”按钮。

现在即生成了一个功能全面的基本外接程序。若要使外接程序能够执行所需的操作，则必须添加相应代码。有关更多信息，请参见[扩展 Visual Studio 环境](http://msdn.microsoft.com/zh-cn/library/esk3eey8(v=vs.90).aspx)。

若要了解“外接程序向导”的后台操作，可以检查它所创建的代码。

## 2.2引用自动化程序集和 DTE2 对象

若要创建自动化应用程序，则必须执行一些步骤获取对自动化成员的访问权。首先必须引用必需的自动化程序集，然后必须获取对顶级对象 [DTE2](http://msdn.microsoft.com/zh-cn/library/envdte80.dte2(v=vs.90).aspx) 的引用。

在 Visual Studio .NET 2002 和 Visual Studio .NET 2003 中，所有的核心自动化项均位于名为 [EnvDTE](http://msdn.microsoft.com/zh-cn/library/envdte(v=vs.90).aspx) 的程序集中，并且其最高分层对象为 [DTE](http://msdn.microsoft.com/zh-cn/library/envdte.dte(v=vs.90).aspx) 对象。该对象是所有核心自动化对象、集合及其成员的父对象。[DTE2](http://msdn.microsoft.com/zh-cn/library/envdte80.dte2(v=vs.90).aspx) 是从 [DTE](http://msdn.microsoft.com/zh-cn/library/envdte.dte(v=vs.90).aspx) 派生的。

在 Visual Studio 2005 和 Visual Studio 2008 中，只对这些核心自动化对象、集合及成员的一部分进行添加和更新。所有新的、更新的功能都位于一个名为 [EnvDTE80](http://msdn.microsoft.com/zh-cn/library/0e105c68(v=vs.90).aspx) (EnvDTE 版本 8.0) 和[EnvDTE90](http://msdn.microsoft.com/zh-cn/library/bb513943(v=vs.90).aspx) (EnvDTE 版本 9.0) 的程序集中，而不是更新现有程序集并损坏现有外接程序和自动化项目的向后兼容性。[EnvDTE80](http://msdn.microsoft.com/zh-cn/library/0e105c68(v=vs.90).aspx) 和 [EnvDTE90](http://msdn.microsoft.com/zh-cn/library/bb513943(v=vs.90).aspx) 中的大多数更新的函数都保持与以前版本相同的名称，但在函数名称末尾附加一个数字。例如，在较新版本中，[TaskItems](http://msdn.microsoft.com/zh-cn/library/envdte.taskitems(v=vs.90).aspx) 集合名为 [TaskItems2](http://msdn.microsoft.com/zh-cn/library/envdte80.taskitems2(v=vs.90).aspx)，[Solution](http://msdn.microsoft.com/zh-cn/library/envdte.solution(v=vs.90).aspx) 对象名为 [Solution2](http://msdn.microsoft.com/zh-cn/library/envdte80.solution2(v=vs.90).aspx)。由于新成员比早期版本的功能更强健，并且包含最新功能，因此推荐在编写新的自动化应用程序时使用新对象和新集合。

虽然新项位于 [EnvDTE80](http://msdn.microsoft.com/zh-cn/library/0e105c68(v=vs.90).aspx) 和 [EnvDTE90](http://msdn.microsoft.com/zh-cn/library/bb513943(v=vs.90).aspx) 中，但是大多数核心自动化功能仍然位于 [EnvDTE](http://msdn.microsoft.com/zh-cn/library/envdte(v=vs.90).aspx) 中。因此，在编写新的自动化应用程序（如外接程序）时，一定要引用 [EnvDTE](http://msdn.microsoft.com/zh-cn/library/envdte(v=vs.90).aspx)、[EnvDTE80](http://msdn.microsoft.com/zh-cn/library/0e105c68(v=vs.90).aspx) 和[EnvDTE90](http://msdn.microsoft.com/zh-cn/library/bb513943(v=vs.90).aspx)。只需一步就可完成此操作，方法是引用 COM 库“Microsoft 开发环境 8.0”和“Microsoft 开发环境 9.0”。另外，如果使用 [EnvDTE](http://msdn.microsoft.com/zh-cn/library/envdte(v=vs.90).aspx) 程序集中的成员，也必须设置对 [DTE](http://msdn.microsoft.com/zh-cn/library/envdte.dte(v=vs.90).aspx) 对象和 [DTE2](http://msdn.microsoft.com/zh-cn/library/envdte80.dte2(v=vs.90).aspx) 对象的引用。这使您可以访问所有的项。

有关获取对项目特定的对象模型的访问权的信息，请参见 [自动化功能组](http://msdn.microsoft.com/zh-cn/library/6aa952xb(v=vs.90).aspx)。

## 2.3在工具栏和菜单上显示外接程序

当使用“外接程序向导”创建外接程序，然后选择将其显示为命令的选项时，默认情况下它将被添加到“工具”菜单。如果修改代码，您也可以将外接程序显示在其他菜单和工具栏上（统称为 CommandBars）。与 CommandBars 相关的类型和成员包含在名为 Microsoft.VisualStudio.CommandBars 的程序集中。

Visual Studio 提供了三种 CommandBar 对象：

* **工具栏** — 包含一个或多个菜单栏。
* **菜单栏** — 工具栏上的命令，如“文件”、“编辑”和“视图”。
* **快捷菜单**（也称为上下文菜单或弹出式菜单） — 当右击某菜单或对象（例如一个文件或项目）时在屏幕上显示的菜单。子菜单层叠出菜单命令或快捷菜单。在 Visual Studio 中，快捷菜单与其他菜单类似。但是，您可以通过指向菜单栏内的某个箭头，或通过右击集成开发环境 (IDE) 中的项访问它们。

# 3 Add-In项目

　　Add-In项目是一个类库项目，仅此而已。该项目包含了“Connect.cs”文件，它定义了Connect类，还有一个配置文件FirstAddin.AddIn。

　　打开Connect.cs，我们仔细分析一下。Connect类实现了两个接口，一是IDTExtensibility2，该接口用于在Add-In和IDE之间进行通信；二是IDTCommandTarget，如果选择了向导中的UI选项，就需要实现它。

　　IDTExtensibility2包含5个方法：

　　OnConnection：在加载Add-In时调用

　　OnStartupComplete：在Add-In随着VS的启动完成加载后调用

　　OnAddInsUpdate：在VS加载或卸载Add-In时调用

　　OnBeginShutdown：在VS关闭时调用

　　OnDisconnection：在卸载Add-In时调用

　 　在文件顶部可以看到引用了若干个命名空间，对于Add-In开发来说最重要的是其中三个：Extensibility、EnvDTE和 EnvDTE80。Extensibility定义了IDTExtensibility2使用的类型；后面两个命名空间则定义了自动化对象模型 （Automation Object Model，以下简称AOM）中的类型。

　　回到前面的5个方法，最重要的一个是 OnConnection，VS在加载Add-In时调用它，通过第一个参数application将AOM的根对象传入，向导产生的代码将该对象的引用 保存\_applicationObject中；同时通过第三个参数addInInst将当前Add-In所对应的AddIn对象传入，保存在 \_addInInstance中。再往下看，这些代码将向Tools菜单添加一个菜单命令（如果你在向导中选中该选项的话），其中包括如下代码：

　　C# Code

if(connectMode == ext\_ConnectMode.ext\_cm\_UISetup)  
{  
}

　　connectMode参数的值表示Add-In是如何加载的。如果Add-In通过菜单命令加载，那么该参数的值为ext\_ConnectMode.ext\_cm\_UISetup。

　 　对于另外4个方法，向导没有产生任何代码。而对于IDTCommandTarget接口的两个方法QueryStatus和Exec，则添加必要的代码 来管理菜单命令以及命令点击事件的处理。Connect类中就这些内容了，那我们在向导中选择的宿主环境、名称描述等信息放在哪里呢？

　　.Addin文件

　　在我们的例子中可以看到，有个文件FirstAddin.AddIn，Add-In通过这个文件向VS进行注册。来看看它的结构如何。

　　它本质上是XML文件（就像模板和Code Snippet的配置文件一样）：

　　XML Code - Add-In配置信息

<?xml version="1.0" encoding="UTF-16" standalone="no"?>  
<Extensibility xmlns="http://schemas.microsoft.com/AutomationExtensibility">  
<HostApplication>  
<Name>Microsoft Visual Studio</Name>  
<Version>9.0</Version>  
</HostApplication>  
<Addin>  
<FriendlyName>MyFirstAddin</FriendlyName>  
<Description>MyFirstAddin, it's so exciting!</Description>  
<Assembly>FirstAddin.dll</Assembly>  
<FullClassName>FirstAddin.Connect</FullClassName>  
<LoadBehavior>0</LoadBehavior>  
<CommandPreload>1</CommandPreload>  
<CommandLineSafe>0</CommandLineSafe>  
</Addin>  
</Extensibility>

　　这些信息主要分为3类：

　　1）宿主环境

　　通过<HostApplication>节点来 配置该Add-In适用于哪些宿主环境，该节点数目、顺序不限。在<Name>节点中说明宿主环境的名称，除了Microsoft Visual Studio还可以是Microsoft Visual Studio Macros，也就是Macros IDE；在<Version>节点中说明支持的版本，还可以是7.1、8.0，也可以用\*表示支持所有版本。

