

# TR0217插件框架用户手册

唐锐

更新时间: 05/23/12 22:34:02

## 目录

---

- [目录](#)
- [前言](#)
  - [本手册使用的格式约定](#)
  - [手册内容的组织结构](#)
  - [您预先需要掌握那些知识](#)
  - [怎样使用本手册](#)
- [第一部分 概览](#)
  - [首先，您想问？](#)
    - [它是什么类型的插件框架？](#)
    - [为什么我会需要这样一个插件框架？](#)
  - [更进一步的信息](#)
    - [它能够运行在那些平台上？](#)
    - [有那些成功的应用案例？](#)
    - [需要付钱吗？从哪里可以获得？](#)
    - [如何获得技术支持？](#)
  - [授权与责任声明](#)
    - [授权](#)
    - [责任声明](#)
  - [关于插件框架的一般性问题](#)
    - [插件框架的基本任务](#)
    - [不得不考虑的界面设计原则](#)
    - [使用任何一款插件框架都需要注意的问题](#)
    - [插件框架还需要完成那些任务](#)
- [第二部分 使用指南](#)
  - [TR0217插件框架入门](#)
    - [TR0217插件框架简介](#)
    - [从Hello, world! 开始](#)
    - [显示在停靠窗口中的Hello, world!](#)
    - [事实上，我想说Hi, beauty!](#)
    - [我要问候任何一个想打招呼的对象](#)
    - [插件间的两种交互方式](#)
    - [内容回顾](#)
  - [TR0217插件框架应用进阶](#)
    - [规划应用程序的目录结构](#)
    - [如何使用日志系统](#)
    - [设计插件的接口](#)
    - [怎么维护界面逻辑](#)
    - [文档窗体、工具窗体和对话框](#)
    - [创建文档模型实现文档窗口](#)
    - [何时发布更新界面事件](#)
    - [和AddIn.Gui交互，创建收藏菜单和收藏工具条](#)
  - [高级主题](#)
    - [创建带有Splash Screen和登陆窗体的宿主](#)
    - [界面逻辑维护与多线程](#)
    - [如何建立完备的服务集合](#)
    - [使用混淆器对插件进行版权保护时的注意事项](#)
    - [深入到此插件框架的内部实现](#)
    - [如何替换UI插件](#)
- [第三部分 编程参考](#)
  - [AppFrame](#)
    - [属性](#)
    - [方法](#)
    - [事件](#)
  - [ServiceCollection/IServiceCollection](#)
    - [属性](#)
    - [方法](#)

- [事件](#)
  - [UIService/IUIService](#)
    - [属性](#)
    - [方法](#)
  - [AddInParser](#)
    - [属性](#)
    - [方法](#)
  - [ILoginDialog](#)
    - [属性](#)
    - [方法](#)
  - [ISplashScreen](#)
    - [方法](#)
  - [LoadAddInEventArgs](#)
    - [属性](#)
  - [LoadMainFormEventArgs](#)
    - [属性](#)
  - [UpdateUiElemEventArgs](#)
    - [属性](#)
  - [附录](#)
    - [推荐的第三方界面组件](#)
- 

# 前言

## 本手册使用的格式约定

为了更加清晰的组织文档内容，本手册采用如下的内容约定。

句子中的**粗体关键字**表示强调。

*缩进一个Tab宽度的斜体段落表示引用*

*（缩进一个Tab宽度的斜体且用括号包围的段落表示提示）*

*（缩进一个Tab宽度的斜体加粗的且用括号包围的段落表示重要提示）*

代码片断缩进一个Tab宽度，其格式如下所示

```
static void Main(string[] args)
{
    AppFrame app = new AppFrame();
    app.Run();
}
```

存在于句子中的这种格式表示**类名**、**实例名**、**方法名**，如AppFrame。

[这表示一个链接](#)，默认在当前窗口中打开这个链接。如果它不是链接到本手册某一章节的内容的话，请不要随意单击它，以防干扰您阅读的流畅性。

## 手册内容的组织结构

为了方便您的使用，将本手册分为五部分。

**前言**部分的目的在于让您能够更好的使用本手册。其包含的内容有文档格式约定、手册的组织结构、首先需要掌握的知识以及为不同的角色推荐了不同的阅读方案。

**概览**部分从总体上对此插件框架进行了介绍，并且讲述了一些必须了解的关于此插件框架使用方面的内容，如怎么获取免费授权、怎么获得技术支持等。为了让您能够将此插件框架或者其它插件框架的作用发挥到极致，本部分还讲述了使用任何一款插件框架都需要注意的问题。

**使用指南**部分循序渐进地讲述了如何使用此插件框架创建完善的专业应用系统。首先以*Hello, World!*为例，讲解了如何创建插件，如何将插件注册到系统中以使用户能够调用插件提供的功能，插件之间怎么交互。

接下来一节以*Mini Internet Explorer*为例详细说明了如何使用此插件框架创建出高专业程度的应用系统。这一节包含的内容有如何规划应用程序的目录结构、如何在插件中调用日志系统、如何分拆提供给用户的功能至不同的插件、如何高效的更新界面逻辑、如何创建并显示多样化的界面。

高级主题部分涉及到一些更加深入的内容:

- 当需要加载的插件有很多时, 如何制作闪屏来改进用户体验。
- 当需要控制登入系统的用户的权限时, 如何在系统启动之初进行用户验证并将验证结果传递到其它插件。
- 关于多线程的若干问题。
- 如何创建完备的服务集合从而真正做到面向用户需求的复用。
- 如何合理的设计插件从而易于使用混淆器进行版权保护。
- 深入到此插件框架的内部实现层面上对其进行了解。
- 如何替换UI插件以使系统轻易具有不同界面风格。这是本插件框架与众不同的优点之一。

编程参考部分供用户日常使用时查询类及其成员方法的使用细节。

附录部分对常见的第三方Winform界面组件进行了点评, 以供您在选取界面组件时进行参考。此外需要说明的是本部分的内容会根据用户的提议及实际需要进行增减。

## 您预先需要掌握那些知识

这一款运行在.NET 2.0及更高平台上的插件框架, 当然首先您应该熟悉.NET平台, 熟悉C# 开发语言。此外还希望您对软件复用有深入的思考。最好还能够熟练使用并深入了解过某款插件结构的软件, 对插件有较深入的认识; 否则可以先阅读一下概览部分的第一节。

## 怎样使用本手册

作为一个开发人员, 如果您不熟悉.NET平台的话, 推荐您首先阅读一下《Programming C#》或者直接阅读《Essential C#》。如果您已经有了.NET平台的开发基础, 即使没有对软件复用做过太多的思考, 没有了解过什么插件框架, 也可以轻松的通过阅读本手册学会此插件框架的使用并且能够获得软件复用和插件框架的一般性知识; 但是首先您必须结合网络上的资料仔细阅读概览部分第一节——“首先, 您想问? ”。一般情况下, 开发人员只需要粗略的阅读概览部分的内容, 然后根据自己的理解的情况安排阅读使用指南部分的进度。编程参考部分用于平时查阅。

如果您是开发小组的领导, 还需要对概览部分的最后一节——“关于插件框架的一般性问题”和使用指南部分的最后一节——“高级主题”稍加注意。

如果您是技术主管, 可以首先通读一下概览部分和使用指南那的最后一节——“高级主题”。此外, 对附录部分也应该稍加留意。

# 第一部分 概览

## 首先, 您想问?

### 它是什么类型的插件框架?

作为有经验的电脑用户和软件工程师,您已经见识过各种各样的插件框架了。比如暴风影音的解码器、Photoshop的滤镜、Vim的语法高亮、Visual Studio、Eclipse、Sharpdevelop.....它们都用到了插件框架, 并且给用户带来了很大的方便。按照其能够达到的目标, 我们可以将其归为如下三类:

1. 使软件某方面的功能能够被灵活地扩展, 如暴风影音的解码器、Photoshop的滤镜、gVim的语法高亮。
2. 允许用户通过编写符合其规范的插件对系统功能进行扩展并且提供了一定程度定制能力, 如Visual Studio。
3. 整个系统由一些插件和一个插件宿主组成从而提供了更加灵活的定制能力和可扩充性, 如Sharpdevelop, Eclipse。

虽然您会说, 他们的优点显而易见并且满足了各自的应用情形; 但是我们不得不承认它们都有不同层次的不足。第一种插件形式的缺点是很明显的, 它仅能提供很有限的扩充能力, 事实上它还算不上框架。对于纯粹的用户来说, 第二种和第三种插件框架之间几乎没有什么区别。当查看了它们的源码或者编写了一个对其进行扩展的插件后, 其中的区别就能够被察觉出来。那就是:

第二种插件框架的宿主完成了太多的功能以至于其提供的基础服务使得系统的扩展能力受到限制, 比如Visual Studio提供了一个应用程序对象用来取得当前文档、活动窗体、各种界面对象等用以插件和宿主之间的交互。对于这种形式的插件框架, 由于宿主已经规定了系统的目的(比如Visual Studio就是一个IDE) 插件之间几乎不需要交互。而对于第三种框架甚至为系统提供交互界面的部分都可以作为一个插件提供。但是好像很遗憾, SharpDevelop, Eclipse都没有将界面部分作为一个插件来提供。虽然有不少其它类Eclipse插件框架(OSGI)的实现将界面部分抽离为一个插件, 但是这个插件很难被替换掉从而难以使系统轻松具有不同风格的界面。SharpDevelop的插件形式多被实现为第二种插件框架, 虽然SharpDevelop本身是全插件形式也有不少模仿SharpDevelop的其它全插件形式的实现, 但都没有考虑将界面部分作为一个插件提供。

本插件框架成功地避免了以上各种现存插件框架的不足。首先, 对宿主进行了最大程度的瘦身, 宿主仅负责插件和服务的装载和卸载以及系统启动的流程控制。第二, 为系统运行提供界面的部分也作为插件提供并且能够轻松替换。整个系统通过两个配

置文件装配起来，一个是注册到系统中的插件列表，另一个是界面说明文件。此外，本插件框架还提供了灵活的插件间交互方式。

## 为什么我会需要这样一个插件框架？

作为公司或者部门的技术主管，您肯定在不断寻求提高生产力的手段和技术。那么这就是一个不错的选择。本插件框架在快速开发WinForm应用程序和C/S系统时有非凡的表现（其实应该说是装配，因为如果使用这个插件框架相同的功能只需开发一次）。每一个有经验的软件工程师都会承认，在很多情况下我们的小组或者部门甚至我们的公司都会专注于软件的某一应用领域。此时客户的需求有很大的重叠度，为了避免重复的工作，我们会寻求各种办法来提高重用性，以期减少重复的工作增加生产力。

您肯定已经采用了各种层次的重用性手段。从代码片段到代码集合的源代码级别的重用，从可重用小构件到较大的模块的动态链接技术层次的重用。有了这样一套插件框架您就可以将重用级别再向前推进一步，面向客户需求的重用。事实上，绝大多数插件框架都是面向客户需求的重用，只不过实现程度不一样而已。只要使用的是在上一节提到的第三种插件机制，就可以重用以前开发的插件快速构建一个原型系统。然后对不满足用户需求的插件进行微调，新增的需求通过编写新的插件来满足。然而，此插件框架的优势并不止于此，其创建插件宿主程序的简易程度绝对能够让你惊愕。

首先在IDE中新建一个WinForm应用程序项目，删除IDE生成的除Program.cs之外的代码文件。然后添加对AddIn.Core.dll的引用。再修改Main方法为：

```
static void Main(string[] args)
{
    AppFrame app = new AppFrame();
    app.Run();
}
```

然后修改宿主程序的图标编译即可。

本插件框架还有另外一个显著的优势——提供系统交互界面的部分作为一个插件提供并且能够很容易地被替换掉。有些时候我们和竞标对手的软件在功能和易用性上势均力敌，如果我们的软件有一个漂亮的交互界面肯定能大大增加夺标的概率。有时候我们的用户更加崇尚简洁，有时候我们的用户不愿意为更加华丽的界面付账，此时我们可以对基本的WinForm控件加以扩展使用。本人开发了一个免费开源的[WinForm扩展控件集合](#)，它不排斥商业使用。使用此开源控件集再配合另外几个开源控件足以开发出专业程度媲美Visual Studio 2005的交互界面。

## 更进一步的信息

### 它能够运行在那些平台上？

这可能是最容易回答的问题，也可能是最不容易回答的问题。它的核心部分纯粹使用C#语言开发，可以运行于.NET 2.0及以上的平台上，也可以运行于其它实现了.NET 2.0及更高版本的操作系统上。

但是他到底能不能跨越操作系统这个平台，还需要根据具体情况来定。如果在这个系统中使用的所有插件都被实现为能够跨平台的那么装配出来的应用程序就是跨平台的。如果任何一个需要在这个系统使用的插件被实现成为平台相关的那么整个系统就是平台相关的。

我所能保证的是随插件框架发布的基础版本的UI插件是使用纯粹C#编写的，是可以跨平台的。

### 有那些成功的应用案例？

至此，您或许想知道这个插件框架在实际使用中能够有什么样的表现。所以您会问，有哪些成功的案例？通过应用案例来考技术的实用性是一个非常重要的手段。当然有基于这个插件框架的成功案例——[Mini Internet Explorer](#)，一个网页浏览器。

到目前为止，整个插件框架的开发只是我一个人的表演，所以不可能选择一个陌生的应用领域来开发应用案例。如果这样的话需求确认和系统开发都将成为一个巨大的任务，而真正需要做的就是测试插件框架在易用性、稳定性、界面分离程度、界面的专业化程度。我选择使用该插件框架开发一个网页浏览器来检测其实用性。网页浏览器是每一个使用电脑的人每一天都在用的软件以至于它的需求确认是如此的容易——我只需要问我自己需要它具有哪些功能。此外，浏览器具有丰富的图形界面。



1. 浏览器的网址输入下拉框需要为浏览提供网址，想想如果所有提供给用户的功能只能通过点击按钮来调用将会是一种什么情况。
2. 需要向浏览器的网址输入下拉框添加最近访问的网址，这代表了选项类界面元素中的选项可以在系统运行过程中发生改变。Visual Studio的查找、撤销就是实例。
3. 浏览器的网址收藏工具条和菜单需要根据收藏夹的具体内容创建，这代表了界面元素能够在启动时根据具体情况被创建，而不是界面配置文件将界面描述为什么样就必须是什么样，改变不了了。这还为插件的插件提供了可能。
4. 浏览器需要一个进度条来指示页面打开进度，需要通过前进、后退按钮的可用性来指明当前用户数据的状态、用户下一步可进行的操作。

完成了这些，首先就能够说明这套插件框架能够创建出专业化程度很高的应用系统。如果要完成这些浏览器插件和UI插件必须要有良好的交互，当然也就说明了本插件框架提供了优秀的插件间交换手段。

(如果需要从更一般的层次了解一个插件框架应该具有的功能，请阅读[关于插件框架的一般性问题](#)一节。)

## 需要付钱吗？从哪里可以获得？

不需要，它是免费的，如果您只使用我发布的基础版本。但是如果需要替换UI插件的话，您或许需要支付购买第三方界面组件的费用。如果您认为有必要让我为贵处封闭开发一套UI插件的话，您还需要支付劳务费用给我。但是我将会免费为您奉送重构了的插件框架；它的易用性、稳定性更高，UI插件的替换也更加容易。关于重构后的插件框架的结构请参考[第二部分 使用指南的第二章](#)。

## 如何获得技术支持？

如果需要让我为您开发UI插件，请发邮件到[tr0217@163.com](mailto:tr0217@163.com)。

如果您只打算使用基础版本，可以发邮件到[tr0217@163.com](mailto:tr0217@163.com)或者到[我的博客](#)上留言说明您遇到的问题。

## 授权与责任声明

### 授权

本插件的基础版本尚未采纳任何其它软件发布协议。本人在此声明：

1. 基础版本完全免费，您可以获得任何使用自由。包括应用于商业项目和对此插件框架进行逆向工程。
2. 如果您对本插件框架进行修正并且再发布，请明确指明您发布的版本与原始版本的区别。
3. 如果您免费使用本插件框架，那么您有义务为改进此插件框架做出贡献。比如通过技术支持信息报告错误、分享您的扩展功能。
4. 本人不强制要求您需要在您的软件中明显声明您使用了本产品。但是如果您由于使用本产品的免费版本带来了问题而破坏我的名誉将是绝对不允许的。
5. 本人不保证不会为本软件的后续版本采纳更为严格的授权协议，但可以保证绝对保证该软件是源代码开放的自由软件。如果本人在后续版本中采纳了其它授权协议，那么您必须按照新采纳的授权协议使用本软件的后续版本。但是您也可以按照本授权声明使用本软件的当前版本。本软件的当前版本包括
  1. AddIn.Core.dll
  2. AddIn.Gui.dll
  3. AddIn.Config.exe
  4. WeifenLuo.WinFormsUI.Docking.dll 版本2.3.1.0 至 2.5.0.0都可以兼容
  5. log4net.dll 版本号：1.2.10.0

