目录

构建插件式的应用程序框架(一)－－－－开篇

作者：纶巾客

说起插件（plug－in）式的应用程序大家应该不陌生吧,记得很早以前有一款很流行的 MP3 播放软件 winmap,它是我记忆里最早认识的一款使用插件模式的应用程序,你可以使用他的插件管理器插入很多的音乐效果器,皮肤,甚至是歌词显示的面板。接下来看到了 Photoshop 使用插件模式管理虑镜。最后发现只要是大一点的应用程序基本都使用了插件式的程序框架，就拿我们最常用的工具来说吧，Visual Studio，Office，Delphi，Eclipse 等等。Eclipse 将插件模式发挥到了及至，因为他是开源的的，所以众多的爱好者，开发出了让人应接不暇的插件。

为什么使用插件式的应用程序框架呢？我的答案就是他为应用程序的功能扩展提供的无限的想象空间。一个应用程序，无论你前期做了多少的市场调查，需求分析做的多么完美，你也只是迎合一部分人的期望，更甚，你只迎合了一部分人的一部分期望，或者一部分人在某一时间的一部分期望。所以当程序发布以后，你依然有机会提供新的功能而不必重新发布程序，人们也可以根据自己的需要来开发新的功能来满足自己的需求，据我所知有很多的软件公司就是专门开发插件来赚钱，真是一举多得，何乐而不为呢？

我们来看一些常见的提供插件模式的应用程序是如何实现插件功能的。据我的使用经验来看，Visual Studio 和 Office 其实都是自动化程序，通过 COM 的方式提供了一组接口。开发人员可以利用这些接口来开发基于 COM 的插件，当插件开发完成后，注册 COM 组件。在 Visual Studio 中你可以使用 Add－in 管理器来选择是否启用插件，而 Office 似乎省去了这一步，一旦你注册了 Office 插件，Office 应用程序在启动的时候会自动加载插件。COM 方式似乎最受微软的宠爱，因为 COM 是一种二进制重用标准，用户可以使用大部分流行的语言来开发插件。当然你也可以使用别的方式，比如普通 DLL，只是这样对于开发人员来说适用面就窄了，因为各个厂商 DLL 的内部结构是不尽相同的，比 VC 开发出的 DLL 和 Borland C＋＋builder 开发出的 DLL 结构就不同，需要专门的工具进行转换。现在，还有另外一种方式，使用 dotNet 的 Assembly，使用 dotNet 的好处是开发简单，使用也同样简单（不需要注册），而且你也可用通过 COM 互操作让开发人员可以使用各种语言进行插件开发，当然用 dotNet 开发还是最简单的，省去不少中间过程。

其实上面介绍的三种方式开发的插件最终还是寄宿在 DLL 中，从中我们就可以看出一些端倪。为什么使用 DLL 呢？DLL 虽然也是 PE 格式，但是他是不能独立运行的，一般情况下，都是在运行时加载到应用程序的内存空间。插件模式正好是利用了这一点，插件不是应用程序的一部分，他以二进制的方式独立存在，由用户决定是否使用他。

那么插件是如何与应用程序进行交互的呢？首先必须有一个契约，应用程序要声明我有哪些功能是可以被插件使用的，并且具备什么条件才能成为我的插件。其次，应用程序不依赖于插件，也就是说，没有你插件，我也可以很好的运行。再次，应用程序必须有一种策略来获取插件存在的位置，比如 Visual studio 是通过注册表的方式。最后，应用程序可以通过某种方式动态的加载插件。

最近工作比较忙，没有时间写 Blog，控件开发总结的那个系列停在那里好久了，汗一个，有空就尽快补上吧。这个系列也先开个头吧，不然又会被自己找各种借口扼杀了。

构建插件式的应用程序框架(二)－－－－订立契约

无论是用 COM 的方式，还是普通 DLL，抑或.NET 方式来实现插件框架，首先要面临的问题

就是如何订立契约。如同我上一篇文章讲到的一样，契约是应用程序和插件之间进行交互的依据和凭证。应用程序必须声明我有什么样的功能可被插件使用，并且插件必须符合什么条件才能被我使用。反之，插件必须要知道应用程序提供什么样的功能，我才能将自己的功能融入到应用程序的体系中。本系列文章主要讲如何使用.NET 实现插件式的应用程序框架，所以其它的方式我就不再提了。