　　2）Add-In信息

　　<Addin>节点指定了Add-In本身的信息。它可以包含如下子节点：

　　<FriendlyName>：可选的，为Add-In指定一个有意义的名称；

　　<Description>：可选的，为Add-In指定有意义的描述信息；

　　<AboutBoxDetails>和<AboutIconData>：都是可选的，如果要在About对话框中显示Add-In的话，该节点用于指定其详细信息和图标；

　　<Assembly>：必填的，Add-In所在的程序集；

　　<FullClassName>：必填的，指定程序集内实现了IDTExtensibility2接口的类，要使用完全限定名称；

　　<LoadBehavior>：可选的，指定VS加载Add-In的方式，0表示VS不会自动加载，必须手工加载；1表示Add-in在VS启动的时候加载；4表示通过命令行方式加载；

　　<CommandPreload>：可选的，指定Add-In应当预先加载；

　　<CommandLineSafe>：可选的，指定Add-In是否是命令行安全的以及是否显示用户界面。

　　3）选项页（Tools Options Page）信息

　　我们可以很容易地在VS的Tools -> Options对话框中添加自己的选项页，从而对Add-In进行配置，不过这里先行略过，在后续的随笔中将会介绍。

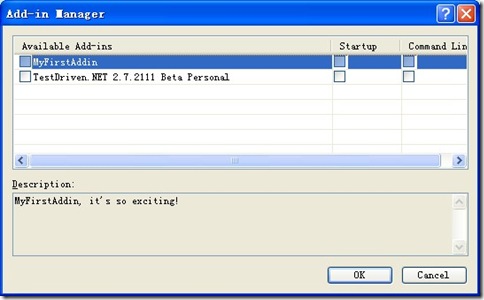
　　CommandBar.resx

　　除了Connect.cs和.AddIn文件，还有一个文件是CommandBar.resx，这里面存放了一个命令条（CommandBar）的文本值的列表。它针对的是不同的自然语言，实际上在Connect.cs中，在获取Tools菜单时就用到了它。

　　想一想，现在有了一个编译好的程序集还有.Addin配置文件，那VS就有足够的信息来启动、管理Add-In了。问题是，把这两个文件放在哪 里呢？在项目当中有一个FirstAddin - For Testing.AddIn文件，这个文件存放的位置是[My Documents Path]Visual Studio 2008Addins，在我们按下F5测试Add-In的时候VS就是使用这个文件来加载的，查看它里面的配置可以看到它指向的程序集正是当前项目编译后 的程序集。所以我们的Add-In编译完毕后，FirstAddin - For Testing.Addin删掉，把程序集和FirstAddin.Addin文件拷贝到[My Documents Path]Visual Studio 2008Addins下，就算是一种最简单的部署了。

　　加载和管理Add-In

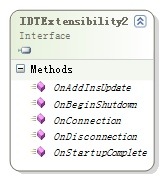
　　在生成Add-In后， 需要把它加载进VS。如果你在向导中选择在VS启动时加载，那么VS会在每次启动时自动加载Add-In。如果选择通过菜单命令加载，你也可以打开VS 后，通过Add-In Manager（菜单Tools -> Add-In Manager）修改相关的设定。



　　　 　本节主要关注的是Add-In向导所产生的代码，其中的重点是Connect.cs和.Addin文件。Connect类是Add-In的实现类，有了 它一个程序集才得以成为一个Add-In；.Addin文件中包含了Add-In的配置信息，VS以此来管理Add-In。有了这些，我们对Add-In 的运行机制就有了更清楚的认识，在下一节中，我将介绍Add-In中的生命周期和事件。

# 4 Add-In的事件

　　Add-In是事件驱动的，可以猜到的事件有加载、卸载、状态改变等等。事实上，这些事件都与IDTExtensibility2接口有关，也就是该接口的5个方法：



　 　如果要了解这些方法如何执行，一个办法是在这些方法中加一个MessageBox，然后通过Add-In Manager进行一些操作，来观察事件的执行。现在使用Add-In向导建立一个简单的Add-In，名字为LifeCycleAddin，不要选择在 Tools菜单显示命令，也不要选择在VS启动时加载。然后把Connect类的代码简化一下：

　　C# Code - Add-In事件演示

///　<summary>The　object　for　implementing　an　Add-in.</summary>  
public　class　Connect　:　IDTExtensibility2  
{  
public　Connect()  
{  
}  
  
///　<summary>  
///　Receives　notification　that　the　Add-in　is　being　loaded.  
///　</summary>  
public　void　OnConnection(object　application,　ext\_ConnectMode　connectMode,   
object　addInInst,　ref　Array　custom)  
{  
\_applicationObject　=　(DTE2)application;  
\_addInInstance　=　(AddIn)addInInst;  
  
MessageBox.Show(string.Format("Event:　OnConnection,　connectMode:　{0}",　connectMode));  
}  
  
///　<summary>  
///　Receives　notification　that　the　Add-in　is　being　unloaded.  
///　</summary>  
public　void　OnDisconnection(ext\_DisconnectMode　disconnectMode,　ref　Array　custom)  
{  
MessageBox.Show(string.Format("Event:　OnDisconnection,　connectMode:　{0}",　disconnectMode));  
}  
  
///　<summary>  
///　Receives　notification　when　the　collection　of　Add-ins　has　changed.  
///　</summary>  
public　void　OnAddInsUpdate(ref　Array　custom)  
{  
MessageBox.Show("OnAddInsUpdate");  
}  
  
///　<summary>  
///　Receives　notification　that　the　host　application　has　completed　loading.  
///　</summary>  
public　void　OnStartupComplete(ref　Array　custom)  
{  
MessageBox.Show("OnStartupComplete");  
}  
  
///　<summary>  
///　Receives　notification　that　the　host　application　is　being　unloaded.  
///　</summary>  
public　void　OnBeginShutdown(ref　Array　custom)  
{  
MessageBox.Show("OnBeginShutdown");  
}  
private　DTE2　\_applicationObject;  
private　AddIn　\_addInInstance;  
}

　　每个方法的注释说明了相应的事件何时触发。OnConnection是在Add-In加载的时候；OnDisconnection是在 Add-In卸载的时候；OnAddInsUpdate是在所有Add-In的集合状态发生改变的时候；OnStartupComplete是在宿主环境 加载完成的时候；OnBeginShutdown则是在宿主环境将被关闭的时候。现在编译项目，然后关闭VS。

　　打开VS，开始第一回合 的观察。由于没有选择在VS启动时加载，所以现在什么也不会发生。打开Add-In Manager，对于LifeCycleAddin，将其设置为可用，确定。这时触发了OnConnection，connectMode为 ext\_cm\_AfterStartup，也就是说在VS启动之后才加载的；然后还触发了OnAddinsUpdate，因为 LifeCycleAddin的状态改变了。再次打开Add-In Manager，对于LifeCycleAddin，将其设置为启动时加载，确定，再次触发OnAddinsUpdate。现在关闭VS，由于Add- In已经加载，所以会触发OnBeginShutdown，然后是OnDisconnection，说明Add-In已经被卸载。

　　打开 VS，开始第二回合的观察。由于选择了在VS启动时加载，所以此时触发了OnConnection，connectMode为 ext\_cm\_Startup，也就是说是在VS启动时加载的；之后连续触发了OnAddinsUpdate和OnStartupComplete，只有 设置为在VS启动时加载才可能触发该事件。至此，5个事件都已经触发过了。

　　比较有意思的是OnAddinsUpdate。现在打开Add-In Manager，改变另一个Add-In的设置，点击确定，该事件也会触发，这就是前面所说的“所有Add-In的集合状态发生改变的时候”。

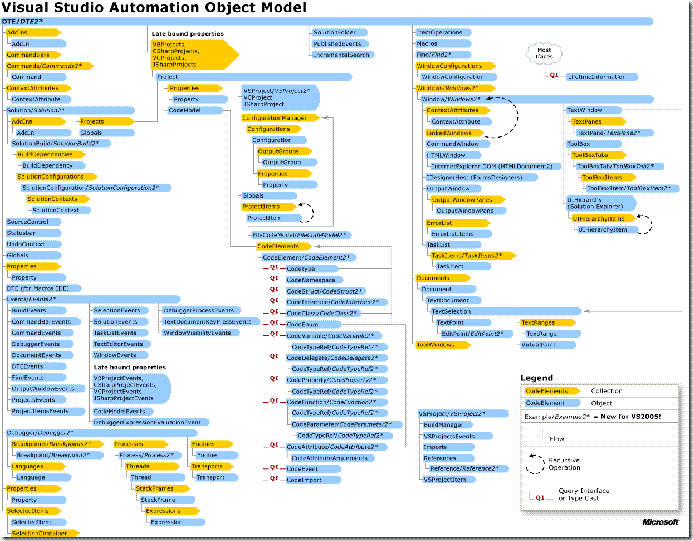
　 　由前面的介绍可以了解到，实现IDTExtensibility2接口是每个Add-In的核心所在。但是仅仅这些显然还不够。我们不仅需要知道VS合 适启动、卸载或改变了Add-In，我们还要能够在这些时候访问VS，否则开发VS的Add-In也就没意义了。这就用到了VS的自动化对象模型 （Automation Object Model）。

　　VS自动化对象模型简介

　　在Connect.cs文件的顶部using部分，可以看到两个命名空间：EnvDTE和 EnvDTE80。EnvDTE表示开发环境工具扩展（Environment Development Tools Extensibility，常简称为DTE），就是在这里定义了VS的自动化对象模型（以下简称AOM）。而EnvDTE80的80表示8.0版本的 AOM，其实还有一个表示9.0版本的EnvDTE90，但没有引用进来。