(这里的版本指的是文件版本。通常文件版本相同的不同程序集版本间都是兼容的)

### 责任声明

由于基本版本准许免费使用，使用本插件框架带来的任何直接和间接问题本人概不负责。如果免费使用本插件框架的基础版本，本人不保证技术支持的及时性。

如果您与我签订协议为贵方重构此插件框架和编写UI插件，那么本人也只能负限于插件框架核心部分和UI插件中由于本人实现所带来的有限责任。关于技术支持和其它责任及义务将会在协议中明确说明。

此外，可以看出本插件框架的基础版本使用了WeifenLuo.WinFormsUI.Docking.dll和log4net.dll。这是两款开源软件，他们使用的协议分别为[The MIT License](#)和[Apache License Version 2.0](#)

这是两款非常友善的协议，都不排斥商业使用。

## 关于插件框架的一般性问题

### 插件框架的基本任务



设计全插件框架时有几个非常基本的问题需要考虑。

1. 插件应该怎样被装载。插件需要被主程序装载然后调用。
2. 是否需要实现界面模块和功能模块的分离，怎么实现。分离是为了让界面模块的更新不至于对功能模块的影响太大。
3. 怎么实现界面和功能的连接。很多功能都是通过界面上的操作调用的，所以连接也是个基本的问题。

关于连接，我们所考虑的当然不能只是怎么将没有参数的方法通过点击界面上的一个按钮调用起来，还需要考虑一些复杂的情形以使装配出来的应用程序符合合理的界面设计原则。

## 不得不考虑的界面设计原则


设计插件框架时同样必须考虑的界面设计原则有：

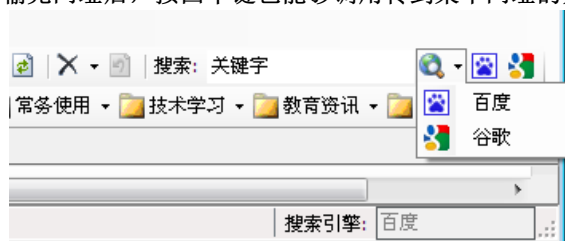
1. 如果系统需要针对登入的用户限制一些功能，最好的做法是将调用当前用户不可执行功能的界面元素的Visible属性设为false。（这并不是必须的）
2. 界面元素的Checked与Enabled的状态以及其它属性需要能够表示出用户当前的操作、当前用户数据所处的状态、用户下一步可进行的操作。




在上面的网页浏览器中，组合框中显示了当前页面的网址。这就是界面元素的属性表示了用户当前的操作——用户正在浏览网址为<http://tr0217.blog.163.com/>的网页。状态栏中的进度条已经消失了，说明当前页面下载完成了：这就是界面元素表示了当前用户数据状态的例子。浏览栏、状态栏、收藏栏都是可见的，同时相应的菜单项的Checked属性都为true：这也可以认为是界面元素表示了当前用户数据的状态。在上图中在“在当前页面中打开新网址”的界面元素处于可用状态，这是由于当前有页面打开：这就是界面元素表示了用户下一步可进行的操作。

3. 界面不应该单调乏味，如界面上只有按钮和菜单。单调乏味的界面不仅影响美观，还会影响易用性。在这个浏览器的界面上如果只有按钮和菜单的话，输入网址应该怎么办呢？弹出一个对话框吗？这复杂了用户的操作，显然不符合界面设计的原则。

如果提供丰富的界面，就会带来另外一些困难。某个界面元素调用的方法所需的参数需要另外一个界面元素提供。如上图所示，界面元素调用的转到某个网址的方法的参数就需要它左边的组合框来提供。当然更多的情况是参数由调用者界面元素本身提供的，在组合框中输入完网址后，按回车键也能够调用转到某个网址的方法。



还有些情形下需要提供不变的参数。如上图设置当前搜索引擎的两个菜单项，他们的参数就是固定的，从上到下分别是：“<http://www.baidu.com/s?wd=,百度>”、“<http://www.google.com.hk/search?q=,谷歌>”。比如使用百度搜索“Plugin”这个关键字时的调用URL为“<http://www.baidu.com/s?wd=Plugin>”。“,”之后的文本则是这个搜索引擎的公认名称。

“<http://www.baidu.com/s?wd=,百度>”其实表示了两个参数，“,”是一个分隔符。设置当前搜索引擎的方法的原型为Void SetSearchEngine(System.String, System.String)，第一个参数表示搜索引擎的基本URL，第二参数表示搜索引擎的公认名称。这个参数用于更新状态栏中指示当前搜索引擎的文本框中显示的文字，如右图。（使用逗号作为参数分割符只是本插件框架的约定。）

## 使用任何一款插件框架都需要注意的问题

为什么我们选择使用插件框架？

第一回答就是我们需要更大程度地重用。是的，使用插件框架能够将重用提升到满足客户需求的程度。我们编写的的一个又一个插件的目标在于满足客户的需求。可重用程度提高了生产力也就提高了。

插件框架带个我们的第二个好处就是可以灵活地定制、自由地扩展。我们可以删除以前开发的系统中不符合用户需求的插件，可以编写新的插件来满足用户不一样的需求。甚至用户也可以遵照插件框架的约定来对系统进行扩展。

明白我的暗示了吗？“遵照插件框架的约定”这就是限制。插件框架在带来这么多好处的同时也带来了限制。所以使用插件框架时我们需要冷静，插件框架并不能取代其它重用手段。我们不能因为有了插件框架就终止了我们在其它重用手段上的努力。有了插件框架我们仍然需要其它层次的可重用性——代码片段、源码库，小二进制组件、功能完善的组件。

这不是在哗众取宠。我已经见过了错误地使用插件框架的活生生的例子。由于他们在其它重用手段上停止了继续努力导致生产力不增反降。也或许他们从来没有评估过自己的生产力。

还有另外一个很重要的问题需要注意。插件框架的目标在于什么？通过重用以前开发的插件快速生产出满足客户需求的应用系统。所以注册到插件框架中的几乎所有插件都在于满足客户一个或者多个功能需求。所以在设计插件时就需要注意，插件的接口必须从满足用户的需求上进行考虑。其次才是提供有限的和其它插件交互的接口以满足交互和扩展需求。

通过上面的分析，我们得出了使用任何一款插件框架都需要注意的两个问题：

1. 绝对不能放弃在其它复用手段上的努力。
2. 设计插件时必须从满足用户需求上进行考虑。

## 插件框架还需要完成那些任务

如果之前提到的三个基本任务都被完成了，那么插件框架已经具有很高的实用价值了。装配出的系统界面友好，能够很容易地更换界面组件使得系统的界面绚丽美观。此时还有其他的问题需要考虑。

每一款软件都不能说100%没有问题，当系统被部署后我们需要日志系统来**诊断和定位问题**。日志信息需要分不同的级别，可以输出到不同的目标（文件、控制台、数据库等）。

此外这是一个插件结构的系统，所有人都可以开发插件注册到其中。我们无法保证每一个插件都能够正确的被加载运行，所以必须要保证任何一个**插件在加载和运行时出现的错误不会影响到系统的稳定性**。

有时在一个系统中将会有一些功能很少被使用，但是客户又有这样的需求。在这些插件加载后，它提供的功能却很少被使用，让这些插件在系统一启动就加载是很低效的做法。如果直到系统退出它的功能仍然没有被使用，那它占据的内存就白费了，加载它所消耗的时间也白费了。解决这个问题的办法就是**延迟加载**，意思是不在系统启动时加载它，当这个插件提供的功能第一次被使用时加载。

还有其它两个功能可以被提供。**动态加载（热加载）**——在系统运行中向系统中增加新的插件。**动态卸载（热卸载）**——在系统运行中卸载某个插件而不是等到系统退出时卸载。如果在系统运行中能够确定某个插件将不再使用就可以将其动态卸载。

---

# 第二部分 使用指南

---

## TR0217插件框架入门

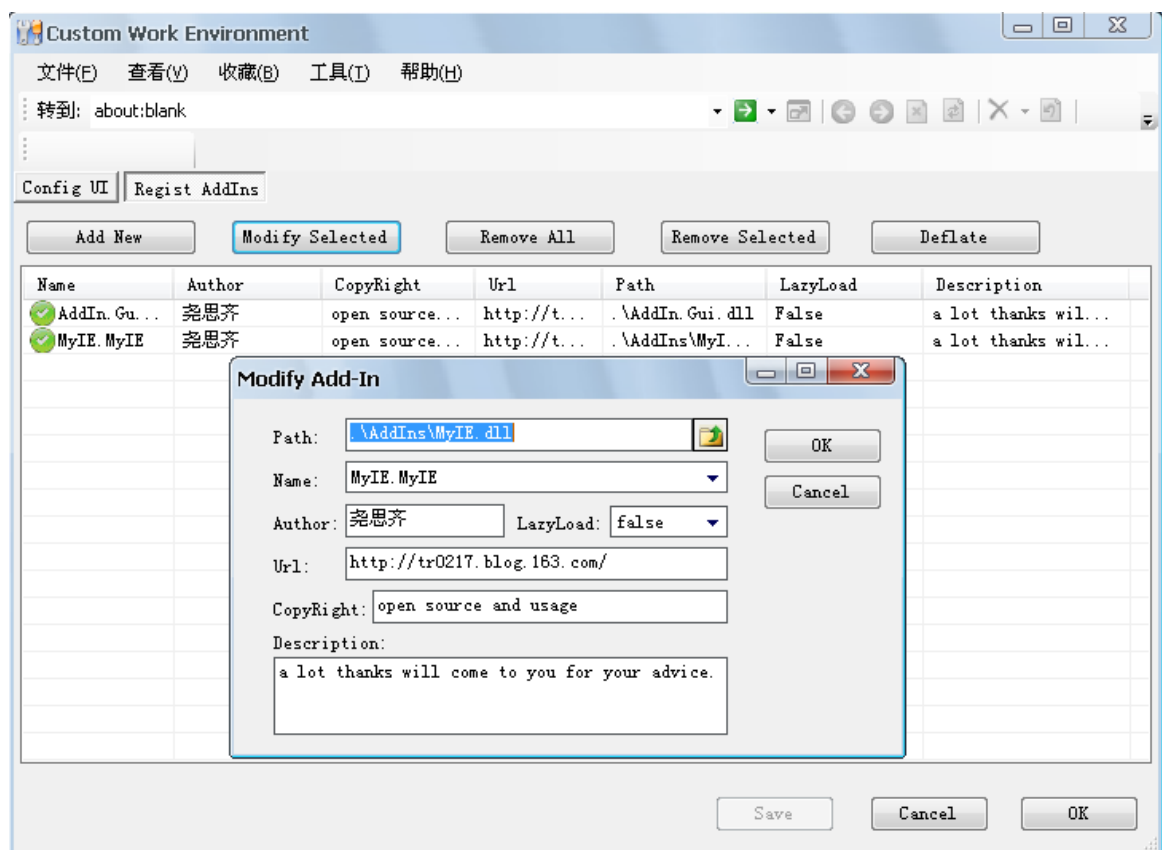
### TR0217插件框架简介

TR0217插件框架是运行.NET 2.0及更高版本上的WinForm及C/S架构的应用程序开发框架。首要目的在于让您能够更加简单地实现应用程序的模块化，并且将各个模块间的耦合性降到最低。这样一来应用程序的开发就像搭积木一样，将已开发的模块组合到一起产生一个应用系统。使用该框架开发的应用程序的定制能力也能像积木一样，可以轻松的去掉某些模块和增加模块。

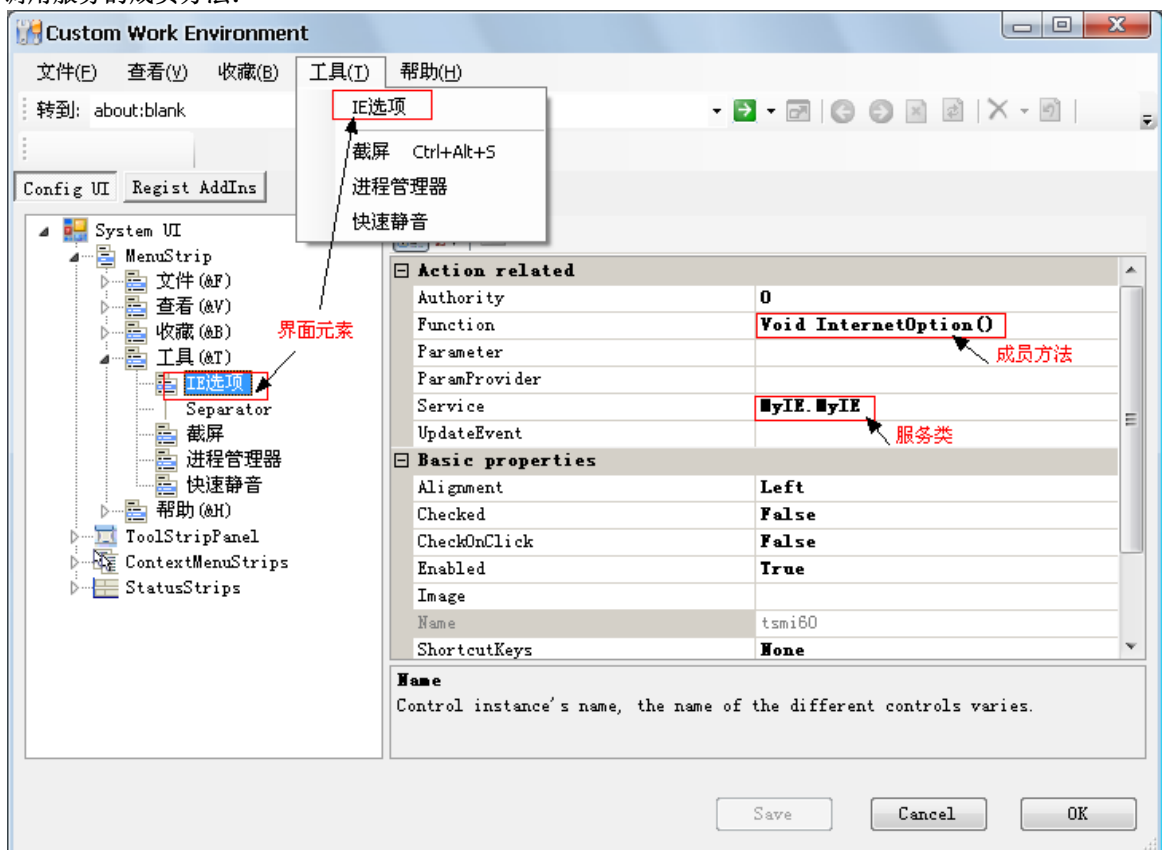
当模块组合到一起时需要一个交互界面来为用户提供功能。产生交互界面的功能也被此插件框架提供了，同样采用模块化的方式，使产生交互界面的模块能够很容易的被替换掉让系统拥有完全不同外观的交互界面。

至此您或许仍然难以形象直观的把握这个插件框架。其实从表面上看这个插件框架其实是这样的。首先创建一个类库项目，在这个类库中至少包含一个外部可见的继承自AddIn.Core.ServiceBase的类（称其为服务），类的成员方法就是需要提供给用户的功能，当然首先需要添加对AddIn.Core.dll的引用。编译为dll文件，再使用插件注册工具将这个dll到系统中，然后在注册工具中创建界面元素（如按钮、菜单等）用来调用服务的成员方法。