　　简单说一下它们的关系。EnvDTE表示VS2005之前的 DTE版本，在每个版本中微软都会修复一些bug或添加新的功能，到了VS2005，微软使用了EnvDTE80表示新版本的变化（包括修复和增强），同 时对于那些旧版本中已经存在的类，在后面加了一个数字2表示该类的新版本，如CodeFunction2表示是EnvDTE80中的新类型，而 CodeFunction则表示EnvDTE中对应的那个类。EnvDTE90与此类似，比如Solution3、Solution2和Solution 分别表示三个版本中表示解决方案的类。对于这些不同版本的类，微软的做法是用新版本的类继承旧的版本，然后进行扩展。

　　但是相对于 EnvDTE80与EnvDTE之间的变化，EnvDTE90的变化要小很多。大部分时候EnvDTE80就够用了，所以默认情况 下，Connect.cs文件没有引用EnvDTE90。以后当你看到带着2或3后缀的类型，就能明白它的来历了。下面是AOM的结构图（点击查看大 图）：



　　不出意外的是，结构很复杂。原因有二：首先VS本身很复杂，DTE用来表示VS中的元素，不能不复杂；其次，AOM和DTE源自COM，在.NET和COM间的互操作增强一些额外的工作。不过不用担心，这些类封装得非常之好，用起来还是比较容易的。

　　这个类型结构的顶端是DTE/DTE2，它是所有其它类型的容器。DTE主要包含5部分内容：解决方案和项目、命令 （Command）、事件、文档、调试器，通过这些，我们就能够操作VS的方方面面（可以先看一下图中类的名字）。在后续的章节中，你将看到这些内容的详 细用法。

　　再议IDTExtensibility2接口

　　现在回到IDTExtensibility2接口，仔细了解一下它的各个方法。

　　1）OnConnection

　　在实现这个接口的时候，我们需要获得DTE对象，这样才能操作VS，这件事要在OnConnection中去做。

　　C# Code - Method Signature

public　void　OnConnection(object　application,　ext\_ConnectMode　connectMode,　  
object　addInInst,　ref　Array　custom)

　　 application参数持有AOM根对象的引用，它同时实现了EnvDTE.DTE和EnvDTE80.DTE2接口，所以在我们的例子中，它被转换 为DTE2。connectMode参数告诉Add-In是以何种方式加载的，它的值来自Extensibility.ext\_ConnectMode枚 举：

　　ext\_cm\_AfterStartup：在VS启动之后加载

　　ext\_cm\_Startup：在VS启动之时加载

　　ext\_cm\_External：在VS外部加载（VS已经不再使用该值）

　　ext\_cm\_CommandLine：从命令行加载

　　ext\_cm\_Solution：在解决方案内加载

　　ext\_cm\_UISetup：在建立用户界面时加载

　　我们可以根据该参数值的不同进行相应的操作，比如如果是ext\_cm\_UISetup，可以在菜单上添加一条命令（就像Add-In向导所做的那样）。

　 　Add-In本身是AddIn接口的一个实例，addInInst参数持有该实例的引用，我们可以将该值保存下来备用。最 后，IDTExtensibility2接口的每个方法都有一个custom参数，Add-In的宿主环境可以通过它来传递宿主相关的信息，不过VS总是 传递一个空的数组（汗。。。）。

　　2）OnStartupComplete

　　OnStartupComplete事件仅仅在Add-In随VS启动加载的时候才会触发。

　　C# Code - Method Signature

void　OnStartupComplete(ref　Array　custom)

　　如果一个Add-In随VS启动而加载，OnConnection并非总是进行初始化的好地方——比如，Add-In加载的较早，而Add-In需要访问的VS组件尚未加载完毕。

　　3）OnAddInsUpdate

　　C# Code - Method Signature

void　OnAddInsUpdate(ref　Array　custom)

　 　在某个Add-In被加载或卸载的时候，OnAddInsUpdate事件会触发。OnAddInsUpdate事件没有提供被加载或卸载Add-In 的信息，不过我们有办法获取到。大体原理是：通过DTE.AddIns/DTE2.AddIns集合我们能够获取到所有的Add-In，里面的元素类型为 AddIn，AddIn有个Connected属性，用以表示该Add-In是否处于加载状态，我们在首次触发OnAddInsUpdate事件的时候记 录所有Add-In的状态，在下次触发的时候就知道那些Add-In状态改变了，这里就不再给出代码了。

　　4）OnBeginShutDown

　　C# Code - Method Signature

void　OnBeginShutdown(ref　Array　custom)

　　如果在一个Add-In运行的时候关闭VS，OnBeginShutDown事件会触发。我们在这个时候可以做一些必要的清理工作。

　　5）OnDisconnection

　　C# Code - Method Signature

void　OnDisconnection(ext\_DisconnectMode　disconnectMode,　ref　Array　custom)

　 　在Add-In的生命周期结束的时候，OnDisconnection事件会触发。它跟OnBeginShutDown事件的不同之处在于，这里结束的 是Add-In而不是VS。disconnectMode参数的值来自Extensibility.ext\_DisconnectMode枚举：

　　ext\_dm\_HostShutdown：因为VS关闭而卸载

　　ext\_dm\_UserClosed：在VS运行时卸载

　　ext\_dm\_UISetupComplete：在用户界面创建完毕后卸载

　　ext\_dm\_SolutionClosed：在解决方案关闭时卸载

　　它的作用类似于ext\_ConnectMode，我们可以根据Add-In卸载方式的不同采取不同的动作。

　　唔，至此Add-In的事件和生命周期介绍完毕。

　　　 　本节主要介绍了VS Add-In的事件和生命周期，通过这些知识，我们能够知道在何时获取需要的信息；同时还简单介绍了VS自动化对象模型。加上Add-In运行机制解析 （上），我们应当对Add-In的运行机制有个基本的了解，为接下来的开发打下基础。到现在我们还不足以写出真正有用的Add-In，从下一节开始我将介 绍如何开发真正有用的Add-In。

## 5 添加新的命令

在之前的章节中，我介绍了Add-In的运行机制，这样对Add-In的事件、生命周期、与VS如何交互可以有个基本的了解了。现在是时候看看如何在VS中完成一些操作，这才是Add-In开发的目的所在。

　　一般的，Add-In应当提供一些界面元素，这样用户可以进行某些操作，比如在主菜单内添加一个菜单项，或者在编辑器的上下文菜单内添加一个菜单项，在本文中就来看看如何实现这些。

　　关于命令（Command）

　　考虑一个极为常见的场景：在编写代码的过程中，选中一段文本，点击Edit->Copy（或工具栏按钮）或者按下Ctrl+C，我们可以把选中文本拷贝到剪贴板，这个过程的背后发生了什么？

　 　是什么完成的拷贝操作呢？答案是命令。可以认为命令就是某个特定的功能，如Copy、Paste、Cut等等。VS本身就内置大量的命令（有数千个之 多），而上面说到的菜单项、工具栏或快捷键则执行了这些命令。通过Tools->Options菜单可以查看命令列表：



　　图1：VS的命令列表

　　值得注意的是，执行命令并非必须通过界面元素，比如在命令窗口中：

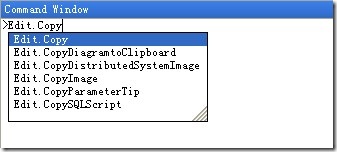


　　图2：在命令窗口执行命令

　 　这样也可以执行同样的命令。现在我们知道存在多种方式来执行命令，即菜单、工具栏、快捷键或命令窗口，这里统称为触发者。在VS中，触发者与命令是分离 开来的：用户通过触发者来执行命令，而命令负责检查自身的状态（名称、是否可见、是否可用等等）并执行。这意味着，命令可以对应一个或多个菜单项，也可以 不对应任何菜单项。

　　另一方面，对于同一个命令，比如Edit.Copy，仍然可能有不同的情况。在文本编辑器内和在解决方案管理器内的 Edit.Copy命令执行内容并不相同。这里有一个命令目标（Command Target）的概念，VS将命令转向给了命令目标，而命令目标按自己的实现来执行该命令。总结下来就是：

对象 职责 触发者 提供一种方式供用户使用 命令 一个逻辑实体，检查自身的状态，可以执行命令，也可以转向一个命令目标 命令目标 根据传递过来的命令，按自己的实现来执行它

　　关于命令栏（CommandBar）

　　命令是个“乖孩子”，首先是我们让它做什么它就做什么，第二就是它不会单独出门，它会跟其它同伴待在一块儿，它们都被放在命令栏中。比如主菜单中File下的那些菜单项。

　　既然这样，如果希望向已经存在的命令栏内添加命令，就得首先找到命令栏才能添加。现在来看一个例子，向Tools菜单中添加一个菜单项。

　　添加一个新命令

　 　使用Add-In向导新建一个Add-In，名字定为NEnhancer，意为对VS进行增强:)，以后关于Add-In的例子都会放在这里面。注意要 选中向Tools菜单添加一个菜单项。在Connect类的OnConnection方法中可以看到添加菜单项的代码如下：

　　C# Code - 向Tools菜单添加新项

　　　 try  
{  
// 如果需要向其它菜单栏添加命令，将Tools改为其它名称（要使用英文的）  
// 关于这些菜单栏的名称，可以查看CommandBar.resx文件  
string resourceName;  
ResourceManager resourceManager =　new ResourceManager("NEnhancer.CommandBar", Assembly.GetExecutingAssembly());  
CultureInfo cultureInfo =　new CultureInfo(\_applicationObject.LocaleID);  
  
if(cultureInfo.TwoLetterISOLanguageName ==　"zh")  
{  
System.Globalization.CultureInfo parentCultureInfo = cultureInfo.Parent;  
resourceName = String.Concat(parentCultureInfo.Name, "Tools");  
}  
else  
{  
resourceName = String.Concat(cultureInfo.TwoLetterISOLanguageName, "Tools");  
}  
toolsMenuName = resourceManager.GetString(resourceName);  
}  
catch  
{  
// 如果没能找到，就使用英文版本的名称  
toolsMenuName =　"Tools";  
}  
  
// 获取VS的主菜单  
CommandBar menuBarCommandBar =  
((CommandBars)\_applicationObject.CommandBars)["MenuBar"];  
  
// 获取Tools菜单  
CommandBarControl toolsControl = menuBarCommandBar.Controls[toolsMenuName];  
CommandBarPopup toolsPopup = (CommandBarPopup)toolsControl;  
  
// 如果需要添加多个菜单项，这个try/catch代码块可以多次使用  
// 但别忘了更新QueryStatus/Exec这两个方法  
try  
{  
// 向Commands集合添加命令  
Command command = commands.AddNamedCommand2(\_addInInstance, "NEnhancer", "NEnhancer",  
"Executes the command for NEnhancer", true, 59, ref contextGUIDS,  
(int)vsCommandStatus.vsCommandStatusSupported+(int)vsCommandStatus.vsCommandStatusEnabled,  
(int)vsCommandStyle.vsCommandStylePictAndText, vsCommandControlType.vsCommandControlTypeButton);  
  
// 向菜单中添加新项  
if((command !=　null) && (toolsPopup !=　null))  
{  
command.AddControl(toolsPopup.CommandBar, 1);  
}  
}  
catch(System.ArgumentException)  
{  
// 如果引发异常，可能是由于同名的命令已经存在了，可以忽略该异常  
}

　　代码的功能可以查看其中的注释。首先通过资源文件获取Tools菜单的名称（文化相关的），然后是获取主菜 单，这里用的就是CommandBar，这里通过名称从DET2.CommandBars集合中进行查找。往下可以看到Tools菜单是主菜单栏下的一个 CommandBarPopup控件，获取了Tools菜单后就可以给它添加命令了，其关键是AddNamedCommand2方法，它的参数信息为：

　　C# Code - Method Signature

Command AddNamedCommand2(AddIn AddInInstance, string Name, string ButtonText,  
string Tooltip, bool MSOButton, object Bitmap, ref　object[] ContextUIGUIDs,  
int vsCommandStatusValue, int CommandStyleFlags, vsCommandControlType ControlType);

　　AddInInstance：用作命令对象的AddIn对象

　　Name：命令名称

　　ButtonText：当命令显示在菜单或工具栏时的文本

　　ToolTip：命令的提示信息

　　MSOButton：如果是true，表示使用预定义的图标，否则使用自定义的图标

　　Bitmap：如果MSOButton值为true，那么该参数的值将用于所用预定义图标的索引；否则用作自定义图标的Id

　　ContextUIGUIDs：VS定义了一些GUID值来标识其状态的改变。比如如果希望进入debug模式命令可用，可将该参数的值设置为vsContextGuidDebugging

　　sCommandStatusValue：命令的默认可用状态

　　CommandStyleFlags：该参数用于控制命令的外观，比如只显示图标、只显示文本或者都显示

　　ControlType：所添加命令对应的UI元素的类型，比如菜单项、组合框等

　 　从本文开头举的例子可以看到，对于VS内置的菜单项，它们对应的命令名称是有一定规律的，即按照菜单的嵌套关系，如Edit.Copy，表示Edit下 的Copy项（有空格的话要去掉）。那么对于我们的Add-In来说，也是有规律的，即 Namespace.ClassName.CommandName，这里就是NEnhancer.Connect.NEnhancer。

　　命令的名称应当反应它的意图，所以这里把AddNamedCommand2的第二个参数改为CommandViewer。前面提到过，VS中有很多命令，它们按照命令栏来组织，这里就做一个Add-In来查看所有的命令和命令栏，方便以后的开发。

　　命令的执行

　 　现在命令和菜单项是添加了，当用户点击菜单时，如何处理它呢？要实现IDTCommandTarget接口，一旦实现了它，我们的Connect类就成 为一个合格的命令目标（Command Target）了。确切地说，当用户点击菜单时，VS要执行它的Exec方法，Exec方法有如下几个参数：

　　C# Code - Method Signature

void Exec(string CmdName, vsCommandExecOption ExecuteOption,   
ref　object VariantIn, ref　object VariantOut, ref　bool Handled);

　　CmdName：命令的全称，Add-In中命令的命名规则是Namespace.ClassName.CommandName

　　vsCommandExecOption：绝大多数情况下，该参数的值是vsCommandExecOptionDoDefault值，用于通知Add-In按部就班地行事

　　VariantIn：如果有数据要传给命令，就使用此参数

　　VariantOut：与VariantIn相反，此参数用于向调用者传递数据

　　Handler：如果设置为true，VS就知道命令已经执行完毕；如果为false，VS会寻找其它可以执行命令的方法，对Add-In来说，这意味着出错——不可能有其它的地方可以执行我们自定义的命令

　　命令的状态

　　有的命令并不总是可用，比如我们开发了一个Add-In，它专门针对于文本编辑器，如果没有任何文件打开，它就不应该是可用的。这里要使用IDTCommandTarget接口的另一个方法QueryStatus，它的参数有：

　　CmdName：命令的全称

　　NeededText：当前来说，该参数的值只会是vsCommandStatusTextWantedNone。但是在开发Add-In的时候，强烈建议要对此进行检查，因为VS还保留了其它的可能值作将来之用（可以参考向导生成的代码）

　　StatusOption：这是最重要的参数（ref），我们要给它赋值，告诉VS传入的这个命令是否支持、是否可用或者是否可见，这几种情况可以同时存在，此时对它们可以使用“或”操作。

　　CommandText：此参数当前VS还没有使用，不要给它赋值

　　好了，了解了命令的概念，也知道如何添加、执行命令了，剩下的就是实现命令的功能了。这里要新建一个窗体，添加一个TreeView来显示命令栏和命令：

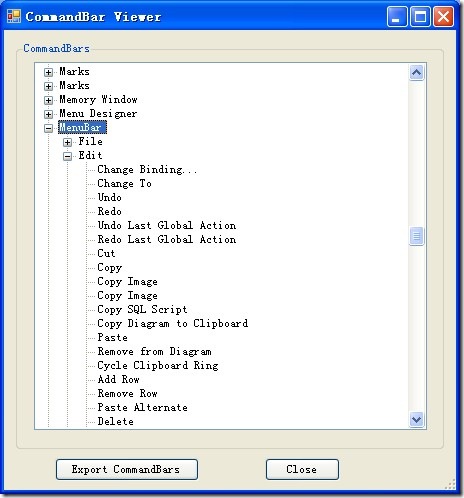


　　图3：CommandBarViewer窗体

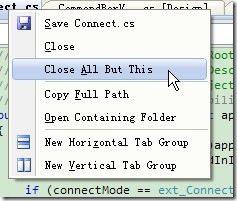
　　现在只要在Connect类中稍作修改显示窗体：

　　C# Code - 执行命令 public　void QueryStatus(string commandName, vsCommandStatusTextWanted neededText,   
ref vsCommandStatus status, ref　object commandText)  
{  
if (neededText == vsCommandStatusTextWanted.vsCommandStatusTextWantedNone)  
{  
if (commandName ==　"NEnhancer.Connect.CommandViewer")  
{  
status = (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported | vsCommandStatus.vsCommandStatusEnabled;  
return;  
}  
}  
}  
  
public　void Exec(string commandName, vsCommandExecOption executeOption,   
ref　object varIn, ref　object varOut, ref　bool handled)  
{  
handled =　false;  
if (executeOption == vsCommandExecOption.vsCommandExecOptionDoDefault)  
{  
if (commandName ==　"NEnhancer.Connect.CommandViewer")  
{  
ShowCmdBarViewer();  
  
handled =　true;  
return;  
}  
}  
}  
  
private　void ShowCmdBarViewer()  
{  
CommandBarViewer viewerForm =　new CommandBarViewer();  
viewerForm.DTEObject = \_applicationObject;  
viewerForm.ShowDialog();  
}

　　通过这个例子，我们可以了解如何添加新的命令并执行它，其中的关键方法包括OnConnection、 AddNamedCommand2、Exec、QueryStatus。本文前面曾提到过命令的命名规则，我们可以由此获知某个菜单项对应的命令，这样就 可以在Add-In中使用该命令而不需要再重新发明一次轮子。

　　执行已有命令

　　先看下图。



　 　我们可以关闭当前文档，也可以关闭其它文档，就是不能关闭所有文档。有时候还是需要这个功能的，我想你应该用过Window->Close All Documents菜单项吧？它实现的正是我们所需要的，现在考虑如何使用这个已有的命令。首先得找到这个命令，按前面的规则，我们到Tools- Options里面去找（见图1），我们可以试着输入window.close，这时就可以看到了：Window.CloseAllDocuments。

　 　接下来跟前面例子类似，首先要添加一个命令，前面是把命令添加到了主菜单栏（通过“MenuBar”获取命令栏），现在要添加到另外一个命令 栏：“Easy MDI Document Window”，这名字有些奇怪（我是在所有的CommandBar中搜了好多次才找到的），在刚才添加Tools菜单项的代码下面添加如下代码：