注册到系统中的插件：



创建界面元素调用服务的成员方法：



## 从Hello, world! 开始

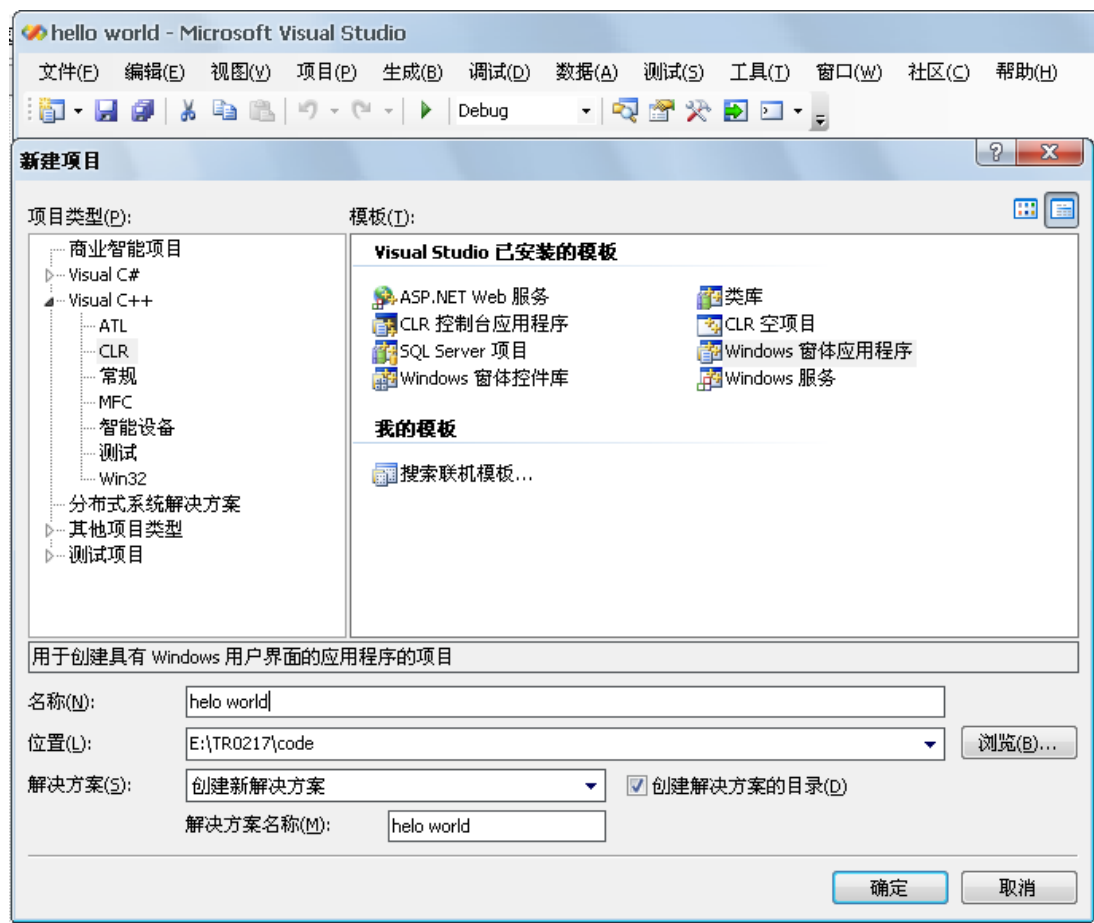
从Hello, world! 开始是学习软件开发技术的一个不朽传统。没必要标新立异，在这里一节里，我们将一步一步地使用此插件框架开发一个应用程序——在窗口中显示一句“Hello World!”。

### 建立宿主程序

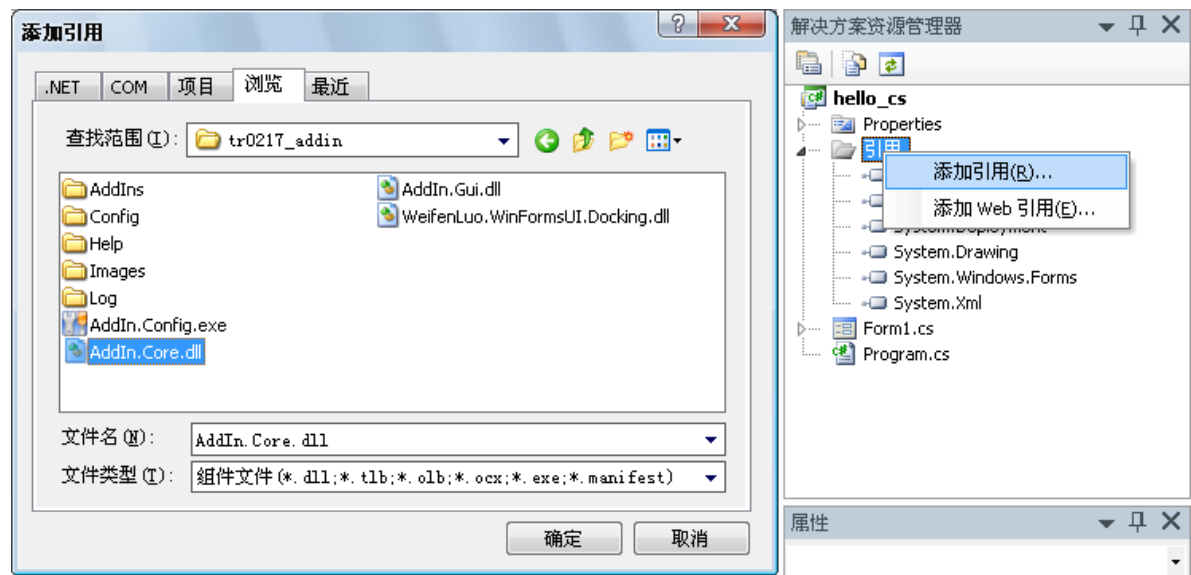
首先启动Visual Studio，新建一个.NET项目，选择Windows窗体应用程序。这一步对于C#、VB、J#语言的使用者来说选择在左边的树形结构中选择Windows，然后在右侧选择Windows应用程序。C++语言的使用者需要选择CLR然后选择Windows窗体应用



程序。



接下来，添加对 **AddIn.Core.dll** 的引用。使用 C#、VB、J# 语言的使用者可以在相应项目的解决方案管理器中的引用上右击选择添加引用，从弹出的对话框中选择浏览标签，然后导航到 **AddIn.Core.dll**，点击确定即可。



C++ 的使用者可以在解决方案的根节点上右击选择“引用(E)...”。在弹出的对话框里点击“添加新引用(N)...”，接下来的操作与其它语言使用者相同。

接下来，删除IDE生成Form1。不光是从解决方案中移出，文件也可以一并删除。

对于C#语言的使用者来说，将Programcs文件中的内容改为：

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using AddIn.Core;

namespace hello_cs
{
    static class Program
    {
```

```

    /// <summary>
    /// 应用程序的主入口点。
    /// </summary>
    [STAThread]
    static void Main(string[] args)
    {
        AppFrame app = new AppFrame();
        app.Run();
    }
}

```

C++语言的使用者需要将源文件中以项目名称命名的文件内容改为：

```

#include "stdafx.h"

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace AddIn::Core;

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
    AppFrame app;
    app.Run();
    return 0;
}

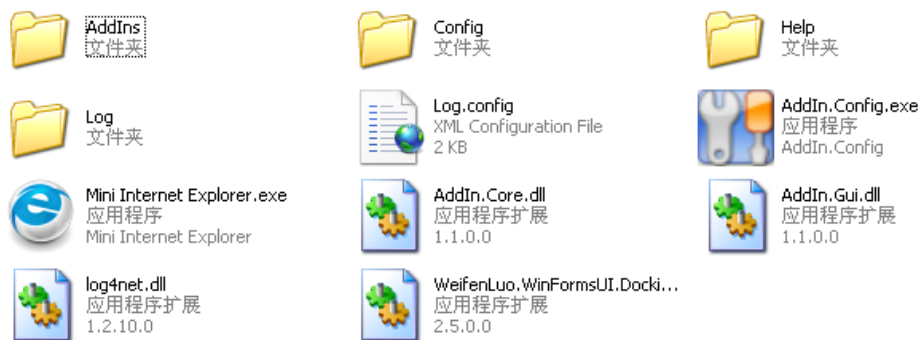
```

接下生成解决方案，可以直接将项目配置改为Release。

*（我们玩个魔术吧！将生成的exe文件拷贝到[Mini Internet Explorer](#)的执行文件目录下，然后双击它。有什么反应？它变成了一个Mini Internet Explorer！）*

## 编写一个插件，在对话框中显示Hello World!

在编写第一个插件前，您需要了解一下本插件框架的目录结构要求。

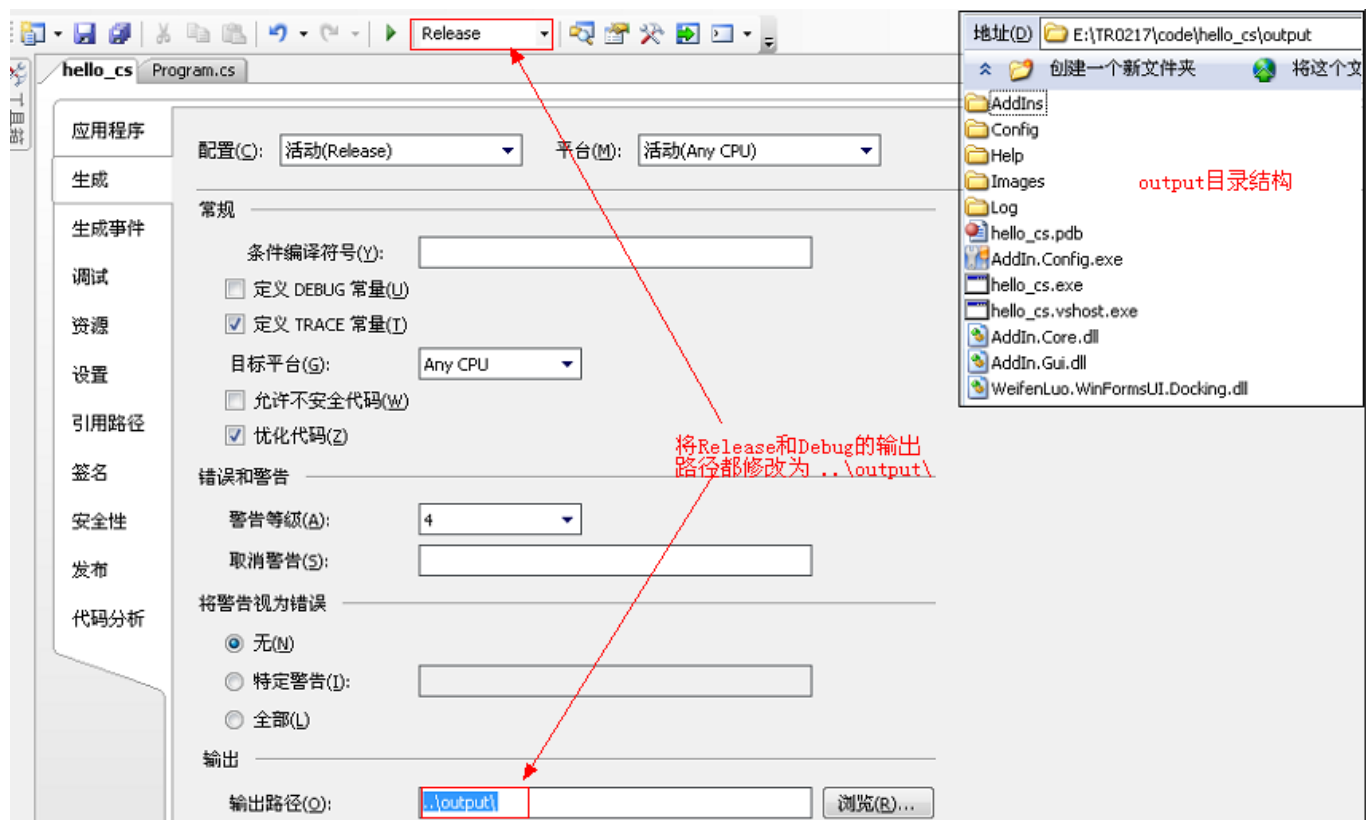


- AddIns文件夹用于放置注册到系统里的插件。并不强制这么做，您也可将自己编写的插件放置到其它地方；但是如果没有特殊的原因还是推荐将其放到AddIns中。如果插件过多可以再建子目录，将每个插件和相关的文件放置到各自的文件夹中。

*（这么做还有另外一个问题需要考虑，您编写的插件所依赖的DLL文件对于系统来说会是不可见的从而导致插件加载失败。解决方案将在本部分的下一章中给出。）*

- Config文件夹，用于放置系统的插件列表文件和界面说明文件。**这是强制的。**
- Help文件夹，用于放置系统的帮助文档和其它说明文档。推荐这么做。
- Images文件夹，用于放置系统界面所需的图片资源，推荐采用png格式的图片。推荐非强制。
- Log文件，用于放置系统运行中的错误日志。日志的存放位置可以通过配置文件指定，所以这也是推荐非强制的。
- 插件框架的核心文件AddIn.Core.dll、AddIn.Gui.dll、AddIn.Config.exe、宿主程序（Mini Internet Explorer.exe）以及AddIn.Gui.dll所依赖的第三方界面组件推荐放置到应用程序的根目录下。

接下来按照这个结构创建好应用程序的输出目录，也可以直接将发布版解压出来的文件夹直接拷贝到解决方案目录。然后修改项目的生成目录。



上图中的“..\output”表示一个项对路径。关于项对路径的知识请自行到网络上搜索。至此，我们已经准备好编写第一个插件了。

第一步，新建一个类库项目。在解决方案的根节点上右击，在弹出菜单里选择“添加(D)—新建项目(N)...”。修改新建项目的名称为hello，项目位置采用默认值；点击确定。然后删除hello中的类class1，样需要添加对AddIn.Core.dll的引用，修改生成目录为“..\output\AddIns\”。第二步，在新建的项目中添加一个类hello。类名可以自由指定。然后打开代码文件，添加对名称空间AddIn.Core的引用，让hello继承自AddIn.Core.ServiceBase。再为类增加一个成员方法SayHello()。最终的代码如下：

```
using System;
using System.Collections.Generic;
using System.Text;

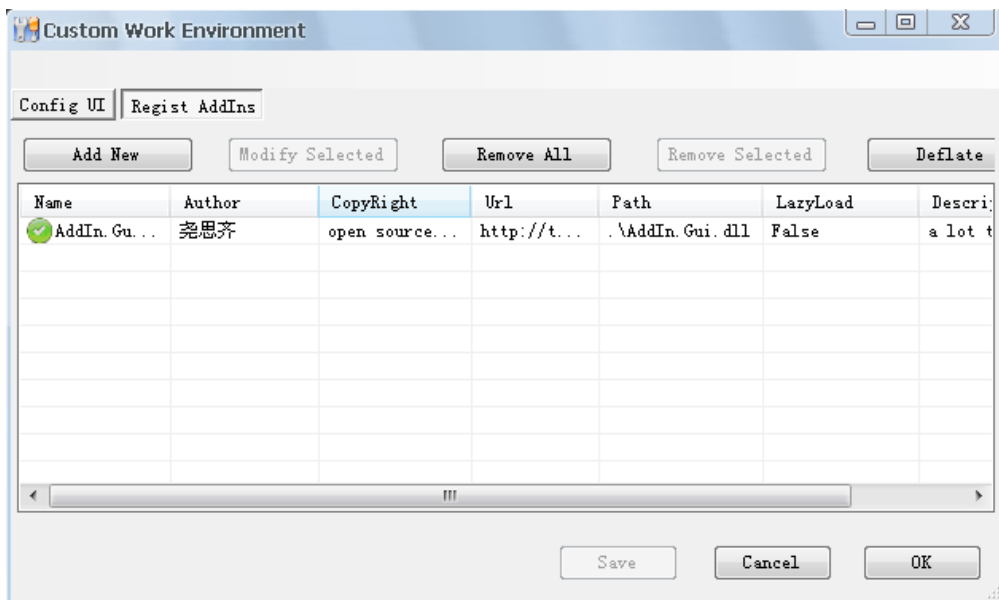
using AddIn.Core;
using System.Windows.Forms;