　　C# Code - 添加CloseAllDocuments命令

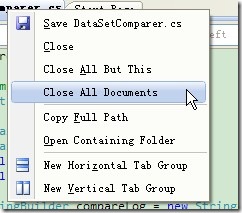
// 通过名称获取要使用的命令栏  
CommandBar mdiDocCommandBar =  
((CommandBars)\_applicationObject.CommandBars)["Easy MDI Document Window"];  
// 添加命令  
Command closeAllDocsCmd = commands.AddNamedCommand2(\_addInInstance, "CloseAllDocuments", "Close All Documents",  
"Close All Documents", false, 0, ref contextGUIDS,  
(int)vsCommandStatus.vsCommandStatusSupported + (int)vsCommandStatus.vsCommandStatusEnabled,  
(int)vsCommandStyle.vsCommandStylePictAndText, vsCommandControlType.vsCommandControlTypeButton);  
  
// 要放在Close All But This菜单项的下面  
CommandBarControl closeAllButThisCmd = mdiDocCommandBar.Controls["Close All But This"];  
// 设定菜单项的索引，注意索引是从1开始的  
int closeAllCmdIndex = (closeAllButThisCmd ==　null) ?　1 : (closeAllButThisCmd.Index +　1);  
closeAllDocsCmd.AddControl(mdiDocCommandBar, closeAllCmdIndex);

　　要使用Window.CloseAllDocuments命令只要一行代码：

　　C# Code - Execute Command

\_applicationObject.ExecuteCommand("Window.CloseAllDocuments", string.Empty);

　　现在菜单项看起来应该是这样的：



　 　赶紧试一试吧！这里再提一下，在测试、调试完成之后，如果要发布Add-In，最简单的方法是将dll和.AddIn放在[My Documents Path]Visual Studio 2008Addins，同时把那个用来测试的.AddIn文件（如NEnhancer - For Testing.AddIn）移掉。

　　本节首先介绍了命令和命令栏的概念，正是通过命令VS才可以与Add-In进行交互。然后通过两个例子解释了如何添加、执行命令，以及如何执行VS内置的命令，接下来我们就有办法操作VS的方方面面了：解决方案、项目、文档、代码等等。

# 6 操作Solution、Project和ProjectItem

大多数时候，我们在VS中进行开发，都是从打开解决方案开始，然后找到项目、打开文件进行开发。本文将介绍与此相关的内容。

　　解决方案、项目和项简介

　 　我相信，这三者对我们.NET开发人员来说是再熟悉不过的了。但它们在AOM（自动化对象模型）的表示方式跟我们想象的会不太一样。看下面的图1，其中 涉及的主要元素有解决方案、解决方案文件夹、项目、文件夹、文件。AOM通过Solution接口表示解决方案；Project接口表示解决方案文件夹和 项目；ProjectItem接口表示文件夹和文件。下面我会通过一个例子来介绍它们。

　　简单的代码生成器

　　在实际 开发中，我们往往需要采用一些特定的技术方案，这些方案就决定了程序的整体架构，表现在程序中就是一些项目、程序集、配置文件的组织。曾经有段时间很喜欢 iBATIS.NET，这里就以它的示例项目NPetshop为例进行演示。NPetshop的基本结构如下：

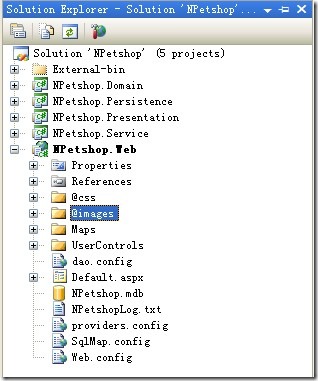


　　图1：NPetshop的程序结构

　 　它的代码可以在NPetshop Source for .NET 2.0一文中找到。External-bin里面是所需要的各个程序集，Domain是实体类，Persistence是数据访问层，Service是业 务逻辑层，Presentation是表现逻辑层，Web则负责最终的表现。如果要采用这样的解决方案进行开发，那么在项目开始我们就得花些时间来搭建整 个解决方案的框架。通过Add-In，我们可以做到“一次编写，多次使用”。

　　现在来看看如何从零开始创建一个解决方案。在本例中，大致过程是：

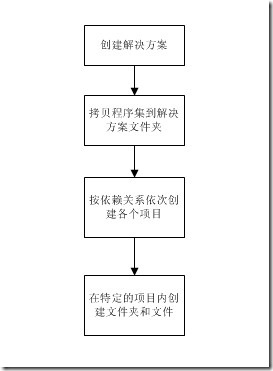


　　图2：解决方案创建过程

　　下面就来一步步实现。

　　0）添加命令

　　之前已经向NEnhancer添加了两个命令了，现在需要添加新的命令，这样重复代码又得多一些了，所以在添加命令之前，先将代码重构一下，提取出几个小方法：

　　C# Code - Connect类的辅助方法

private string GetCulturedMenuName(string englishName)  
{  
string result = englishName;  
  
try  
{  
string resourceName;  
ResourceManager resourceManager = new ResourceManager("NEnhancer.CommandBar", Assembly.GetExecutingAssembly());  
CultureInfo cultureInfo = new CultureInfo(\_applicationObject.LocaleID);  
  
if (cultureInfo.TwoLetterISOLanguageName == "zh")  
{  
System.Globalization.CultureInfo parentCultureInfo = cultureInfo.Parent;  
resourceName = String.Concat(parentCultureInfo.Name, englishName);  
}  
else  
{  
resourceName = String.Concat(cultureInfo.TwoLetterISOLanguageName, englishName);  
}  
  
result = resourceManager.GetString(resourceName);  
}  
catch  
{  
result = englishName;  
}  
  
return result;  
}  
  
private string GetCommandFullName(string cmdName)  
{  
return "NEnhancer.Connect." + cmdName;  
}  
  
private CommandBar GetCommandBarByName(string cmdBarName)  
{  
return ((CommandBars)\_applicationObject.CommandBars)[cmdBarName];  
}  
  
private void AddNamedCommand2(CommandBar cmdBar, string cmdName, string buttonText, string toolTip,   
bool useMsoButton, int iconIndex, int position)  
{  
// Do not try to add commands to a null bar  
if (cmdBar == null) { return; }  
  
// Get commands collection  
Commands2 commands = (Commands2)\_applicationObject.Commands;  
object[] contextGUIDS = new object[] { };  
  
try  
{  
// Add command  
Command command = commands.AddNamedCommand2(\_addInInstance, cmdName, buttonText, toolTip,  
useMsoButton, iconIndex, ref contextGUIDS,   
(int)vsCommandStatus.vsCommandStatusSupported + (int)vsCommandStatus.vsCommandStatusEnabled,   
(int)vsCommandStyle.vsCommandStylePictAndText, vsCommandControlType.vsCommandControlTypeButton);  
if (command != null && cmdBar != null)  
{  
command.AddControl(cmdBar, position);  
}  
}  
catch (ArgumentException)  
{  
// Command already exists, so ignore the exception.  
}  
}

　　有了这些方法，再添加命令就变得更简单：

　　C# Code - 添加命令

OnConnection()  
{  
  
AddNamedCommand2(toolsPopup.CommandBar, NPETSHOP\_SLN\_GENERATOR\_COMMAND\_NAME, "Generate NPetshop Sln",  
"Generate NPetshop Solution", true, 59, toolsPopup.Controls.Count + 1);   
  
}  
  
QueryStatus()  
{  
  
else if (commandName == GetCommandFullName(NPETSHOP\_SLN\_GENERATOR\_COMMAND\_NAME))  
{  
status = (vsCommandStatus)vsCommandStatus.vsCommandStatusSupported | vsCommandStatus.vsCommandStatusEnabled;  
return;  
}  
}  
  
Exec()  
{  
  
else if (commandName == GetCommandFullName(NPETSHOP\_SLN\_GENERATOR\_COMMAND\_NAME))  
{  
GenerateNPetshopSln();  
  
handled = true;  
return;  
}  
}

　　1）创建解决方案

　　唯一需要用户输入的信息是解决方案的名称（这里假定各项目的名称都以此为前缀）和目录，这需要创建一个简单的窗体（NPetshopSlnGenerator）来收集信息。

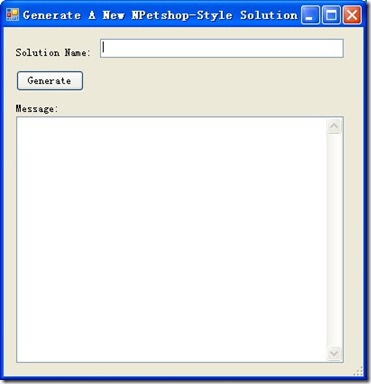


　　图3：Generator窗体

　　在AOM中，表示解决方案的类型是EnvDTE.Solution（以及其后续版本Solution2、Solution3），它的重要方法和属性有：

　　AddFromFile()：添加既有项目；

　　AddFromTemplate()：根据指定项目模板添加新的项目；

　　AddSolutionFolder()：添加解决方案文件夹；

　　Close()：关闭解决方案；

　　Create()：创建一个空的解决方案；

　　Open()：打开一个解决方案；

　　Remove()：移除一个项目；

　　SaveAs()：保存解决方案；

　　FileName：解决方案文件名；

　　FullName：解决方案文件全名；

　　IsOpen：表示解决方案是否已打开；

　　Projects：解决方案内的项目集合；

　　SolutionBuild：用于管理解决方案的Build过程。

　　Solution的完整成员列表请参看MSDN。要访问VS中的当前解决方案，可使用DTE.Solution属性，下面是创建解决方案的代码：

　　C# Code - 创建解决方案

Solution3 sln = (DTEObject.Solution as Solution3);  
sln.Create(currentSlnPath, slnName);

　　一旦调用了Create方法，VS就会打开新创建的解决方案。

　　2）创建解决方案文件夹

　　在这一步，首先要把所需的程序集等文件拷贝到新解决方案所在目录，然后创建解决方案文件夹，将文件添加进去：

　　C# Code - 创建SolutionFolder

Project sfProj = sln.AddSolutionFolder(ExternalBinDirectoryName);  
foreach (string file in Directory.GetFiles(ExternalBinPath))  
{  
sfProj.ProjectItems.AddFromFile(file);  
}

　　很简单，就是调用AddSolutionFolder方法。可以看到，解决方案文件夹被看作是一种Project，通过其ProjectItems添加文件（关于Project的更多内容请看下面）。

　　3）创建各个项目

　　由于项目之间会有依赖关系，所以考虑按依赖关系进行创建，也就是Domain->Persistence->Service->Presentation->Web。

　　表示项目的类型为EnvDTE.Project，它的主要方法和属性有：

　　Delete()：将项目从解决方案内移除；

　　Save()：保存项目；

　　SaveAs()：保存解决方案、项目和文件项；

　　FileName/FullName：项目文件的名称/全名；

　　Kind：表示项目类型的GUID值；

　　ProjectItems：项目所包含的项的集合；

　　Project的完整成员列表请参看MSDN。创建Domain项目的代码大致如下：

　　C# Code - 创建项目

string classLibProjTemplatePath =  
sln.GetProjectTemplate("ClassLibrary.zip", "CSharp");  
string domainProjName = slnName + "." + "Domain";  
sln.AddFromTemplate(classLibProjTemplatePath, Path.Combine(currentSlnPath, domainProjName),  
domainProjName, false);  
Project domainProj = GetProjectByName(sln, domainProjName);  
VSProject vsDomainProj = domainProj.Object as VSProject;  
vsDomainProj.References.Add(Path.Combine(ExternalBinPath, "IBatisNet.Common.dll"));

　 　Domain是类库项目，它有对应的项目模板，此时可以使用Solution.GetProjectTemplate方法获取模板路径，然后使用 AddFromTemplate方法添加项目。本来AddFromTemplate方法返回一个Project对象，按道理应该就是新创建的项目，但是 MSDN上说对于C#和VB.NET项目来说，返回值为null！只好自己写一个方法GetProjectByName了，也就是循环各个项目，根据名称 找到匹配的项目。最后，要给Domain项目添加对IBatisNet.Common.dll的引用，这里要使用VSProject的 References集合来添加。

　　接下来是Persistence项目，过程与Domain基本相同，但是要给它添加对Domain项目的引用，此时应使用References.AddProject方法：

　　C# Code - 添加项目引用

vsPersistProj.References.AddProject(domainProj);

　 　照这样下去，剩下的Service、Presentation、Web项目也可以顺利创建，只是在创建Web项目的时候要用 WebApplication项目模 板：Solution.GetProjectTemplate("WebApplicationProject.zip", "CSharp")。

　　4）创建文件夹和文件

　　在AOM中，文件夹和文件统一表示为ProjectItem类型。它的主要方法和属性为：

　　Delete()：从项目中删除该项；

　　ExpandView()：展开解决方案管理器来显示该项；

　　Open()：打开该项；

　　Remove()：从该项所包含的项中移除一项；

　　Save()/Save()：保存

　　Kind：该项的类型；

　　Name：该项的名称；

　　ProjectItems：该项所包含的其它项；

　　ProjectItem的完整成员列表请参看MSDN。Project接口有一个ProjectItems集合属性，可以访问项目所包含的项，同时该属性还可用来添加新的项。在本例中，要添加几个文件夹和文件：

　　C# Code - 添加文件夹和已有文件

webProj.ProjectItems.AddFolder("Maps", Constants.vsProjectItemKindPhysicalFolder);  
webProj.ProjectItems.AddFromFileCopy(  
Path.Combine(GetAddinPath(), @"ibatis-configdao.config"));

　　这里使用了AddFolder和AddFromFileCopy方法。如果需要添加新的文件，也很简单，类似于项目，这里要使用项模板：

　　C# Code - 根据模板添加文件

string templatePath =  
sln.GetProjectItemTemplate("Interface.zip", "CSharp");  
webProj.ProjectItems.AddFromTemplate(templatePath, "MyInterface.cs");

　　关于ProjectItems的更多信息请参看这里。至此我们就完成了一个具备初步功能的代码生成器。

　 　在VS中，解决方案、项目和文件（夹）是其基本组织形式，也是我们开发人员最为熟悉的元素了，本文介绍的就是与此相关的基本操作。这些操作是通过开发一 个具备初步功能的代码生成器来演示的，其中可以了解如何从解决方案开始，自上而下逐步生成项目、文件夹和文件。但是需要说明的是，这里没有对 Solution Explorer的操作，也没有涉及到对文件内容的操作，这些都将在后续的章节中介绍。

# 7 操作Solution Explorer

之前介绍了对于Solution、Project、ProjectItem的基本操作。 可以认为它们面向的是解决方案内容的物理（文件）表示，我们需要使用VS提供的解决方案管理器（Solution Explorer）来管理它们。毫无疑问，解决方案管理器是VS中最重要的UI元素之一，本文将介绍对它的操作。

　　工具窗口内的层次结构

　 　如果你观察一下解决方案管理器和服务器管理器（Server Explorer），就会发现它们都使用树形结构来表现背后的数据。在AOM中，UIHierarchy、UIHierarchyItems和 UIHierarchyItem用于表示这样的层次结构。UIHierarchy表示根节点，它的UIHierarchyItems集合表示其所包含的第 一级子节点（UIHierarchyItem），每一个UIHierarchyItem同时也有UIHierarchyItems属性，如此递归下去。这 种结构很像它们所表示的数据：Solution、Project以及ProjectItem。在使用这些对象之前，先大致了解一下它们的主要成员：

　　1）UIHierarchy

　　Parent：节点对象的父节点；

　　SelectedItems：当前节点选中的子节点集合；

　　UIHierarchyItems：当前节点的子节点集合；

　　DoDefaultAction()：对节点进行默认操作，类似于进行双击或按下回车键；

　　GetItem()：按指定路径返回一个子节点；

　　SelectDown()：选中当前选中节点的下个节点；

　　SelectUp()：选中当前选中节点的上个节点；

　　更多信息请参看MSDN。

　　2）UIHierarchyItems集合

　　Expanded：获取或设置所表示的节点是否已展开；

　　Parent：节点集合的父节点；

　　Item()：返回集合中的一项；

　　更多信息请参看MSDN。

　　3）UIHierarchyItem

　　IsSelected：获取节点是否被选中；

　　Name：节点对象的名称；

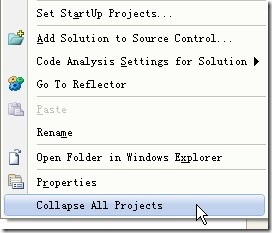
　　Select()：选中节点；

　　更多信息请参看MSDN。

　　有了这些知识，我们现在有能力去探索对解决方案管理器的操作了。

　　CollapseAllProjects示例

　　项目刚开始的时候，项目的数量也许还不太多，随着程序规模的增大，项目数量也会不断增加，这时要找到某个项目或者某个文件，就变得越来越麻烦，你得先把大量的项目折叠起来。如果有一个命令，可以快速地折叠起所有项目，就方便多了：



　　这里的思路很简单，只要找到所有的项目节点，依次查看每个项目，如果项目展开了，就把它折叠起来。

　　0）添加命令

　　之前我们曾添加过CloseAllDocuments和NPetshopSlnGenerator命令，它们分别加在文本编辑器的标签和Tools菜单上，这里的过程没什么不同：

　　C# Code - 添加CollapseAllProjects命令

OnConnection()  
{  
Visual Studio 2008 可扩展性开发(六):操作Solution Explorer  
//　Get　"Solution　Explorer"　command　bar  
CommandBar　slnCommandBar　=　GetCommandBarByName("Solution");  
//　Add　a　new　command  
AddNamedCommand2(slnCommandBar,　COLLAPSE\_ALL\_PROJECTS\_COMMAND\_NAME,  
"Collapse　All　Projects",　"Collapse　All　Projects",　false,　0,　slnCommandBar.Controls.Count　+　1);  
}  
  
QueryStatus()  
{  
Visual Studio 2008 可扩展性开发(六):操作Solution Explorer  
else　if　(commandName　==　GetCommandFullName(COLLAPSE\_ALL\_PROJECTS\_COMMAND\_NAME))  
{  
status　=　(vsCommandStatus)vsCommandStatus.vsCommandStatusSupported　|　vsCommandStatus.vsCommandStatusEnabled;  
return;  
}  
}  
  
Exec()  
{  
Visual Studio 2008 可扩展性开发(六):操作Solution Explorer  
else　if　(commandName　==　GetCommandFullName(COLLAPSE\_ALL\_PROJECTS\_COMMAND\_NAME))  
{  
CollapseAllProjects();  
  
handled　=　true;  
return;  
}  
}

　　这里通过“Solution”找到解决方案节点的上下文菜单。

　　1）找到所有项目节点

　　C# Code - 查找所有的项目节点

///　<summary>  
///　Solution　Explorer　Window  
///　</summary>  
protected　UIHierarchy　SolutionExplorerNode  
{  
get  
{  
return　\_applicationObject.ToolWindows.SolutionExplorer;  
}  
}  
  
///　<summary>  
///　Gets　project　nodes.  
///　</summary>  
public　List<UIHierarchyItem>　GetProjectNodes(Solution　solution)  
{  
string　solutionName　=　solution.Properties.Item("Name").Value.ToString();  
return　GetProjectNodes(SolutionExplorerNode.GetItem(solutionName).UIHierarchyItems);  
}  
  