namespace hello
{
    public class Hello : ServiceBase
    {
        public void SayHello()
        {
            MessageBox.Show("Hello, world!");
        }
    }
}
```

然后编译。至此，一个简单的插件就算完成了。

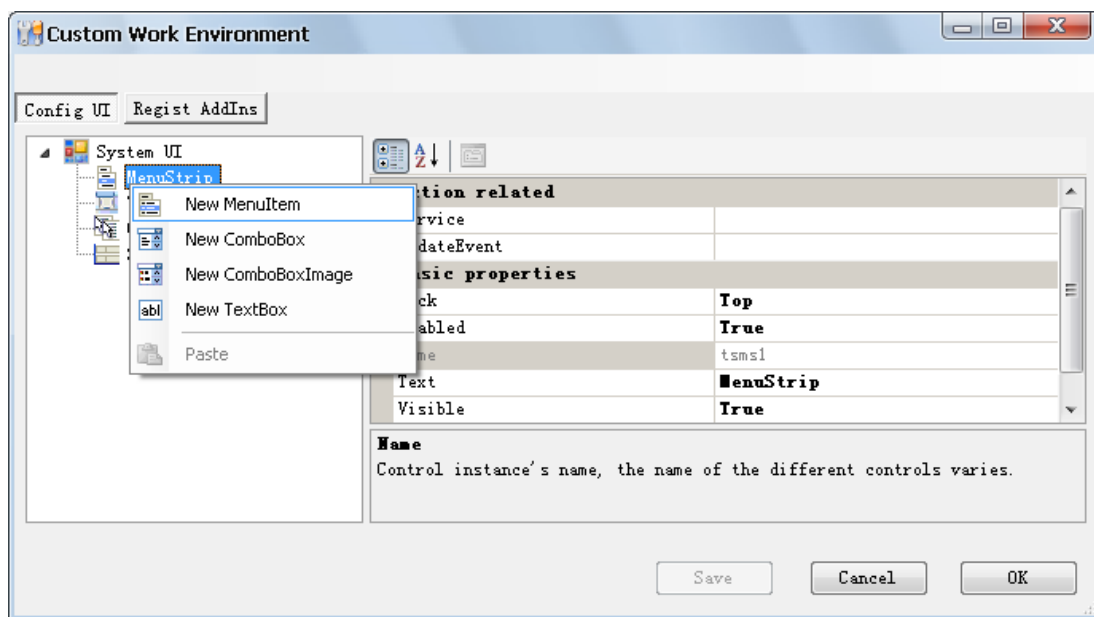
### 将插件注册到系统中

启动AddIn.Config.exe，切换到Regist AddIns页。（如下图所示）

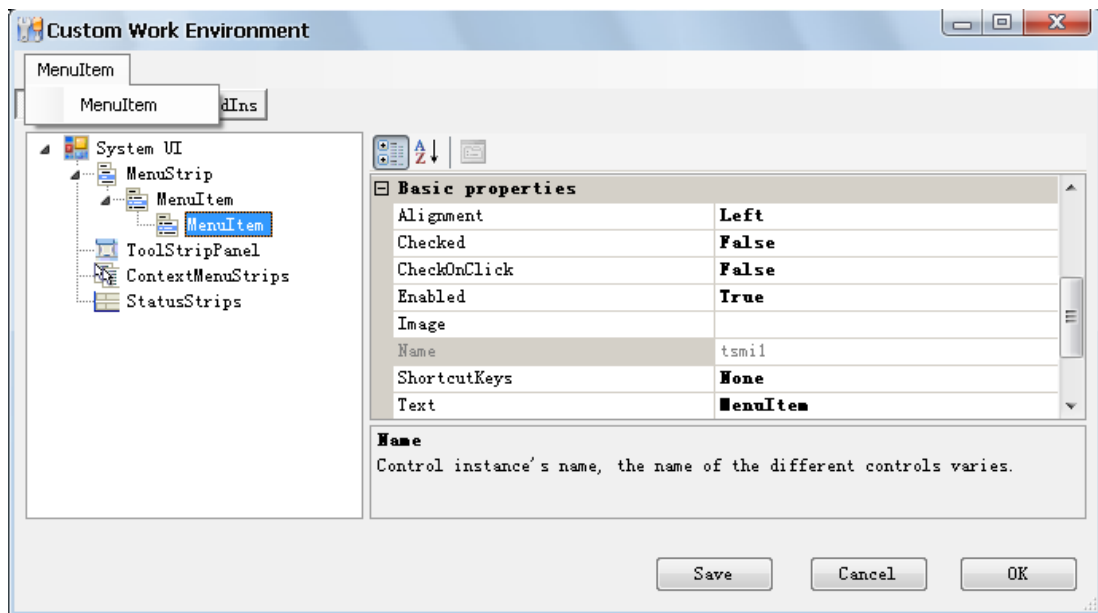


可以看到，AddIn.Gui.dll已经注册到其中了。如果AddIn.Gui.dll没有注册的话配置工具是无法启动的，因为AddIn.Config.exe也是一个插件宿主，系统需要AddIn.Gui.dll提供界面才能正常启动。

点击Add New，弹出注册插件对话框。如下图所示，点击Path文本框右侧的小按钮打开插件。程序会自动将Name文本框填为插件Dll中第一个找到的继承自AddIn.Core.ServiceBase的类名。如果插件加载失败会弹出提示对话框。LazyLoad保留false即可。其它几项根据实际情况进行填写，也可以留空。点击确定即可完成注册。然后切换到Config UI页，配置用于调用SayHello()的界面元素。

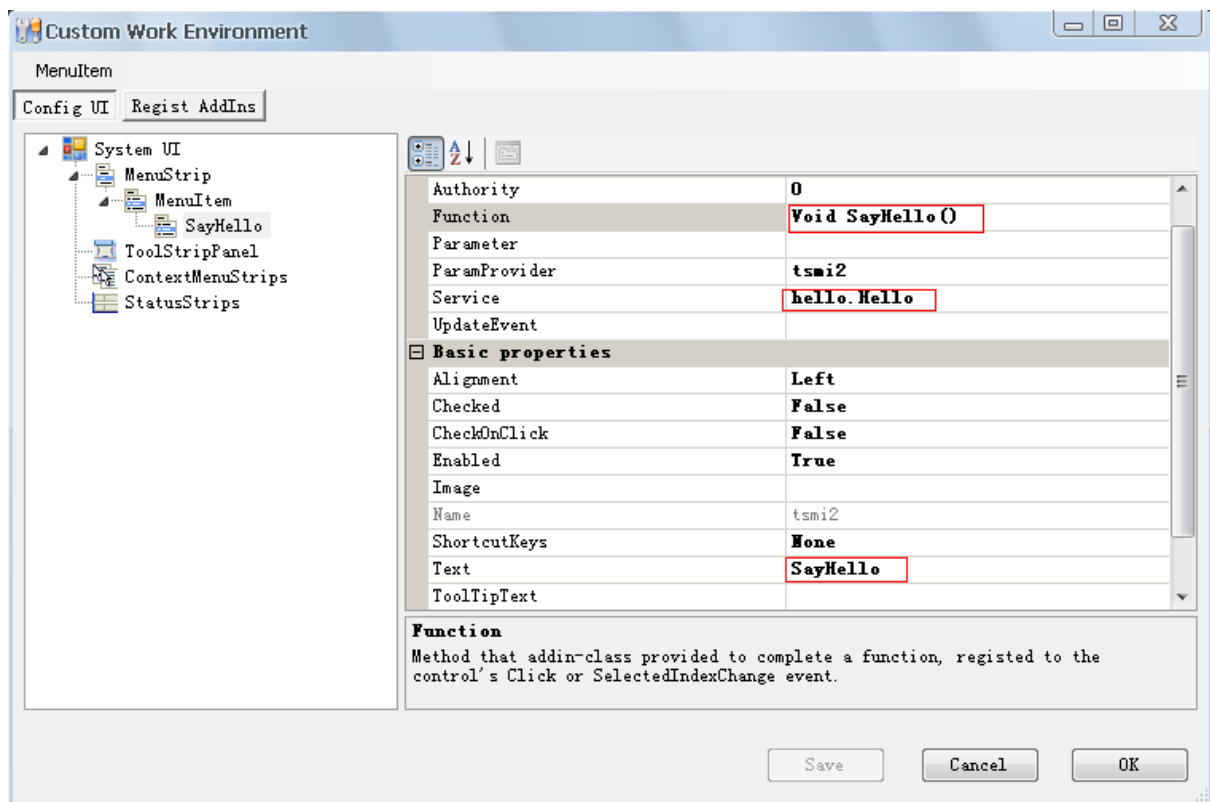


在MenuStrip节点上右击，从弹出菜单上选择“New MenuItem”。然后展开MenuStrip节点，在刚才新建的MenuItem上右击，同样选择“New MenuItem”。配置工具能实时反映出界面配置的改变，如下图，配置工具的菜单栏已经有了刚才新建的两个菜单项。



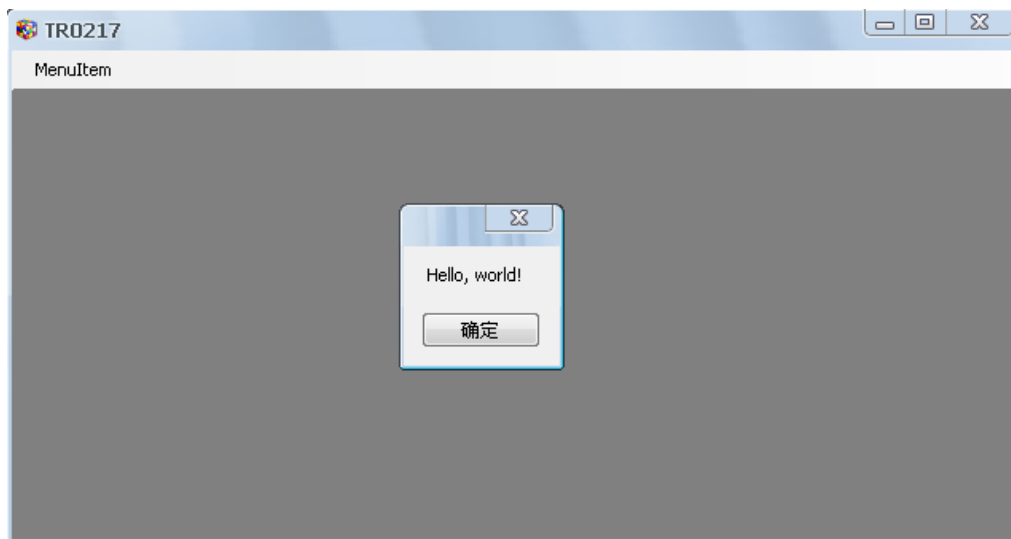
选中层次最深的MenuItem，修改其显示文字并将其与SayHello()方法连接起来。可以看到右侧的属性面板和Visual Studio的属性面板差不多，操作方式也是一样的。修改Text属性为SayHello，即可将显示文本修改为SayHello。

然后修改Service属性，这个属性表示为用户的提供功能的方法所属的服务类。从下列列表中选择hello.Hello。接下来修改Function属性，这个属性表示为用户提供功能的方法。从下拉列表中选择Void SayHello()。点击Save，退出程序；也可以以直接点击OK。



至此，这个插件已经成功注册到系统中了。然后启动hello\_cs.exe。点击SayHello菜单，其运行效果如下图所示。





您也可以进行一些探索性学习。比如修改其它节点的属性看看对应用程序有何影响。

## 显示在停靠窗口中的Hello, world!

这一节里我们一起实现一个稍微复杂的*hello world*插件——将hello, world显示到停靠窗口里。

打开上一节建立的hello\_cs解决方案，添加一个名称为helloDock的类库项目。删除项目中的Class1.cs，添加对AddIn.Core.dll和WeifenLuo.WinFormsUI.Docking.dll的引用。在项目中添加一个窗体，然后修改HelloForm.cs，让它其继承自AddIn.Core.DocFormBase。表示从形式上约定HelloForm是一个文档窗口，关于文档窗口、工具窗口将在*TR0217插件框架应用进阶*一章讲述。从工具箱中脱出一个Label放到窗体上，在属性面板里修改显示文体为“Hello, world!”。

接下来需要新建一个服务类，将其命名为HelloDock，用于将这个可停靠窗口停靠到主窗体中。为HelloDock也添加一个名称为SayHello的方法，不过需要不同的实现。为了将这个可停靠窗体停靠到主窗体上必须和AddIn.Gui交互。

在AddIn.Core.dll中定义了界面服务的接口IUIService。为HelloDock增加一个IUIService成员用户保留获取的界面服务，这可以避免每次需要界面服务时都需要重新获取。这里有个问题不得不注意。假设传入的参数是没有问题的，如果取得的服务为null，说明了什么？是没有向系统注册相应插件，还注册了还没有加载，又或者加载失败。需要实现一种机制让这个问题的回答更加明确——那就是保证只有在加载失败或者没有向系统注册的情况下才会取得null值。所以需要在所有的插件都加载完成后再获取需要与之交互的服务。

AppFrame类通过静态事件FinishLoadAddIn向外发布所有不需要延迟加载的插件都加载已加载完成。在插件服务类的构造函数中订阅这个事件，能够保证一定能够订阅到这个事件。

获取界面插件的服务也很简单。首先调用AppFrame的静态方法GetServiceCollection()取得IServiceCollection。然后调用其成员方法GetService<T>()取得界面服务，传入的泛型参数就是需要获得的服务的接口。关键代码片断：

```
private IUIService _uiService;

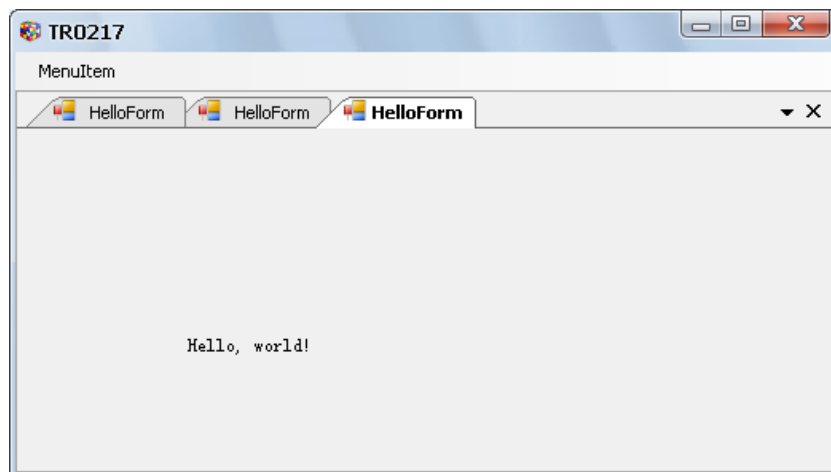
public HelloDock()
{
    AppFrame.FinishLoadAddIn += new LoadAddInHandler(AppFrame_FinishLoadAddIn);
}

void AppFrame_FinishLoadAddIn(LoadAddInEventArgs e)
{
    _uiService = AppFrame.GetServiceCollection().GetService<IUIService>();
}
```

接下来的事情就非常简单了，调用\_uiService的方法将新建的HelloForm停靠到主窗体上。代码如下：

```
public void SayHello()
{
    HelloForm frm = new HelloForm();
    _uiService.ShowDocForm(frm);
}
```

使用注册工具，添加一个MenuItem，将HelloDock注册到系统中。其运行效果如下图所示：



## 事实上，我想说Hi, beauty!

或许比起“Hello, world!”您更想说“Hi, beauty!”。这一节我们编写一个插件用于在文档窗口中显示“Hi, beauty!”。如果仍然上一节一样，您肯定会有点不耐烦的。这个插件的目标是与上一节编写的插件交互，将文档窗体中的文字修改为“Hi, beauty!”。

实现上一节的插件（HelloDock）时没有做任何需要和外部交互相关的考虑，即使在同一个程序集的另一个对象里也很难修改窗口中显示的文字。所以首先修改HelloDock，为其提供用于交互的接口。

```
public interface IHelloDock
{
    void SayHello(string str);
}
```

为了实现这个接口，还得为HelloForm增加一个Property用于修改label的显示文本。

```
public string Hello
{
    get { return label1.Text; }
    set { label1.Text = value; }
}
```

将IHelloDock实现为：

```
public void SayHello(string str)
{
    HelloForm frm = new HelloForm();
    frm.Hello = str;
    _uiService.ShowDocForm(frm);
}
```

接下来编写一个新插件HiDock，用于使用参数“Hi, beauty!”调用HelloDock提供的方法void SayHello(string str)来在停靠窗体中显示“Hi, beauty!”。为了同HelloDock交互，HiDock中必须保有指向HelloDock的引用，并且能够通过这个引用调用void SayHello(string str)方法。但是插件的目标就在于降低耦合性，使插件之间能够不直接引用而交互。此时就要借助.NET的基础特性来为我们提供的便利。可以通过一个对象的类型说明类——Type——来调用其成员方法。为了达到这个目的我们需要两个对象：

```
Type _helloType; //HelloDock的类型说明
ServiceBase _helloService; //指向HelloDock实例的引用
```

获取HelloDock服务的机制同在HelloDock中获取UI服务的机制一样，不过其方法应该实现为：

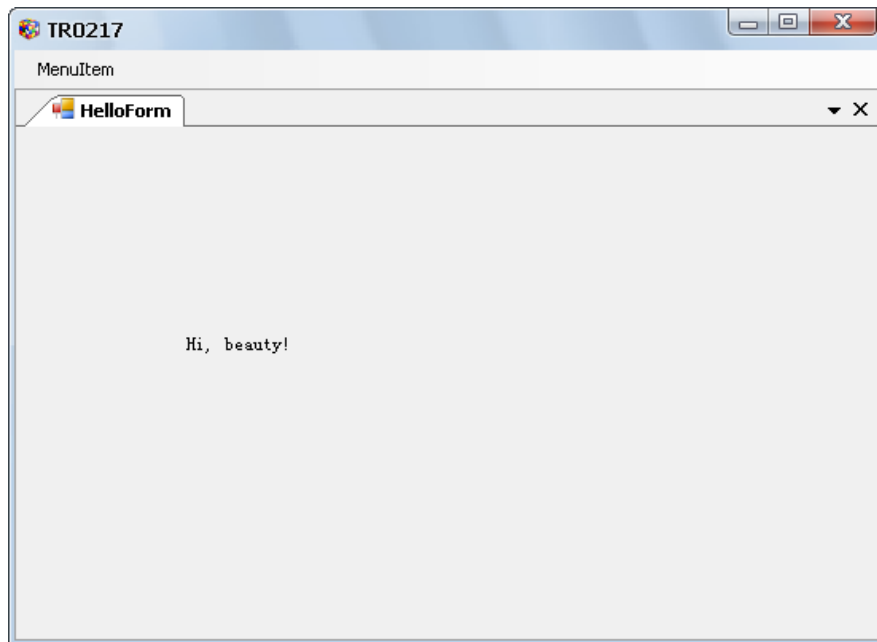
```
void AppFrame_FinishLoadAddIn(LoadAddInEventArgs e)
{
    try
    {
        //根据名称获取服务对象
        _helloService = e.ServiceCollection.GetService("helloDock.HelloDock");
        _helloType = _helloService.GetType();
    }
    catch { }
}
```

编写一个成员方法用于完成这一节的目标。