///　<summary>  
///　Gets　the　project　nodes.  
///　</summary>  
///　<param　name="root">The　root.</param>  
///　<returns></returns>  
public　List<UIHierarchyItem>　GetProjectNodes(UIHierarchyItems　topLevelItems)  
{  
List<UIHierarchyItem>　projects　=　new　List<UIHierarchyItem>();  
foreach　(UIHierarchyItem　item　in　topLevelItems)  
{  
if　(IsProjectNode(item))  
{  
projects.Add(item);  
}  
else　if　(IsSolutionFolder(item))  
{  
GetProjectNodesInSolutionFolder(projects,　item);  
}  
}  
  
return　projects;  
}  
  
private　void　GetProjectNodesInSolutionFolder(List<UIHierarchyItem>　projects,　UIHierarchyItem　item)  
{  
if　(!IsSolutionFolder(item))　{　return;　}  
  
foreach　(UIHierarchyItem　subItem　in　item.UIHierarchyItems)  
{  
if　(IsProjectNode(subItem))  
{  
projects.Add(subItem);  
}  
}  
}  
  
private　bool　IsSolutionFolder(UIHierarchyItem　item)  
{  
return　((item.Object　is　Project)　&&  
((item.Object　as　Project).Kind　==　ProjectKinds.vsProjectKindSolutionFolder));  
}  
  
private　bool　IsProjectNode(UIHierarchyItem　item)  
{  
return　IsDirectProjectNode(item)　||　IsProjectNodeInSolutionFolder(item);  
}  
  
private　bool　IsDirectProjectNode(UIHierarchyItem　item)  
{  
return　((item.Object　is　Project)　&&　((item.Object　as　Project).Kind　!=　ProjectKinds.vsProjectKindSolutionFolder));  
}  
  
private　bool　IsProjectNodeInSolutionFolder(UIHierarchyItem　item)  
{  
return　(item.Object　is　ProjectItem　&&　((ProjectItem)item.Object).Object　is　Project　&&  
((Project)((ProjectItem)item.Object).Object).Kind　!=　ProjectKinds.vsProjectKindSolutionFolder);  
}

　　也许比预想的要复杂些，主要的原因是解决方案文件夹的存在，解决方案文件夹本身也被看作Project对象，同时它又可以包含其它真正的项目，所以在查找项目的时候要分两种情况。先查找解决方案下面的项目，然后再查找解决方案文件夹下面的项目。

　　2）折叠所有项目节点

　　C# Code - 折叠所有项目节点

private　void　CollapseAllProjects()  
{  
Solution　sln　=　\_applicationObject.Solution;  
List<UIHierarchyItem>　projects　=　GetProjectNodes(sln);  
foreach　(UIHierarchyItem　item　in　projects)  
{  
CollapseProject(item);  
}  
}  
  
private　void　CollapseProject(UIHierarchyItem　project)  
{  
if　(project.UIHierarchyItems.Expanded)  
{  
if　(IsDirectProjectNode(project))  
{  
project.UIHierarchyItems.Expanded　=　false;  
}  
else　if　(IsProjectNodeInSolutionFolder(project))  
{  
project.Select(vsUISelectionType.vsUISelectionTypeSelect);  
SolutionExplorerNode.DoDefaultAction();  
}  
}  
}

　　这里就简单了，对于每个项目，通过Expanded属性判断它是否已展开，如果是的话将其折叠起来，此时也要分两种情况进行考虑。

　　以后就不用再为那些包含数十个项目的解决方案发愁了:)

在解决方案、项目和项之后，本文介绍了对解决方案管理器的操作，现在我们有办法来解决这些方面的问题了。

## 8 关于用户界面的种种

Add-In大多需要一定的用户界面，前面的随笔中采用的主要方式是菜单和自定义窗体。对于菜单，可以是VS的主菜单，也可 以是在某个特定的上下文菜单，比如编辑器；对于自定义窗体，用起来也很简单，就像在Windows应用程序中添加一个窗体一样。本文将介绍关于用户界面的 其它内容，包括：

　　1) 访问VS中的各个窗口，如Output窗口、Command窗口等；

　　2）创建自定义的工具窗口

　　关于VS中的窗口

　　我们在VS中会遇到各种窗口，如Solution Explorer、Toolbox、Output Window或者是打开的文本编辑器，虽然它们看起来各不相同，但是在AOM模型中它们仍然有共通之处。

　　1）访问VS中的窗口

　　首先想到的一个问题，如何获取上述窗口的引用？DTE2接口有一个Windows集合属性，通过它可以访问某个特定的窗口。

　　该接口实现了IEnumerable，所以我们可以使用foreach语句对它进行遍历：

　　C# Code - 遍历VS的窗口集合

foreach (Window win in \_applicationObject.Windows)  
{  
\_applicationObject.ToolWindows.OutputWindow.ActivePane.OutputString(win.Caption + Environment.NewLine);  
}

　　另外，通过Windows集合的Item方法，我们可以使用数字索引或窗口的标题获取某个特定的窗口，比如：

　　C# Code - 使用数字索引获取Window引用

Window win = \_applicationObject.Windows.Item(1);  
win.SetFocus();

　　这种方式的问题在于，窗口的数字索引是不确定的，而窗口的标题也是如此。对于Solution Explorer，当我们打开一个解决方案时，它的标题是“Solution ‘NEnhancer’ (6 projects)”这样的格式。

　　幸运的是，对于VS内的工具窗口（Solution Explorer、Toolbox、Output Window等等），它们都有一个唯一的索引，这些索引是GUID，这样就很容易获取想要的窗口了：

　　C# Code - 使用GUID访问Window

Window slnWin = \_applicationObject.Windows.Item(EnvDTE.Constants.vsWindowKindSolutionExplorer);  
slnWin.SetFocus();

　　GUID显然难以记住，所以这里要使用EnvDTE.Constants中定义的常量，这样不仅方便获取窗口，性能也要比使用数字或标题索引好一些（你可以试一下获取Server Explorer）。

　　现在有办法获取窗口的引用了，不过Window接口毕竟是针对所有窗口的通用接口，如果要针对某个窗口完成一些特定的任务，就需要考虑别的方法了。

　　2）Window接口的Object属性

　　使用Object属性可以获取特定于某个窗口的对象，看下面的例子：

　　C# Code - 获取工具窗口所承载的对象

Window taskListWin = \_applicationObject.Windows.Item(EnvDTE.Constants.vsWindowKindTaskList);  
TaskList taskList = taskListWin.Object as TaskList;  
MessageBox.Show(string.Format("You have {0} tasks.", taskList.TaskItems.Count));

　 　关于这个Object属性，也许在看了后面的自定义工具窗口就很容易明白了。这里通过TaskList接口访问Task List窗口。其它的窗口如Command Window、Error List等与此类似。对于这些最重要的工具窗口，AOM提供了更为方便的API。

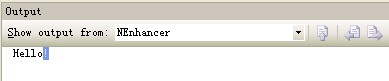
　　3）ToolWindows属性

　　通过EnvDTE的ToolWindows属性，我们可以快速地访问那些重要的工具窗口，比如OutputWindow：

　　C# Code - 操作OutputWindow

OutputWindow outputWin = \_applicationObject.ToolWindows.OutputWindow;  
outputWin.OutputWindowPanes.Add("NEnhancer");  
outputWin.OutputWindowPanes.Item("NEnhancer").OutputString("Hello!");

　　运行这段代码后，看起来是这样的：



　　4）UIHierarchy对象

　 　更为特殊的是，对于一些用于表现树形结构的窗口——如Solution Explorer、Server Explorer和Macro Explorer——它们背后的对象是UIHierarchy，与之相关的还有UIHierarchyItems和UIHierarchyItem对象， 这个在前面操作Solution Explorer时已经看到过了，不再赘述。

　　有了这些准备知识，操作各个工具窗口就很容易了。下一步，我们来考虑如何添加自己的工具窗口。

　　创建自定义工具窗口（ToolWindow）

　　VS内置的工具窗口毕竟是有限的，总有时候，我们需要创建自己的工具窗口。可以采用两种方式，一是使用ActiveX控件，二是使用.NET用户控件。这里将介绍第二种方式的用法。

　　我们知道VS内置的命令对应着大量的快捷键，不太可能一下子把它们完全记住。如果某个快捷键忘记了，如果在VS中可以查阅就方便了。所以这里考虑创建一个工具窗口来完成这个功能。这里最关键的是Windows2.CreateToolWindow2方法，它的签名是：

　　C# Code - CreateToolWindow2方法签名

Window CreateToolWindow2(AddIn Addin, string Assembly, string Class,  
string Caption, string GuidPosition, ref object ControlObject);

　　各个参数的作用是：

　　Addin：用于创建工具窗口的外接程序的实例，一般为当前的Addin；

　　Assembly：包含该用户控件的程序集的完整名称或文件路径；

　　Class：实现该用户控件的类的全名；

　　Caption：要在新工具窗口中显示的标题；

　　GuidPosition：新窗口的唯一标识符，可用于在 Windows 集合中查找窗口；

　　ControlObject：要在新工具窗口中承载的用户控件。

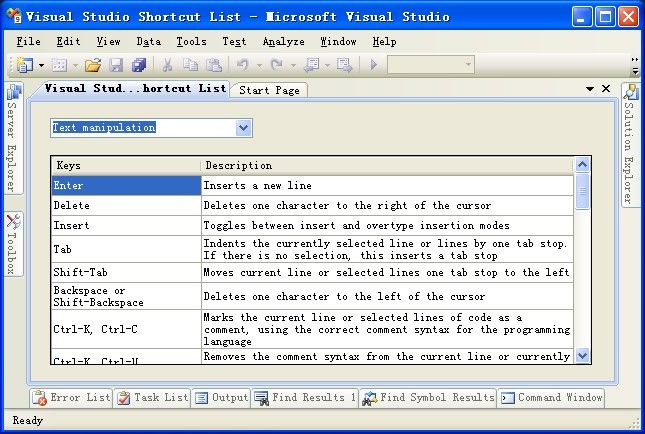
　 　通过参数的信息大体可以了解到，我们首先得定义一个用户控件，该控件将被承载在新的工具窗口中。需要注意的是在调用 CreateToolWindow2 创建一个工具窗口之前，您应将用户控件 (ControlObject) 移动到与外接程序的程序集中，或设置用户控件上的所有属性，以使其对 COM 是完全可见的。否则控件不会正确封送，而且CreateToolWindow2将返回一个空值。现在可以用下面的代码来创建工具窗口了：