```
public void SayHi()
{
}
```

```
//通过实例的类型和实例调用成员，即调用SayHello方法，传入的参数为"Hi, beauty!"
if (_helloType != null)
    _helloType.InvokeMember("SayHello", System.Reflection.BindingFlags.InvokeMethod,
        null, _helloService, new object[] { "Hi, beauty!" });
else MessageBox.Show("未能获取helloDock.HelloDock服务，调用目标失败！");
}
```

将这个插件注册到系统中，运行效果如下图所示。



## 我要问候任何一个想打招呼的对象

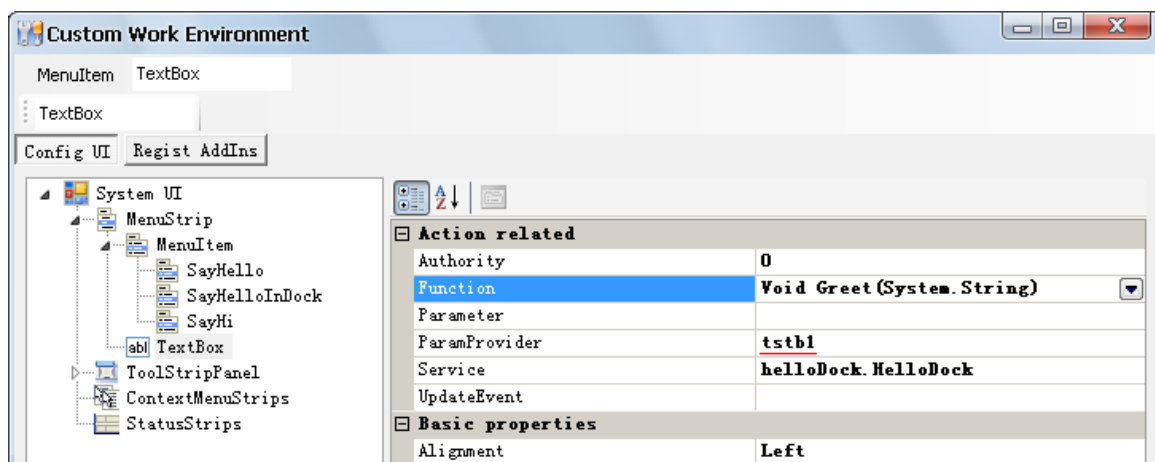
我要问候任何一个想打招呼的对象。这说明我们需要一个将某个对象作为参数的方法，这个方法能够产生问候语问候这个对象。喂，你有对象了吗？开个玩笑！就软件开发人员的理解，对象是一种客观存在，包括有形的实体（如人、猫、狗）、无形的概念或思想（如时间、马克思主义）。

我们通过为HelloDock插件增加一个方法来完成这个目标，这样可以节省一些劳力。很简单直接将代码贴出来。

```
private string[] greetings = new string[] { "Hello, ", "Hi, ", "Hey, " };
Random random = new Random(3);

public void Greet(string str)
{
    int i = random.Next(3);
    HelloForm frm = new HelloForm();
    frm.Hello = greetings[i] + str;
    _uiService.ShowDocForm(frm);
}
```

重新生成插件，使用配制工具将这个方法和一个能够提供输入的界面元素连接起来。此处的关键点是ParamProvider的值，它是这个界面元素的名称。



## 插件间的两种交互方式

在前面几个插件的实现中已经使用了两种插件间交互的方式。在HelloDock的实现中我们使用了基于接口的插件间交互的方式，获取界面服务（UIService）和调用界面服务都是通过接口进行的。

```
_uiService = AppFrame.GetServiceCollection().GetService<IUIService>();

_uiService.ShowDocForm(frm);
```

在HiDock的实现中我们使用的是基于元数据和反射的方式，获取HelloDock和调用其SayHello方法都是用过类型名称进行的。

```
//根据名称获取服务对象
_helloService = e.ServiceCollection.GetService("helloDock.HelloDock");
_helloType = _helloService.GetType();

//通过实例的类型和实例调用成员，即调用SayHello方法，传入的参数为"Hi, beauty!"
if (_helloType != null)
    _helloType.InvokeMember("SayHello", System.Reflection.BindingFlags.InvokeMethod,
        null, _helloService, new object[] { "Hi, beauty!" });
```

这两种交互方式有各自的优势和缺陷，因而也有各自适用的情形。

基于接口的交互方式有严格的限制，通过接口只能调用接口的成员方法；但是获取和调用服务的过程更加自然。基于反射和元数据的交互方式没有接口的限制，只要知道了名称和参数形式就可以获取和调用服务；但是获取和调用服务的过程不够自然，同时由于没有约束会导致接口稳定性问题。

基于接口的交互方式适用于构建一个软件系统的基本功能插件。基本功能插件对外提供的接口必须有很高的稳定性；并且基本功能插件之间、基本功能插件和其它插件之间需要大量的交互；基于接口的交互方式恰好能够满足这个需求。此时可以将各个功能插件的接口放入同一个单独的程序集中，所有基本功能模块和需要和基本功能模块交互的插件都引用此程序集。这样可以避免插件之间直接引用导致的高耦合性。对HelloDock插件来说就是新建一个类库项目，其中只包含接口IHelloDock。然后HelloDock引用这个类库使类型IHelloDock对其可见。

由于本插件框架不依赖接口，所以对于一些非通用插件或者一些满足用户特殊需求的插件可以不为其设计接口。基于元数据和反射的交互方式作为对基于接口的交互方式的一种补充，适用于需要和这些没有为其设计接口的插件交互或者需要调用插件接口之外的功能的情形。

## 内容回顾

本章从Hello, world!开始循序渐进的讲解了如何编写插件。涉及到的主题有插件创建、注册插件、配置界面以调用服务类提供的方法、插件间的交互。

- 插件创建，这是最简单的一个主题。一个插件就是包含至少一个访问修饰为public的服务类的类库。新建一个类库项目，添加对AddIn.Core.dll的引用，增加一个公有的继承自AddIn.Core.ServiceBase的类，编译通过，一个插件就创建好了。
- 注册插件。运行AddIn.Config.exe，在Regist AddIns页将插件文件成功添加到下方的列表中即可完成注册。成功添加的标志是列表前方的图片为🟢。具体操作请参考[将插件注册到系统中](#)一节。
- 配置界面以调用服务类提供的方法，为无参数的方法配置界面非常简单，为有参数的方法配置界面还必须给出参数或（和）参数由哪个界面元素提供。其实，本章并没有包含配置界面的所有内容。如需要，还应指明维护界面元素逻辑的事件；关于这个内容请参考下一章[怎么维护界面逻辑](#)一节。
- 插件间的交互，有两种方式，依赖接口的和不依赖接口的。依赖接口的方式用于构建一个软件系统的基本功能插件，更深入的内容请参考[高级主题的如何建立完备的服务集合](#)一节。不依赖接口的方式作为依赖接口的方式的补充手段。

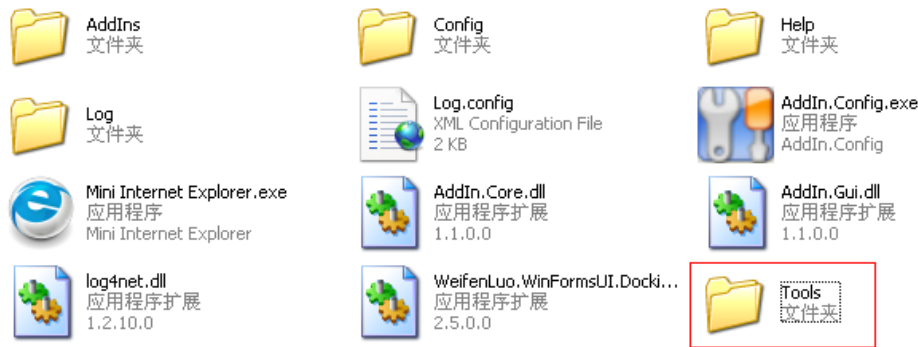
本章还有另外一个内容，插件框架的目录结构，参考[从Hello, world! 开始](#)一节。更深入的内容请查看下一章[第一节规划应用程序的目录结构](#)。

## TR0217插件框架应用进阶

### 规划应用程序的目录结构

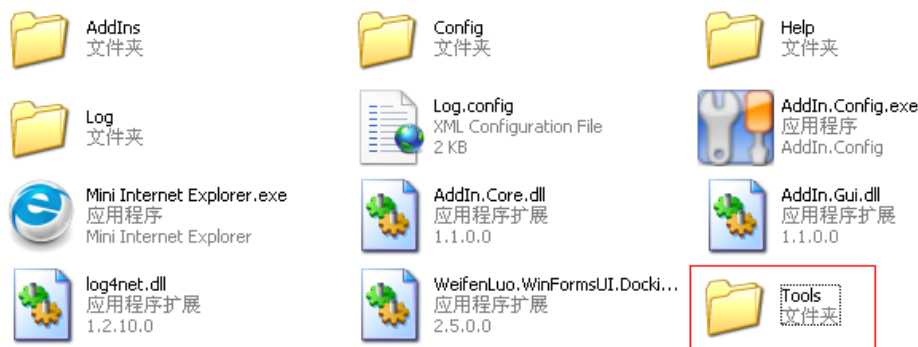
在上一章的第二节[从Hello, world! 开始](#)中已经对此插件框架的目录结构进行了介绍。所以这一节直接进入目录规划的主题。

插件框架依赖的几个基本文件（AddIn.Core.dll、AddIn.Gui.dll、log4net.dll、WeifenLuo.WinFormsUI.Docking.dll、AddIn.Config.exe）的存放位置最好保持不变，放在应用程序的根目录下，即宿主目录下。此时和Mini Internet Explorer.exe文件放在同一个目录下。



上图与上一章的第二节中的图片相比多了一个Tools文件夹。这里边放的是一些可以单独运行的小工具。这些工具是由AddIn.Gui.UiService的Void Exexute(System.String)进行调用的。这个方法的目的是执行某个路径下的应用程序。

每一个插件单独放到AddIns目录下的一个文件夹里。



一个插件需要的固定的图片资源可以放到其下的Images文件夹里。如果有可以加载到这个插件中运行的插件再新建一个文件存放之；如MyIE Plugin，其下存放的是可以自当前打开的页面中执行以完成某些特殊功能的Javascript。调用插件的插件的界面元素需要的图片资源直接存放到子插件的目录下。如调用解除右键菜单限制的MenuItem左侧显示的图片就存放在插件MouseUnlock目录下。



你或许会说一些插件并不是如此简单，只有一个dll文件。它很可能还需要引用其它dll文件。事实确实是这样，那么我们该如何处理呢。.NET程序启动时会首先到GAC里寻找需要的引用，然后是当前运行程序的目录，最后会查找App.Config文件中配制的目录。所以如果某些程序集只会被某个插件所引用，那么将这些程序集放到插件目录中，然后在配制文件夹中添加这个目录为引用目录。引用目录的配制节的形式为：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath=""/>
    </assemblyBinding>
  </runtime>
</configuration>
```

使用时只需要修改privatePath的值，如果有多个目录使用半角英文分号分隔。如果浏览器插件还需要引用一些只会被它使用程序集，那么把这些程序集放到MyIE目录下。然后将AddIns\MyIE增加为引用目录，即 privatePath="AddIns\MyIE"。

插件运行中使用的不确定的资源，如浏览器的网站图标（FavoritesIcon），就单独存放到插件目录下的一个文件夹中。

系统主要功能的帮助文档放到Help文件夹下。某个插件的帮助文档推荐放到插件的目录下，然后再重写插件服务的About方法时，给出交互界面将帮助文档显示出来。

## 如何使用日志系统

本框架的日志系统采用的是开源日志系统——log4net。在此采用了将日志对象在配制文件中进行说明，然后在类中通过log4net.LogManager.GetLogger方法获取这个日志对象。如，在Log Config中使用下面一节配制了一个日志对象，名称为"AddIn.Core"。

```
<logger name="AddIn.Core">
```



```
<level value="DEBUG"/>
<appender-ref ref="CoreLogFileAppender" />
</logger>
```

在AppFrame类中使用GetLogger方法通过日志名称获取了这个日志对象。

```
public static log4net.ILog FrameLogger = log4net.LogManager.GetLogger("AddIn.Core");
```

为了让您能够更好的使用日志系统，在此，对log4net进行一个简要的说明，更细致的消息请参考log4net的帮助。在下面列出几个log4net的汉语参考：

- [Log4Net使用指南](#)
- [日志系统Log4net的学习笔记系列](#)
- [Log4net源码分析](#)

在log4net中所有与日志有关的对象都配制在配制文件的log4net节。首先需要在configSections节对log4net节进行必要的说明。

```
<configSections>
  <section name="log4net"
    type="log4net.Config.Log4NetConfigurationSectionHandler,
    log4net-net-1.0"
  />
</configSections>
```

然后在log4net配制各种日志对象。

```
<log4net>

  <root>
    <level value="DEBUG" />
    <appender-ref ref="ConsoleAppender" />
  </root>

  <appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender" >
    <layout type="log4net.Layout.PatternLayout">
      <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] &lt;%X{auth}&gt;%n - %m%n" />
    </layout>
  </appender>

  <logger name="AddIn.Core">
    <level value="WARN"/>
    <appender-ref ref="CoreLogFileAppender" />
  </logger>

  <appender name="CoreLogFileAppender" type="log4net.Appender.FileAppender" >
    <param name="File" value="Log\\Core.log" />
    <param name="AppendToFile" value="false" />
    <layout type="log4net.Layout.PatternLayout">
      <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] &lt;%X{auth}&gt;%n - %m%n" />
    </layout>
  </appender>

</log4net>
```

log4net中的日志对象有：

- logger，实现了ILog接口。通过调用logger的不同方法来记录不同级别的日志。logger拥有一个记录级别，如果需要记录的日志的级别低于这个logger的级别这个日志就不会被记录。如名称为“AddIn.Core”的日志对象的级别为“WARN”，则调用Info和Debug方法就不能将日志信息输出到目标。

级别	方法	是否有效	说明
OFF			不记录任何日志
FATAL	void Fatal(...);	bool IsFatalEnabled;	严重错误
ERROR	void Error(...);	bool IsErrorEnabled;	错误
WARN	void Warn(...);	bool IsWarnEnabled;	警告
INFO	void Info(...);	bool IsInfoEnabled;	消息
DEBUG	void Debug(...);	bool IsDebugEnabled;	调试消息
ALL			所有消息

(日志级别从上到下递减。)

在Log.Config中可以看到root节的配制内容和logger节的内容很相似。其实root节也配制了个日志对象。由于在实际应用中，许多logger可能具有相同的特点，便将这些相同的特点配制进root节中，在logger节中就不用再配制了。在上面的配制文件中root拥有一个名称为“ConsoleAppender”的日志对象，则所有的日志都会拥有这么一个appender对象。root的日志级别为DEBUG，如果不现实指名logger的级别，则默认为DEBUG级别。

- appender，定义了存储日志的目标和方式。一个logger对象可以包括多个appender对象。一个appender对象也可以被多个logger对象引用。logger实际上是通过appender对象将日志信息记录到目的地的。

appender对象可以通过单独的一节配制，如上面的配制文件中的“ConsoleAppender”。在日志节中通过appender-ref属性指名对其引用，如<appender-ref ref="ConsoleAppender" />。

- filter，有些时候需要对日志的过滤条件并不是低于某个级别的日志就不记录。所以需要为appender对象指定一个filter对象来对日志进行过滤。在log4net.Filter的名字空间下已经有几个预定义的过滤器，完全满足日常需要，具体配制方法参考官方的使用手册。
- layout，日志需要按一定的格式输出，所以appender对象还拥有一个layout对象。

打开Mini Internet Explorer的Log.Config文件会发现其中多了两部分内容。

```
<logger name="MyIE.MyIEService">
  <level value="ERROR"/>
  <appender-ref ref="MyIELogFileAppender" />
</logger>

<appender name="MyIELogFileAppender" type="log4net.Appender.FileAppender" >
  <param name="File" value="Log\MyIE.log" />
  <param name="AppendToFile" value="true" />
  <layout type="log4net.Layout.PatternLayout">
    <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] &lt;%X{auth}&gt;%n - %m%n" />
  </layout>
</appender>
```

在一个新的logger节定义了一个名称为MyIE.MyIEService的logger。这个logger对象使用了一个名称为“MyIELogFileAppender”的appender对象，其配制信息紧随其后。type属性说明这是一个文件类型的appender，将会把日志输出到文件。第二行使用value字段指明File的位置为“Log\MyIE.log”。第三行指明日志的输出类型为追加到末尾，如果将value的值修改为false，系统每次启动都会清除上一次运行记录的日志。

(以后的章节中将会以Mini Internet Explorer为例，从实际应用方面对此插件框架的使用进行讲解)

## 设计插件的接口

这个插件框架的目标是快速创建可定制的应用系统，注册到系统中的插件的目标在于满足用户的需求，而不是提供开发应用系统的基础功能组件。所以从上至下的插件开发过程是合适的，即首先需要提供给用户的功能和用户的交互方式确定下来。

基础功能是什么呢？它是不需要用户直接调用从而应该对用户隐藏的功能。比如，为了开发多语言版本的系统，通常需要文本字典功能，用于将用户关键字映射为各种语言的版本，提供字典功能的组件就不应该设计为插件。这样做的好处有很多。