　　C# Code - 创建自定义的工具窗口

object programmingObj = null;  
string guidString = "{41F8DEA8-EB07-45b7-9B1D-EB969DC43EC5}";  
Windows2 windows2 = \_applicationObject.Windows as Windows2;  
Assembly asm = Assembly.GetExecutingAssembly();  
shortcutListWindow = windows2.CreateToolWindow2(\_addInInstance, asm.Location,  
"NEnhancer.ShortcutListControl", "Visual Studio Shortcut List",  
guidString, ref programmingObj);

　　这里的 ShortcutListControl是一个用户控件，它里面实现了查看VS快捷键的功能。按我的理解，VS将该控件的一个实例加载到一个工具窗口中， 这样工具窗口就可以使用控件所实现的功能；另一方面，VS可以不考虑新窗口中实现的是什么功能，对所有工具窗口一视同仁。这样自定义的工具窗口也可以像内 置的工具窗口那样操作，如floating、dockable等等。

　　这段代码要放在哪里呢？通常我们希望VS在加载Add-In时创建工具窗口，可以使用OnConnection方法，另外可以采取比较安全的方 式，就是判断connectMode参数，如果它的值为ext\_cm\_AfterStartup，才进行创建，这时VS主窗口已经加载完毕了。在创建之 后，在View菜单中添加一个菜单项来显示该窗口，此时只要将Window实例的Visible属性设置为true就可以了。（可以在本文的最后下载到这 些代码）



[图片看不清楚？请点击这里查看原图（大图）。](http://img.ddvip.com/2009_05_11/1242032684_ddvip_6969.jpeg)

　　从图中可以看到正以标签页的形式显示窗口，在窗口中按分类显示快捷键及其描述信息，所有这些信息来自dofactory。

　　关于使用ActiveX控件添加ToolWindow，可以参考Working with Microsoft Visual Studio 2005一书的228页。

　　可以下载代码，也可以下载可运行的Add-In（解压缩后将文件放在[My Documents Path]Visual Studio 2008Addins下）。

　　我们身在何处？

　　本文介绍了VS工具窗口相关的内容。首先是如何访问、操作那些内置的工具窗口，如Solution Explorer、Output Window等等；然后介绍了如何创建自定义的工具窗口，在这个窗口中可以查看VS的快捷键列表。

　　出处：[http://anderslly.cnblogs.com](http://anderslly.cnblogs.com/)

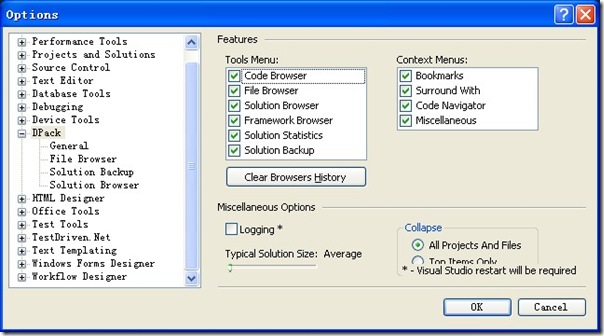
　关于用户界面的种种（上）一文介绍了VS中的工具窗口。关于用户界面还有更多的内容，比如选项页（Options Page）、菜单图标，具体来说，本文介绍的内容将包括：

　　1）创建自定义的选项页；

　　2）设置菜单项的图标；

　　创建自定义的选项页

　　在安装了DPack后，可以在VS的选项页（Tools->Options）中看到新添加的一项：



　　。

　　这看起来很专业。很多时候我们开发的Add-In需要进行一定的配置，选项页是最佳选择：VS的用户一般习惯于在此处进行各种配置，所以把配置界面放在其它地方就显得突兀了。

　 　对Add-In来说，我们需要实现IDTExtensibility2接口，这样可以将其交由VS来管理，选项页的实现与此类似，不过这里的接口是 IDTToolsOptionsPage（从Add-In和选项页的开发过程中，我们也可以考虑如何开发自己的“插件式”应用程序）。具体来说，分为两 步：

　　1）创建用户控件（User Control）来实现IDTToolsOptionsPage

　　该接口有5个方法，具体可以参看这里。其中最重要的是OnAfterCreated()和OnOK()，前者在VS呈现选项页时执行，后者在用户点击OK时执行。

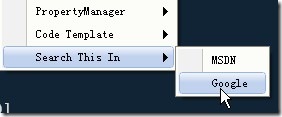
　　2）在.Addin文件中添加新选项页的配置

　　此时需要节点

　　下面通过一个简单的例子来介绍具体的实现过程。

　　搜索编辑器中的选中文本

　　我们在编写代码的时候，经常需要搜索一些相关的内容，比如某个类型的信息，这时就复制类的名称，打开浏览器，打开Google，搜索那个类，有时可能会在其它搜索引擎或MSDN搜索。如果能把搜索的过程集成到VS中，是不是就方便了呢？



　　实现这个功能很简单。只要获取当前选中的文本，然后通过默认浏览器打开相应的URL就可以了：

　　C# Code - Search Text(version 1)

private　static　readonly　string　MSDN\_URL\_FORMAT　=  
"http://social.msdn.microsoft.com/Search/en-AU/?query={0}&ac=8";  
private　static　readonly　string　GOOGLE\_URL\_FORMAT　=  
"http://www.google.cn/search?source=igchina&hl=zh-CN&q={0}&btnG=Google+%E6%90%9C%E7%B4%A2&aq=f";  
  
private　void　LaunchUrlInDefaultBrowser(string　url)  
{  
System.Diagnostics.Process.Start(url);  
}  
  
private　void　SearchInMsdnCmdEvent\_Click(object　CommandBarControl,　ref　bool　Handled,　ref　bool　CancelDefault)  
{  
string　url　=　string.Format(MSDN\_URL\_FORMAT,　helper.GetSelectedText());  
LaunchUrlInDefaultBrowser(url);  
}  
  
private　void　SearchInGoogleCmdEvent\_Click(object　CommandBarControl,　ref　bool　Handled,　ref　bool　CancelDefault)  
{  
string　url　=　string.Format(GOOGLE\_URL\_FORMAT,　helper.GetSelectedText());  
LaunchUrlInDefaultBrowser(url);  
}

　 　可以在代码里面把URL作为常量来写，不过做成可配置的要好一些，这样修改时无须编译，而且可以添加新的搜索引擎，现在通过选项页来实现。首先得创建一 个用户控件SearchTextOptionsPage，大体思路是在OnAfterCreated方法中根据xml中的配置动态生成控件供用户设置，并 在OnOK方法中保存用户所做的设置：

　　C# Code - 实现IDTToolsOptionsPage接口

public　void　OnAfterCreated(DTE　DTEObject)  
{  
List

　　SearchEngineConfigManager是对xml文件进行操作的类。接下来要把选项页注册一下，在.Addin文件的根节点下添加如下内容：

　　XML Code - 注册Options Page

　　现在选项页看起来是这样的：



　　设置菜单项的图标

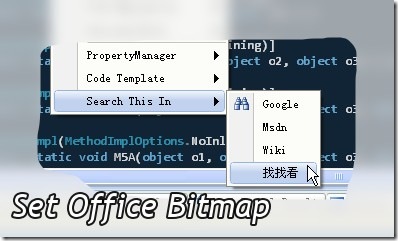
　　在VS的内置的菜单项中，相当一部分拥有自己的图标，这些图标使得所对应的命令更为直观，现在来看看如何添加这样的图标。

　 　在此之前，我们曾经使用过两种方法添加菜单项，一是AddNamedCommand2方法，它的两个参数MSOButton、Bitmap共同指定了是 否使用Office位图以及使用哪一个位图，如果使用Office位图的话，就将MSOButton设置为true，并通过Bitmap指定所使用位图的 索引，如果是59，那就是经典的笑脸了。

　　第二种方法是通过CommandBarButton控件添加，比如：

CommandBarButton　searchCmd　=　helper.AddButtonToPopup(searchThisPopup,　searchThisPopup.Controls.Count　+　1,  
e.Name,　"Search　in　"　+　e.Name);

　 　对于CommandBarButton来说，它的FaceId属性可用来指定Office位图，其取值跟上面的Bitmap参数是一样的，这样我们可以 在添加CommandBarButton时设定图标了。如果要了解所有的位图索引信息，可以参考这个页面Office CommandBarButton FaceId。



　　最后，我们还可以使用自己创建的图标，具体过程可以参考Displaying custom bitmap for VS add-in，或者《Working with Microsoft Visual Studio 2005》一书的148页，此处不再赘述。

　　本文着重介绍了如何为Add-In添加选项页（Options Page），这样可以为用户提供较为专业的配置界面，然后简单介绍了如何设置菜单项的图标。这一次的例子是SearchTextInWeb，可以直接打开浏览器搜索VS的选中文本，相信会你带来一些方便。