- 首先，可复用性提高了；这种组件非常有可能用到不使用此框架开发的系统。
- 第二，减少用户破坏系统配置的可能性；用这个插件框架开发的系统是允许用户定制的，如果注册到系统中的与用户需求无关的插件越多，用户错误地订制系统的几率就越大。

所以设计在本框架中使用的插件的第一指导原则就是尽量不考虑将基础功能设计为插件（其实使用Sharpdevelop之类的插件框架也一样）。当然在设计插件接口时也不要加入一些提供基础功能的接口。唯一的特例就是，这些基础功能不可能在别的地方重用，而是为需要注册到这个系统中的全部或者绝大多数插件提供基础服务时，才能将这些功能设计为基础插件。提供届面服务的AddIn.Gui.dll插件便是这种情形。

第二个指导原则，相关性的一系列功能应该作为同一个类的成员方法，用于更新调用这些功能的界面元素状态的事件（将在下一章详细讲解）也应该包含在这个类中。插件的目标在于满足用户需求，所以在设计插件的接口时应该从用户的需求入手。设计时不仅要考虑到需要向用户提供的功能，还要考虑到功能之间的相关性和流程性。相关性和流程性就是通过更新界面元素状态体现出来的。

下面以这个浏览器为例进行说明。

首先需要分析用户对于浏览器的功能需求。当然，浏览器的第一功能就是浏览某个网址。接下来就是与浏览网页有关的一系列功能：当用户向前浏览后可以后退，后退之后可以前进；有些时候需要刷新和停止某个页面；有时候需要在页面中查找某个关键字。

由此便可以设计出浏览器最基本的接口。接下来要考虑这些功能之间的关联性——完成某个功能的先决条件和功能完成后的后续结果。只有向前浏览之后才可以后退——向前浏览便是后退的先决条件，可以后退便是向前浏览的后续结果。这种相关性便

需要用事件来完成，这些事件最终会被插件框架订阅用来更新调用这些功能的界面元素的状态。

考虑到这个一个标签式多文档界面的插件框架，当然也会实现为一个标签式的多页浏览器。还需要一些与页面有关的基本功能。如，关闭当前页面，关闭所有页面，关闭所有非当前页面，恢复最近关闭的页面等。

综合所有常用的需求，以及我们对浏览器一些其它需求，比如解除右键菜单，清除页面上的飞行广告，让页面变成某种适合阅读的颜色，将页面内容保存为图片，截取页面上某部分内容为图片等。我们设计出了Mini Internet Explorer中的MyIE插件的接口。

到此为止仍然让我们疑惑的或许就是下面这些事件了。这就是用来维护界面逻辑的事件，将在下一章详细讲解。

```
//用于页面关闭后维护界面逻辑的事件
event AddIn.Core.UpdateUiElemHandler UpdateClose;
//用于页面下载完成后更新界面逻辑
event AddIn.Core.UpdateUiElemHandler UpdateComplete;
//用于确认是否可以进行后退操作，以更新完成后退功能的界面元素的状态
event AddIn.Core.UpdateUiElemHandler UpdateGoBack;
//用于确认是否可以进行前进操作，以更新完成前进功能的界面元素的状态
event AddIn.Core.UpdateUiElemHandler UpdateGoForward;
.....
```

## 怎么维护界面逻辑

上一节设计出的插件的接口中除了UpdateUiElemHandler，其它的都好理解。虽然前文已经指明插件框架靠订阅这些事件来更新界面元素的状态，但是插件怎么得知需要更新界面元素至什么状态。

界面元素的状态在第一章的第三节已经有所提及。它包括界面元素的Enabled、Checked、Visible属性，界面上显示的文字，选项界面元素（如：ListBox、ComboBox）的选择项或者选择索引，能提供数值的界面元素（如：ProgressBar）的数值等。这些都是可以用来指示系统当前所处的状态和用户下一步可进行的操作。

UpdateUiElemHandler的原型如下：

```
public delegate void UpdateUiElemHandler(object sender, UpdateUiElemEventArgs e);

public class UpdateUiElemEventArgs : EventArgs
{
    private bool _checked;
    private bool _enabled;
    private bool _visible;
    private int _count;
    private int _maximum;
    private string _text;
    private object _value;
}
```

值得注意的是UpdateUiElemHandler的最后一个参数——UpdateUiElemEventArgs的实例。正是这个实例将界面元素需处于的状态传递给插件框架的。插件框架对这个类的各个成员有标准的理解，所以在编写方法时如果需要更新界面状态就要按插件框架的理解为它的实例赋值。

- Checked——用于设置界面元素十分应该处于选中状态。
- Enabled——用于设置界面元素的可用性。
- Visible——用于设置界面元素的可见性。
- Count——用于更新界面元素的数值属性。
- Maximum——用于确认界面元素数值属性的最值。
- Text——用于更新显示在界面上的文本。
- Value——用于更新各种界面元素的各种其它值。也用来设置Combox或者其它选项类控件的选定值，或者选项类控件中添加值。

在这个浏览器中将前进和后退的按钮的UpdateEvent分别设置为UpdateGoForward和UpdateGoBack，初始状态都设置为false。当在当前页面中向前浏览后，需要发布UpdateGoBack事件将参数的Enabled字段设置为true；向后浏览后需要进行类似的操作。当切换页面后需要发布UpdateGoForward和UpdateGoBack事件，用于让启用和停用前进和后退按钮以指示在当前页面中是否可进行前进和后退操作。当然在发布这些事件时Checked应为false，Visible应该为true。

在状态栏中的一个进度条的UpdateEvent设置UpdateProgress，用于指示当前页面的打开进度。当当前页面的下载进度发生变化时发布UpdateProgress事件，将参数的Count设置为当前的进度值，Maximum设置为下载完成时的进度值，Visible、Enabled、Checked保留默认值，分别为true，true，false。框架在设置ProgressBar的进度值使用的计算公式为Count\*ProgressBarMaximum/Maximum。

状态栏中还有一个文本框，其Enabled的属性始终都是false。其显示出来的文字用于指示当前使用的默认搜索引擎。它的更新事件为UpdateSearchEngine。当默认搜索引擎发生变化时，发布此事件，将Text字段设置为当前默认搜索引擎的名称，Enabled字段设置为false。

对于输入网址的组合框，不仅要更新其显示的文字，有时还要想其下拉列表中添加历史访问记录。在此种情况下本框架约定：

如果参数的Text成员不等于null或Empty，首先将选项类控件的显示文本设置为Text，并且约定不引发SelectedIndexChanged事件。对于ComboBox来说，如果Text不等于null就将ComboBox的显示文本设置为Text。当Value成员也不为null时，就将Value对象插入到选项类控件成员容器的第一个位置。对于ComboBox来说，就是将Value插入到Items的索引为0的位置。此时会检查选项类控件的子项目是否多于Maximum个，如果多于Maximum就从最后移出一个，对于ComboBox，就是移出Items的最后一项。

否则，首先判断Count是否是一个合适的SelectedIndex。如果是将选项类控件的选择索引设置为Count，此时引发SelectedIndexChanged事件。如果不是一个合适的SelectedIndex则测试Value是否是null，如果不为null，就将选项类控件的选择项设置为value。如果替换了本框架的UI插件，且本UI组件的选项类控件的选择项对象比较大，请不要使用Value更新选择项。

## 文档窗体、工具窗体和对话框

所有功能强大的应用程序中都有三种窗体——对话框、文档窗体、工具窗体。

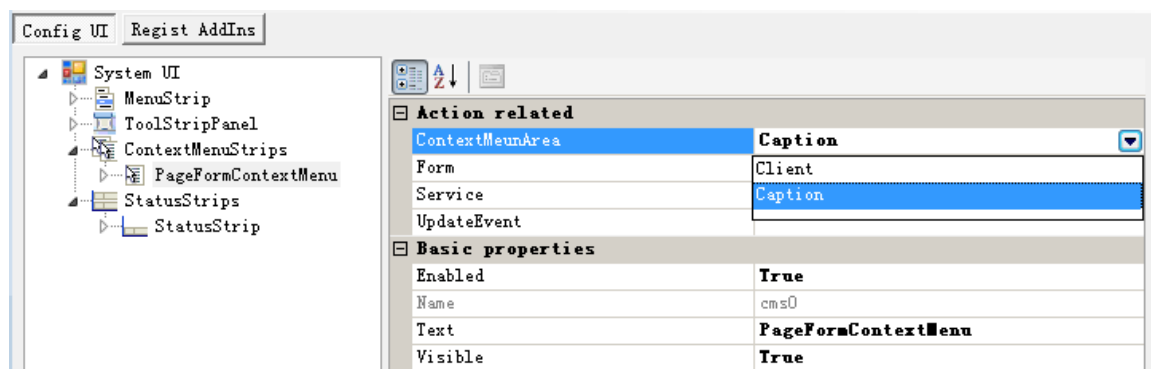
对话框是一种非常简单的窗体，都以模态弹出，即对话框弹出时在当前应用程序中只能在对话框上进行操作。对话框的作用是为了让用户做出简单的选择或者提供一些必要的输入。比如，在用户退出程序时如果没有保存的改变，应该弹出对话框询问用户是否保存。打印文档时弹出对话框让用户输入打印的页码范围、份数，选择打印机等。在.NET环境中，已经提供多种进行基本操作的对话框，这些对话框运行时会和系统的风格保持一致。

文档窗体是用来表现用户数据的，它最明显的特征是可以拥有多个实例，单文档应用系统除外。大多数情况下，文档窗体需要以一定的布局显示到应用程序主窗体内部。对于插件框架来说，就是文档窗体的显示需要框架的管理。所以插件提供了DocFormBase这样一个文档窗口基类。使用此插件框架时所有文档窗体都需要继承自此类。此类不仅提供用于管理文档窗口布局的基本功能，还自动获取配置给某类文档窗口的弹出菜单。对于某个窗口一般需要两种文档菜单，一个在右击标题栏时弹出，用于控制整个文档，如关闭、保存等；另一个在右击文档内容时弹出用于对文档内容进行编辑。DocFormBase提供了两个property用于获取配置到窗口标题栏和客户区的右键菜单。

```
public ContextMenuStrip ContextMenuStripCaption
{
    get { return _contextMenuStripCaption; }
}

public ContextMenuStrip ContextMenuStripClient
{
    get { return _contextMenuStripClient; }
}
```

所以将右键菜单指定给某个窗体时一定要指明所属位置。如下图，这个右键菜单指定给页面文档的标题栏。



工具窗口用来控制用户数据或者提供其它与操作文档不冲突的功能，即不需要以模态形式显示，比如Visual Studio中的工具栏和解决方案管理器。工具窗口的特性是只有一个实例或者至少让用户感觉只有一个实例。肯定不能让用户点击一次“显示工具栏”菜单就新显示一个工具栏。单是为了内存使用效率，也最好让工具窗口只有一个实例。工具窗口一般有两种类型，第一种就像Visual Studio的工具栏，需要停靠到主窗体中。这中工具窗口需要插件框架来管理其显示。另一种像对话框一样是弹出窗口，但是不是模态的，这种工具窗口不需要插件管理器显示。在本框架中只提供了第一种工具窗口的基类ToolWinBase。它同样可以自动获得配置给本类工具窗口的弹出菜单，当然也提供了同样的用于获得弹出菜单的property。

## 创建文档模型实现文档窗口

建立好需要处理的文档的模型，然后新建一个窗体或者类让其继承自DocFormBase。在文档窗体中实现一些方法用于将文档对象显示出来，并且提供一些浏览功能。这样一个文档窗口就算完工了。

事实上，现在绝大多数组件产品对于文档对象都提供了对应的控件来显示浏览甚至是编辑，就像WebBrowser控件一样。仅仅将他们拖放到新建的文档窗口上，Dock属性设为Fill。运行时，将文档对象和控件关联起来，控件立刻就能显示出文档对象，并能提供浏览、编辑等功能。

对于那些自定义的文档对象，也希望你能编写这样的控件。这样做能够实现更高级的重用性，对程序的模块划分也更加明确，维护起来也更加容易。

创建好文档窗口后仍然有些问题需要考虑。界面元素需要指示的指示当前文档内容的状态。当切换文档窗口后，需要获得新激活的窗口关联的文档的内容的状态。所以在多文档系统中每一个文档窗口还要记录文档内容的状态，如果文档对象没有记录的话。

首先需要分析的是有那些状态是需要记录。对网页来说，页面的下载进度，是否可以前进或者后退，当前的地址就是必须记录的状态。对于其它文档来说还需要记录文档内容与打开时相比是否改变了，是否有操作可以撤销，是否有操作可以重做，剪贴板上是否有内容可粘贴，是否需要指示文档中选中的项的格式等信息，需要在状态栏显示的文本以及一些其它的自定义信息等。

在Mini Internet Explorer中将一个扩展后的WebBrowser拖放到继承自DocFormBase的窗体（PageForm）内就完成了页面窗口。由于WebBrowser的Progress和Complete存在不一致性，所以要在PageForm中实现了两个Property用于确认当前框架是否下载完成，以及当前框架的下载进度，用于分别更新表示进度的ProgressBar的可见性和进度值。

```
public bool Complete
{
    get { return _complete; }
}

public int Progress
{
    get { return _progress; }
}
```

此外，还须要将WebBrowser的一些表示页面状态改变的事件发布出来。有两种方式可以选择，第一种直接通过属性取得WebBrowser，然后直接订阅WebBrowser的事件；另一种就是将在PageForm中订阅WebBrowser的事件再重新发布。推荐以第一种方式进行，如果要对状态进行预处理则采用第二种方式。

## 何时发布更新界面事件

通常插件向用户提供的某些功能完成后，必然会导致文档中的数据发生改变。所以首要的发布更新界面事件的时机就是在完成用户功能的方法退出时。

当然有些时候并非只有显式的通过界面元素实现的功能才会导致文档数据的改变。用来显示、浏览、编辑用户数据的控件可能实现了一些右键菜单功能或者快捷键，这些操作同样会导致文档数据的改变。对于优秀的此类控件都会在文档数据的不同改变时发布相应事件。所以在打开或者新建一个新文档的方法完成时，应该订阅这些文档的数据改变事件以便在数据改变时发布更新界面事件。

这便引发了一个问题，手动调用提供给用户的功能后通常也会导致控件发布数据改变事件。如果一次数据改变发布两次界面更新事件是很低效的做法。使用本插件框架时应该详细思考并作出协调。其实并非只有使用本插件框架时需要注意这个问题，采用普通方式编写程序时也需要注意这个问题。

由于界面元素始终要表示的是当前文档的状态，所以在文档窗口激活或者失活时需要发布更新界面的所有事件。如果系统支持一种文档类型，则可以只在文档窗口激活或失活时发布事件。如果系统支持多种类型的文档就需要在激活和失活时都发布这些事件。这是由于，如果有多种文档类型，当新激活的文档类型和原来的活动文档类型不同时，需要改变的界面元素很可能和激活同类型文档时不同。而表示原来文档状态的界面元素应该处于不可用状态。

推荐的做法是在激活和失活时都发布更新界面事件。插件框架是灵活的，非常有可能加入另外一个插件就会引入一种新的文档类型。

这些问题其实在使用普通方式编写WinForm应用程序时也应该注意。在下一节会涉及到另外一个主题，多线程与界面元素。其中也有部分关于发布更新界面事件的注意事项。

## 和AddIn.Gui交互，创建收藏菜单和收藏工具条

系统的运行界面并非被界面说明文件完全限制给此插件框架带来了更大的灵活性。对这个浏览器来说，就是可以根据实际情况创建收藏菜单和收藏工具条。

首先需要在界面上配置好收藏菜单和收藏工具条。在IE插件中获取UI服务之后立即通过唯一名称或者路径获取对它们的引用。

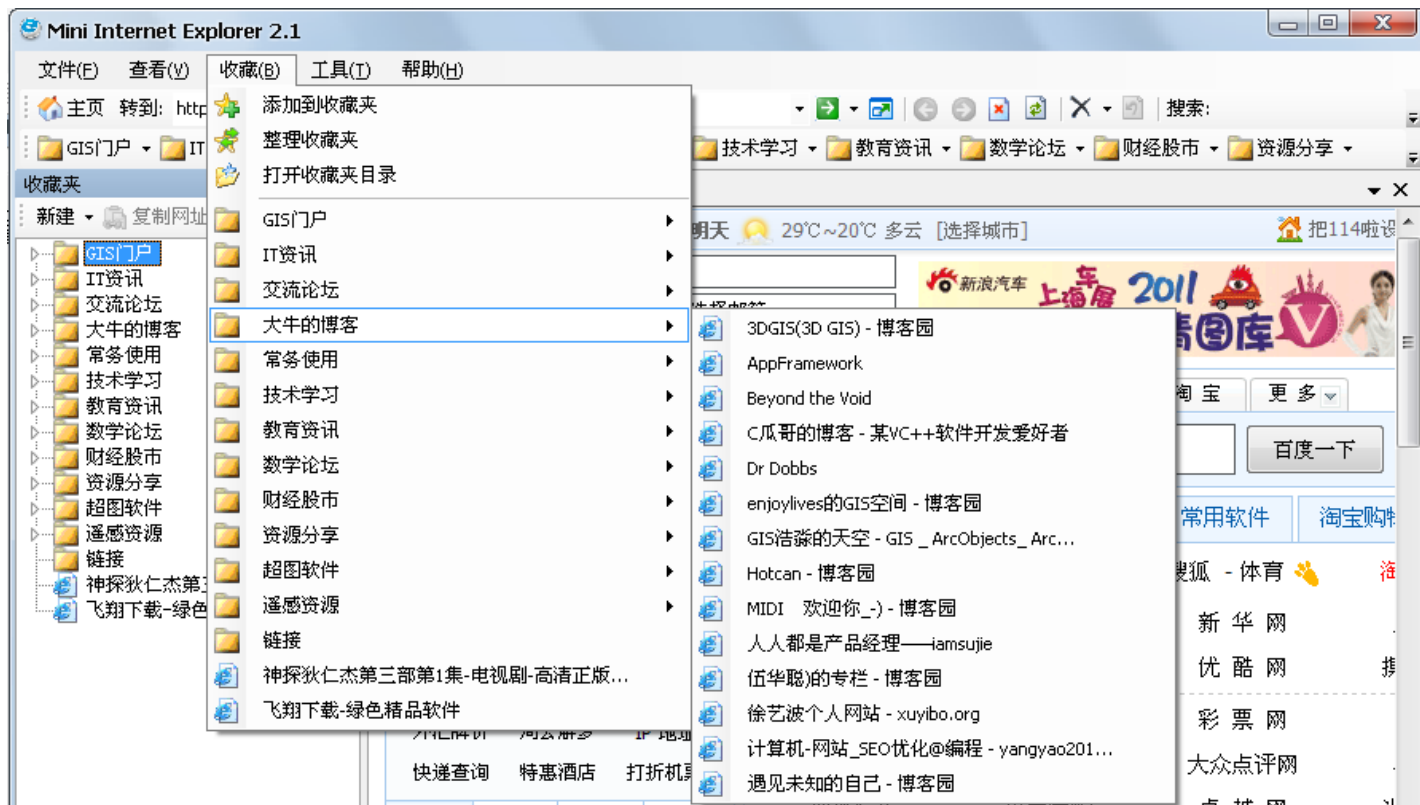
```
_services = AppFrame.GetServiceCollection();
_uiService = (IUiService)_services.GetService("AddIn.Gui.UiService");
_uiService.MainForm.WindowState = FormWindowState.Maximized;

_favoritesMenu = _uiService.GetToolStripItem("MenuStrip/收藏(&B)/") as ToolStripMenuItem;
_favoritesMenu.DropDownItems.Add(new ToolStripSeparator());
_tsmi = _favoritesMenu;
_favoritesStrip = _uiService.GetToolStrip("ts1", true);
```

由于同时需要对收藏夹的结构创建收藏菜单、收藏工具条、收藏夹工具栏中的树形结构。所以有必要新建一个类FavoritesAgent用于对收藏夹结构进行代理和管理。为了减少对收藏夹树的遍历次数同时为了降低耦合性，让FavoritesAgent在初始化收藏夹树的时候发布事件，然后按照需要订阅这个事件创建收藏菜单、收藏工具条。收藏夹菜单和收藏夹工具条的树形结构中子节点都需要添加到父节点的节点容器中，如下图，代表各个大牛博客网址的MenuItem需要添加到



代表代表“大牛的博客”的文件夹的MenuItem的DropDownItems中。



所以子节点在创建之前父节点必须已经被创建了，也就是FavoritesAgent中加载父节点加载事件必须在子节点加载之前；因而FavoritesAgent中加载收藏夹树时应该采用先根遍历的方式。请注意下面的代码，文件夹的处理放在链接文件之前。由于收藏夹中文件夹和链接文件的层次不同在收藏工具条中要使用不同的界面元素代表，所以发布事件时还需将\_level信息一并发送出去。

```
int _level = 0;
private void ProcessFavoritesDir(FavoritesDir favoritesDir)
{
    _level++;

    foreach (string dir in Directory.GetDirectories(favoritesDir.Path))
    {
        FavoritesDir fDir = new FavoritesDir();
        fDir.Path = dir;
        favoritesDir.FavoritesDirList.Add(fDir);
        if (FavoritesAgent.OnAddFavoritesItem != null)
        {
            FavoritesEventArgs arg = new FavoritesEventArgs(_level, fDir, null);
            FavoritesAgent.OnAddFavoritesItem(this, arg);
        }
        this.ProcessFavoritesDir(fDir);
    }

    foreach (string file in Directory.GetFiles(favoritesDir.Path))
    {
        if (file.EndsWith(".url", true, null))
        {
            UrlFile urlFile = new UrlFile();
            urlFile.FromFile(file);
            favoritesDir.UrlFileList.Add(urlFile);
            if (FavoritesAgent.OnAddFavoritesItem != null)
            {
                FavoritesEventArgs arg = new FavoritesEventArgs(_level, null, urlFile);
                FavoritesAgent.OnAddFavoritesItem(this, arg);
            }
        }
    }
    _level--;
}
```

接下来就是订阅事件创建收藏夹菜单和收藏工具条了。同样由于子节点需要放入父节点的容器中，所以在处理子节点时能够获得对树节点的引用。下面以收藏工具条为例说明如何编写处理OnProcessFavoritesStrip事件的代码。下面那个level表示当前处理的层次的父层。对其的更新放在创建收藏菜单的代码中，查看源码可以看到其方法的最后一句是\_level2 = e.Level;。

```
int _level2 = 0;
```

```

ToolStripDropDownItem _tsddi;
void FavoritesAgent_OnProcessFavoritesStrip(object sender, FavoritesEventArgs e)
{
    ToolStripItem tsi = null;
    ToolStripItemCollection tsic = null;

    if (e.Level == 1)
    {
        if (e.UrlFile != null)
        {
            tsi = this.CreateToolStripButton(e.UrlFile);
        }
        else
        {
            tsi = this.CreateToolStripDropDownButton(e.FavoritesDir);
        }
        tsic = _favoritesStrip.Items;
    }
    else
    {
        this.ResetTsi(e.Level, _level2);
        if (e.UrlFile != null)
        {
            tsi = this.CreateToolStripMenuItem(e.UrlFile);
        }
        else
        {
            tsi = this.CreateToolStripMenuItem(e.FavoritesDir);
        }

        tsic = (_tsi as ToolStripDropDownItem).DropDownItems;
    }

    tsic.Add(tsi);
    _tsi = tsi;

    _level2 = e.Level;
}

```

声明一个ToolStripDropDownItem类型的成员\_tsddi用于保留在创建收藏夹工具条菜单项时当前父界面元素的引用。在事件信息中，如果e.UrlFile != null说明当前加载的是链接文件，否则就是收藏夹下面的文件夹。由于是先根遍历，所以首先处理的肯定是文件夹；否则，只能说明收藏夹下面没有其它的文件夹。根据约定收藏夹下的第一层文件夹的level为1，所以首先执行的是最外层else块的if (e.Level == 1)子块。结果是将代表这个文件夹的ToolStripDropDownButton添加到收藏工具条中，且使\_tsddi指向它。接下开始解析这个文件下的文件夹，执行的是最外层else块的else子块，此时e.Level显然大于0。执行的结果是将代表这个文件夹的ToolStripMenuItem添加到\_tsddi中，即添加到代表上层文件夹的界面元素的子元素集合中，然后将\_tsddi指向这个ToolStripMenuItem。如此往复只到某个文件夹下全是链接文件或者什么都没有，然后开始回到上一层开始处理下一个文件夹，就这样处理完所有收藏夹中的内容。

## 高级主题

### 创建带有Splash Screen和登陆窗体的宿主

假如使用该框架编写的系统包含很多插件，那么系统启动时加载这些插件会耗费一些时间。这样一来，当用户双击后需要等待较长的时间系统主界面才能显示出来，这是很差的用户体验；显示出一个Splash Screen将系统目前正在进行的工作告知用户，能够让用户感觉到的等待时间减少，从而提高了用户体验。

有时需要对使用系统的用户进行限制，就不得不首先弹出一个登陆窗口来验证用户的身份。等到主窗口加载完成后再来验证用户的身份，是很不划算的；假如用户等待至系统启动完成后，失败地通过一次身份验证后发现忘记了口令将会是一件比较郁闷的事情，当然这也是个仁者见仁智者见智的事情。但是在这个插件框架里，如果有登录窗口的话，登录窗口将会首先弹出，如果有SplashScreen接着弹出；因为登录是在一个单独的线程里执行的，在等待用户登录的过程中可以在后台完成插件的加载；从而减少了SplashScreen的保持时间，有时候SplashScreen只用保持设定的最短时间就可以了。

### 创建登录窗口

如果您实际动手编写过一个宿主或者细心查看过随本文档所附带的示例代码，就会发现AppFrame类拥有两个Property——LoginDialog和SplashScreen；LoginDialog就是登录窗口。

在宿主工程中创建一个窗体，让其实现接口ILoginDialog。从窗口编辑器中打开这个窗口，制作好登录窗口的外观。ILoginDialog有一个ShowDialog方法其返回值为bool，这个方法只有在窗口关闭后才返回，返回值表示用户是否确认登录。所以应该为窗口提供一个确认（或者OK以及其它能够表示这个意思的）按钮，并且让这个按钮的返回值是DialogResult.OK。在实现接口的ShowDialog方法中调用base.ShowDialog()，并判断其返回值是否是DialogResult.OK。

```

public new bool ShowDialog()
{

```

```

        bool ret = (DialogResult.OK == base.ShowDialog());
        return ret;
    }

```

在ShowDialog返回之前需要将其bool属性Valid设置为用户通是否过了验证。验证用户权限和这个过程可以放在确认按钮的消息处理函数中进行，如果验证过程过长可以在另外一个线程里进行。此时可以将确认按钮的DialogResult设置为None，点击确认后新开一个线程用于处理验证过程，验证过程结束后需指定对话框的返回值，并将验证结果赋值给Valid，下面是其实现代码。

```

private void btnOK_Click(object sender, EventArgs e)
{
    btnOK.Enabled = false;
    _LoginThread = new Thread(new ThreadStart(Login));
    _LoginThread.Start();
}

private void Login()
{
    Thread.Sleep(5000); //验证用户过程,并将验证结果赋值给_valid
    this.Invoke(new MethodInvoker(delegate() { this.DialogResult = DialogResult.OK; }));
}

```

## 创建SplashScreen

实例化一个实现了ISplashScreen的类，并在类中完成SplashScreen的功能，将其赋给AppFrame的Property——SplashScreen，即可完成SplashScreen功能。还可以通过为AppFrame对象的SplashInterval属性赋值来指示SplashScreen最少展示的时间，其单位是毫秒。

在宿主工程中创建一个窗体，从窗口编辑器中打开这个窗体，制作好外观。让其实现接口ISplashScreen。其实现代码如下。SetInfo用来在界面上显示系统启动中需要显示的一些信息，告诉用户系统当前正在干什么比现实一个静态的窗口要有好的多。其它两个就是显示SplashScreen和关闭SplashScreen，通过调用窗口的ShowDialog和Close方法对其进行了简单的实现。值得注意的是其中采用线程安全的调用方式，因为在AppFrame里SplashScreen是放在另外一个线程里使用的。

```

public void SetInfo(string info)
{
    try
    {
        this.Invoke(new MethodInvoker(delegate() { this.label1.Text = info; }));
    }
    catch
    { }
}

public void CloseSplash()
{
    this.Invoke(new MethodInvoker(this.Close));
}

public void ShowSplash()
{
    this.ShowDialog();
}

```

此时通过编写简单如下的语句就能简单得实现登录窗口和SplashScreen。注册事件在SplashScreen上显示消息的方法可以参考在随文档的示例代码。

```

[STAThread]
static void Main(string[] args)
{
    AppFrame app = new AppFrame();
    app.LoginDialog = new LoginDialog();
    app.SplashScreen = new SplashWin();
    app.SplashInterval = 8000;
    app.Run();
}

```

## 界面逻辑维护与多线程

在这个插件框架里闪屏和登陆窗口在一个线程里运行，所有的插件的服务方法都在主线程（UI线程）里运行，所以服务方法应该在不影响用户体验的时间范围内；那么遇到繁重的任务就需在服务方法里启动一个线程单独执行。有些用于操作文档对象的控件本身具有多线程的特性。比如ESRI、SuperMap的GIS基础平台软件中的三维场景控件，用于在三维中显示地理对象。它的每一个实例都使用一个线程来处理对象的渲染。那么每一个文档对象都运行在一个独立的线程当中。

每个编写过.NET平台多线程的程序都应该知道，WinForm的控件是不能在非创建这些控件的线程中使用的。如果在多个线程里都触发了更新界面的事件，这样就会产生竞争访问界面元素的情形。

所以如果存在多线程，发布界面元素状态事件时一定要判断发布了这个事件后界面逻辑是否合理，此外还需要考虑效率问题。

对于包含多线程的文档操作控件，在直接订阅控件的事件里发布更新界面元素状态事件时一定要首先判断发布这个事件的文档窗体是否是当前文档窗体。

在本浏览器插件中就有这种情况。其代码如下：

```
void page_StatusTextChanged(object sender, EventArgs e)
{
    PageForm page = sender as PageForm;

    if (!page.IsActivated)
        return;

    if (UpdateStatus != null)
    {
        UpdateUiElemEventArgs arg = new UpdateUiElemEventArgs();
        arg.Text = page.WebBrowser.StatusText;
        UpdateStatus(this, arg);
    }
}

void page_ProgressChanged(object sender, WebBrowserProgressChangedEventArgs e)
{
    PageForm page = sender as PageForm;

    if (!page.IsActivated)
        return;

    if (UpdateProgress != null)
    {
        UpdateUiElemEventArgs arg = new UpdateUiElemEventArgs();
        arg.Count = page.Progress;
        arg.Maximum = 100;
        arg.Visible = page.Progress < 100;
        UpdateProgress(this, arg);
    }
}
```

由于本插件框架提供了灵活的插件之间的交互方式，用户可以获取UI 插件并取得界面元素的引用。如果用户需要直接操作界面元素，多线程运行时界面逻辑问题需要自己解决。如果调用UI插件的方法时，则不用考虑多线程问题。

## 如何建立完备的服务集合

这是一个企业级用户的命题。某个企业或某个部门通常会专注于软件的某一应用领域。随着时间的推移，当然会有很多积累。此时可以抽调人力对这些积累进行分析，从中提取出绝大多数用户都会需要的功能，将其按照相关性组织成多个插件。让这些插件构成基础服务集合，为其创建一个接口定义程序集。所有用户的特殊需求都可以实现新的插件独立完成或者与基础服务集合交互完成。

每一个基础服务插件都需要引用接口定义程序集，以使接口对于所有需要交互插件是可见的。通过调用取得的服务列表的获取服务的泛型方法取得需要与之交互的服务类实例。其原型为：

```
public T GetService<T>() where T : class;
```

实际上应该将T约束为接口的，但是好像目前没有这类语法支持。使用时将接口类型参数传入即可获得相应的服务类实例。为什么就一定能够获取呢？本插件框架还隐含了另外一个约定，在一个插件列表中，每一个服务接口只能有一个实现类，因为接口的每一个方法定义了唯一需要提供给用户的功能。而不是像SharpDevelop一样接口定义了一个抽象的Run方法用来完成功能。

请注意，无论使用那种交互方式都不能在需要交互的时候去获取服务交互完成就将获取到的服务丢掉。凡是需要和本插件交互的插件，都应该为之声明一个成员，当接收到FinishLoadAddIn消息时，将其指向从ServiceCollection中获取的服务类的实例。下面是HelloDock程序中获取界面服务的代码。

```
private IUIService _uiService;

public HelloDock()
{
    AppFrame.FinishLoadAddIn += new LoadAddInHandler(AppFrame_FinishLoadAddIn);
}

void AppFrame_FinishLoadAddIn(LoadAddInEventArgs e)
{
    _uiService = AppFrame.GetServiceCollection().GetService<IUIService>();
}
```

事实上界面服务并不需要获取，因为它是如此普遍的被需要以至于需要让其作为**ServiceBase**的成员。

```
public abstract class ServiceBase
{
    public ServiceBase();

    public IUIService UIService { get; }

    public virtual void About();
    public virtual void Config();
}
```

如果一个插件和另外一个插件的交互很少，那就说明需要重新设计。对于良好的插件设计的效果是，插件之间的交互不多不少，如果太多则应该考虑他们是否应该放到一个服务中，如果太少则考虑是否将需要交互的方法从该服务中分离出去。

## 使用混淆器对插件进行版权保护时的注意事项

开发这个系统时，我就希望它非常开放的，在授权声明中已经指出您有任何使用的自由。所以您可以在商业软件中使用该插件框架而不用公开代码，并且可以对编译结果进行加密以保护知识产权。

由于宿主通过注册的插件服务的类名实例化服务，并且通过接口或者方法名调用服务类提供的方法。所以使用混淆器时必须注意：

- 服务类的类名不得改变；
- 服务类的共有方法名不得改变；
- 服务类中用来更新界面的事件名不得改变。

## 深入到此插件框架的内部实现

### 一些基本概念

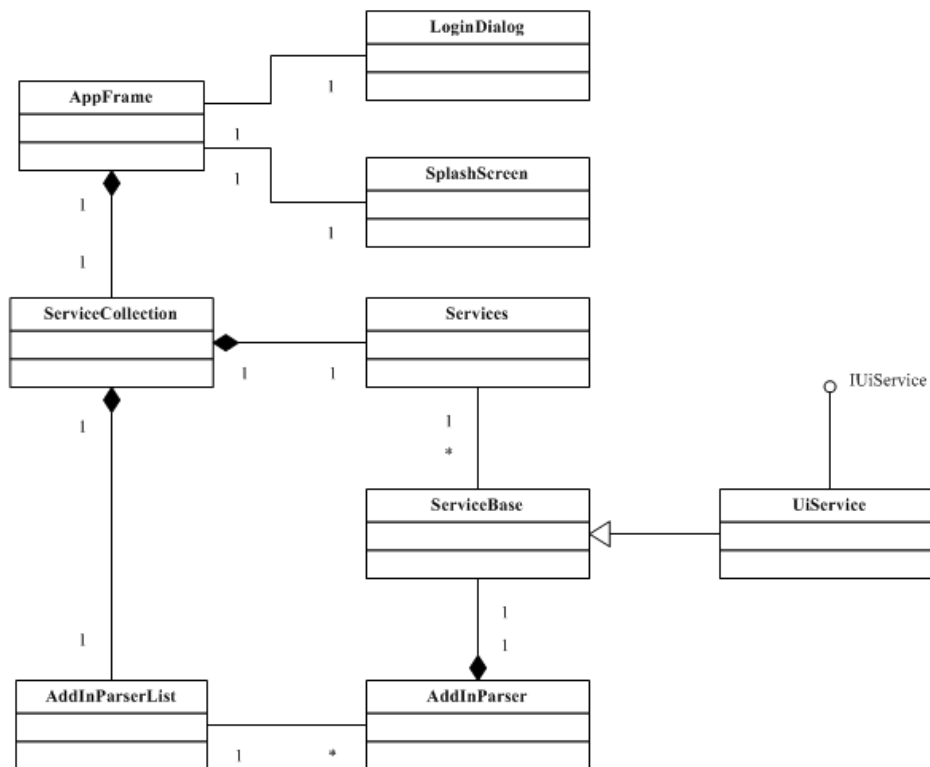
- 服务（Service）——提供一系列功能的对象。在这个插件框架中所有的服务类都必须继承自ServiceBase类。
- 插件（AddIn）——至少包含一个访问控制级别为public的服务类的程序集。
- 插件解析器（AddInParser）——插件解析器，用于解析插件配置文本并在需要的时候根据配置对服务类进行实例化。
- 服务集合（ServiceCollection）——所有服务类的实例（即服务）都包含在这个集合里。一个应用程序只有一个服务集合的实例，可以通过AppFrame类的静态属性获取。服务集合提供了一系列方法用于获取服务。
- 插件宿主应用程序（AppFrame）——在应用程序入口方法里，实例化一个AppFrame，调用其Run方法既可以创建一个插件宿主应用程序。

### 启动过程

首先一个AppFrame的实例被创建，然后进入Run方法执行。在Run方法里会创建一个闪屏线程，用于弹出登录对话框和显示SplashScreen。同时，在主线程里ServiceCollection通过AddInParser将所有插件信息解析出来，创建服务类的实例并将实例放到ServiceCollection的内部容器中；然后从ServiceCollection中获取UIService，将主窗体加载出来；在主线程里等待闪屏线程结束，然后调用Application.Run方法将主窗体显示出来，启动应用程序。UIService在加载主窗体时根据界面配置文件完成界面元素和服务提供的功能之间的连接。

### 系统结构图





本插件框架只定义了一个扩展点——`IUIService`。使用该插件框架开发系统时，需要针对系统需要的功能自定义出所有的扩展点，然后在插件中实现。比如使用该框架开发Mini Internet Explorer的时候，就是定义了一个`IMyIE`接口。开发插件对系统的功能进行扩展的时候，可以将扩展功能定义为接口，也可以不定义接口。扩展功能需要和已有插件进行交互时，通过`ServiceCollection`获取已有插件。

## 如何替换UI插件

UI插件的开发和一般插件开发一模一样，首先创建一个类库项目，然后添加对`AddIn.Core.dll`的引用。新建一个类命名为`MyUIService`，继承自`ServiceBase`，实现`IUIService`接口。在`AddIn.Core.dll`中只需要使用`LoadMainForm`方法，因此只要事实上实现了`LoadMainForm`，插件框架就可以使用了。下面是一个非常简单的演示实现。

```

public System.Windows.Forms.Form LoadMainForm()
{
    return _mainForm = new Form1();
}

public System.Windows.Forms.Form MainForm
{
    get { return _mainForm; }
}

```

然后手动修改插件配置文件的内容为：

```

<?xml version="1.0" encoding="utf-8"?>
<AddIns>
  <AddIn name="MyGui.MyUIService" author="" version="0.0.0.0" copyright="" url="" lazyload="False">
    <path>.\MyGui.dll</path>
    <description>
    </description>
  </AddIn>
</AddIns>

```

编译之后，将所有有关文件集中到一起，将`Startup.exe`添加进来。



然后执行`Startup.exe`，即可将主窗体显示出来。



其它的功能可以自由实现，比如界面定制、插件注册功能等。如果包含界面定制功能，界面的描述文件格式也可以完全自定义。

---

## 第三部分 编程参考

---

### AppFrame

插件宿主应用程序类，实例化该类运行其Run方法就可以创建一个宿主应用程序。一个宿主应用程序只能实例化一个该类。

#### 属性

- static AppFrame Instance { get; }

获取AppFrame的实例。

- UInt16 Authority { get; }

获取运行程序的用户的权限级别。默认值为0，权限级别由框架从登录窗口获得，数值的具体权限含义由您自定义。如果某界面元素的Authority大于AppFrame的Authority值，那么这个界面元素就不会显示。

- Form MainForm { get; }

获取应用程序的主窗体。

- public string[] Args { get; set; }

设置应用程序运行的参数。设置应用程序的参数应该在调用Run方法直接进行。

- ILoginDialog LoginDialog { get; set; }

获取和设置登录对话框。登录对话框的实现方式参考高级主题一章的创建带有Splash Screen和登陆窗体的宿主节。

- ISplashScreen SplashScreen { get; set; }

获取和设置闪屏对象。闪屏的实现方式参考高级主题一章的创建带有Splash Screen和登陆窗体的宿主节。

- UInt16 SplashInterval { get; set; }

获取和设置闪屏最少显示时间。单位毫秒。

- static IServiceCollection ServiceCollection { get; }

获取服务列表。

- static log4net.ILog FrameLogger { get; }

获取日志对象。

#### 方法

- static AppFrame GetInstance()

获取AppFrame的实例。

- static ServiceCollection GetServiceCollection()

获取插件列表。通过该插件列表对象可以获取插件系统中加载的插件所提供的服务对象。

- void Run()

开始运行插件宿主应用程序。

- void RunConfig()

开始运行插件配置工具。

## 事件

- static event LoadAddInHandler AfterLoadOneAddIn

每个插件加载完成后会触发该事件。参数为LoadAddInEventArgs。一般用于在SplashScreen上显示提示信息。

- static event LoadAddInHandler BeforeLoadOneAddIn

每个插件加载前会触发该事件。参数为LoadAddInEventArgs。一般用于在SplashScreen上显示提示信息。

- static event LoadAddInHandler FinishLoadAddIn

所有插件都加载完成后会触发该事件。参数为LoadAddInEventArgs。可用于在SplashScreen上显示提示信息和在插件中获取需要与之交互的服务。

- static event LoadMainFormHandler BeforeLoadMainForm

主窗口加载前会触发该事件。参数为LoadMainFormEventArgs。一般用于在SplashScreen上显示提示信息。

- static event LoadMainFormHandler AfterLoadMainForm

主窗口加载结束后会触发该事件。参数为LoadMainFormEventArgs。可用于在SplashScreen上显示提示信息和用于在主窗体加载完成后立即显示初始界面。

## ServiceCollection/IServiceCollection

### 属性

- System.Collections.Generic.List<AddInParser> AddInParserList { get; }

获取插件信息列表。本插件框架规定向系统注册插件的功能在界面插件中实现，本属性用于实现向系统注册插件。将新增加的插件描述对象AddInParser插入到该列表中，然后调用SaveConfig将列表存储一下就可以将插件注册到系统中。

- System.Collections.Generic.Dictionary<string, ServiceBase> Services { get; }

获取插件服务列表。向系统注册插件时还需要指定，插件提供的某个功能由哪个界面元素触发。本属性用于从插件服务类中反射出实现功能的方法，然后绑定到具体的界面元素上。

### 方法

- ServiceBase GetService(string name)

根据注册的服务名获取服务。

- T GetService<T>() where T : class

根据指定类型获取服务。

- void Load()

加载所有插件。

- void SaveConfig(string configPath)

保存插件配置文件。

## 事件

- event LoadAddInHandler AfterLoadOneAddIn

每个插件加载完成后会触发该事件。参数为LoadAddInEventArgs。一般用于在SplashScreen上显示提示信息，不直接使用这个事件而是使用AppFrame的同名事件。

- event LoadAddInHandler BeforLoadeOneAddIn

每个插件加载前会触发该事件。参数为LoadAddInEventArgs。一般用于在SplashScreen上显示提示信息，不直接使用这个事件而是使用AppFrame的同名事件。

## UIService/IUIService

### 属性

- System.Windows.Forms.Form MainForm { get; }

获取应用程序的主窗体。

### 方法

- void Execute(string exe,string parameter)

执行一个外部程序，exe表示外部程序的路径，可以使相对路径；parameter外部程序运行的参数。

- void Exexute(string exe)

不用参数运行一个外部程序。相当于给方法void Execute(string exe,string parameter)传入的parameter为null。

- void Exit()

退出程序。

- void Config()

配置本插件。

- System.Windows.Forms.ContextMenuStrip GetContextMenuStrip(string str, bool byform);

获取右键菜单。如果byfrom为true，表示通过窗口类型名称获取；如果为false表示通过右键菜单的Text属性获取。

- System.Windows.Forms.StatusStrip GetStatusStrip(string str);

根据状态栏的Text属性获取状态栏。str状态栏的Text值。

- System.Windows.Forms.ToolStrip GetToolStrip(string str, bool byname);

获取工具栏。如果byname为true，表示根据工具条的Name属性获取；否则更具工具条的Text属性获取。

- System.Windows.Forms.ToolStripItem GetToolStripItem(string path)

获取工具条项目。容纳在工具条和状态栏上的按钮等控件都是工具条项。path表示工具条项目的路径，为以斜杠分割的从最上层UI项目一直展开到本UI项目的Text属性值，如MenuStrip/收藏(&B)/。

- System.Windows.Forms.Form LoadMainForm()

根据UI配置载入主窗体。

- void ModifyAddIns()

向系统中注册插件和配置界面。

- void SetStatusStripVisible(bool visible, string name)

设置Name为name的状态栏的可见性。

- void SetToolStripVisible(bool visible, string name)

设置Name为name的工具条的可见性。

- void ShowDocForm(System.Windows.Forms.Form docForm)

显示文档窗体。

- void ShowToolWin(System.Windows.Forms.Form toolWin, System.Windows.Forms.DockStyle dockStyle)

显示工具窗体并指定显示的方式。

## AddInParser

插件列表描述对象，用于从插件列表配置文件中某个插件配置节点解析出有关该插件的一些信息。

### 属性

- System.Reflection.Assembly Assembly { get; set; }

获取和设置该插件描述对象所代表的插件的程序集对象。

- string Author { get; set; }

获取和设置作者。

- string Copyright { get; set; }

获取和设置版权描述信息。

- string Description { get; set; }

获取和设置插件描述信息。

- bool Lazyload { get; set; }

获取和设置是否延迟加载该插件。

- string Name { get; set; }

获取和设置插件服务类的名称。

- string Path { get; set; }

获取和设置插件存放的路径，可以使相对路径。

- ServiceBase Service { get; }

获取该插件描述对象所代表的插件的服务类。

- string Url { get; set; }

获取和设置该提供该插件更详细描述信息的网址。

- bool Valid { get; }

获取和设置该插件是否是有效的。

- string Version { get; set; }

获取设置该插件的版本号。

### 方法

- ServiceBase GetService()

获取该插件描述对象所代表的插件的服务类。

## ILoginDialog

### 属性

- ushort Authority { get; }

成功登录后AppFrame通过该字段获取登录的用户的权限值。数值的实际权限含义由用户自定义。如果某界面元素的Authority大

于AppFrame的Authority值，那么这个界面元素就不会显示。通过AppFrame的Authority属性获取登录到系统的用户的权限值。

- bool Valid { get; }

登录对话框关闭后AppFrame通过该字段获取用户是否通过认证。如果通过认证，使该字段为true；如果该字段为false应用程序将会退出。

## 方法

- bool ShowDialog()

显示登录对话框。对话框关闭后返回true表示用户确定登录，返回false表示用户取消登录。如果返回了false应用程序将会退出。

## ISplashScreen

## 方法

- void SetInfo(string info)

在splashscreen上显示文本。AppFrame会在splashscreen线程显示SplashScreen，在其它线程里调用该方法显示提示，请确保实现时考虑到了线程安全性。

- void ShowSplash()

显示splashscreen。AppFrame会在splashscreen线程中调用该方法显示SplashScreen，在SplashScreen关闭之前该方法不得返回，调用CloseSplash方法后该方法返回，结束显示splashscreen。

- void CloseSplash()

关闭splashscreen。AppFrame会在主线程中调用该方法，请确保实现时考虑到了线程安全性。调用该方法后splashscreen结束显示，ShowSplash方法返回。

## LoadAddInEventArgs

### 属性

- IServiceCollection ServiceCollection { get; }

获取服务列表。

- AddInParser AddInParser { get; }

获取当前加载完成的插件的配置信息。

## LoadMainFormEventArgs

### 属性

- string[] Args { get; }

获取应用程序的执行的参数。如果在某个插件中需要使用该参数，则通过该参数获得。

- IUiService UiService { get; }

获取界面服务。

- Form MainForm { get; }

获取应用程序的主窗体。

## UpdateUiElemEventArgs

### 属性

- bool Checked { get; set; }

插件框架会使用该属性设置界面元素的Checked属性。



- `public bool Enabled { get; set; }`

插件框架会使用该属性设置界面元素的Enabled属性。

- `bool Visible { get; set; }`

插件框架会使用该属性设置界面元素的Enabled属性。

- `int Count { get; set; }`

插件框架会使用该属性设置进度条类界面元素的进度值，或者设置滑块控件指示的值，或者选项控件的选项索引。注意当ComboBox控件的选项索引改变时，将会导致选项索引改变事件触发。一般选项索引改变事件也会用来调用服务方法完成功能。

- `int Maximum { get; set; }`

当使用Count属性设置进度条和滑动条界面元素的值时，需要使用该值指定最大进行值和滑动条的最大值（滑动条最小值为0）。如果该属性取其默认值0，则框架自动采用上次的提供的最大值。

- `string Text { get; set; }`

插件框架会使用该属性设置界面元素的Text属性。

- `object Value { get; set; }`

插件框架使用该属性值设置选项类界面元素的选择项。注意，设置选项类界面元素的选择项时优先使用Count属性。如果要使用该属性，则将Count属性值设置为小于0的值。

---

## 附录

---

### 推荐的第三方界面组件

其实我没有需要推荐的界面组件。下面是我对界面组件库的风格一致性和性能的要求：

1. 窗口的滚动条、listView等控件的滚动条风格要和界面风格协调；这个Janus做不到，DotNetBar要做到还比较麻烦。
2. 如果窗口作为多文档子窗口使用，最小化后的风格仍然需要和正常的结果一致；这个Janus也做不到，DotNetBar也做不到。
3. 绝对不能闪烁；这个DotNetBar做不到。
4. 内存消耗不能有增无减；这个东日的那个IrisSkin2做不到，在随便两个风格间来回换内存就不断增加。
5. 提供了和整体风格一致的MessageBox、OpenFileDialog、SaveFileDialog；这个除了用IrisSkin2方式进行换肤的界面组件库外，在我所测试过的界面组件库中几乎没有能够做到的。

如果需要推荐，那就是《[开源.NET WinForm控件集1.20](#)》。这是对基本控件进行扩展而来的界面组件库，和操作系统的界面风格一致性很高；开发的应用程序界面的整体风格一致性也很高；性能当然也很不错。将这套界面组件搭配其介绍文档中推荐的其它开源界面组件使用，完全能够制作出高度专业的界面。

---