如何使用.NET 订立契约呢？首先想到的 Interface，其次是抽象类，但是在插件模式中我使用接口，因为我们是在满足应用程序的主要目的的基础上来提供附加的插件功能，就这一点来说，接口更灵活，更容易扩展。接下来，如何订立契约的内容呢？这就要根据你的业务需求了，为了讲解的方便，我们定义一个最最基本的插件式应用程序的插件契约。我们做一个假定，我们的应用程序是一个多文档的应用程序，包含一个主菜单栏，一个工具栏，菜单栏可以在程序的上下左右四个方向停靠，另外还有一个状态栏。到后边，如果有必要，我会扩展这个应用程序，让他本身提供更多的可供插件使用的功能。所以就目前而言，我想实现的功能就是让插件为主程序添加工具条，菜单项并实现一些简单的功能。

应用程序向插件提供服务有两种方式，一种是直接再应用程序接口中声明属性或者方法，一种是将应用程序接口声明成一个服务容器。我打算两种方式都用，明确的功能就在接口中直接声明成属性或者方法，另外将应用程序声明成一个服务容器，以方便插入更多的服务功能，提高应用程序的可扩展性。

下边是一个非常简单的应用程序接口定义，对于我们的假定已经足够了。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel.Design;
using System.Windows.Forms;

namespace PluginFramework
{
    public interface IApplication:IServiceContainer
    {
        ToolStripPanel LeftToolPanel { get;}
        ToolStripPanel RightToolPanel { get;}
        ToolStripPanel TopToolPanel { get;}
        ToolStripPanel BottomToolPanel { get;}

        MenuStrip MainMenuStrip { get;}
        StatusStrip StatusBar { get;}
    }
}
```

插件的接口定义：

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

```
using System.Text;

namespace PluginFramework
{
        public interface IPlugin
        {
                IApplication Application { get;set;}
                String Name { get;set;}
                String Description { get;set;}
                void Load();
                void UnLoad();

                event EventHandler<EventArgs> Loading;
        }
}
```

时间又不早了，今天就写到这里，明天接着写。

构建插件式的应用程序框架(三)－－－－动态加载

不管你采用什么方式实现插件式的应用程序框架，核心还是动态加载，换句话说，没有动态加载技术也就无所谓插件式的应用程序框架了。使用 Com 实现的话，你可以利用 Com 的 API 通过 ProgID 来动态创建 COM 对象，如果使用普通 DLL，你需要使用 Windows 的 API 函数 LoadLibrary 来动态加载 DLL，并用 GetProcAddress 函数来获取函数的地址。而使用.NET 技术的话，你需要使用 Assembly 类的几个静态的 Load（Load，LoadFile，LoadFrom）方法来动态加载汇集。

一个 Assembly 里可以包含多个类型，由此可知，一个 Assembly 里也可以包含多个插件，就像前一篇文章所讲，只要它从 IPlugin 接口派生出来的类型，我们就承认它是一个插件类型。那么 Assembly 被动态加载了以后，我们如何获取 Assembly 里包含的插件实例呢？这就要用到反射（Reflection）机制了。我们需要使用 Assembly 的 GetTypes 静态方法来得到 Assembly 里所包含的所有的类型，然后遍历所有的类型并判断每一个类型是不是从 IPlugin 接口派生出来的，如果是，我们就使用 Activator 的静态方法 CreateInstance 方法来获得这个插件的实例。.NET 的动态加载就是这几个步骤。下来，我做一个简单的例子来演练一下动态加载。首先声明一点，这个例子非常简单，纯粹是为了演练动态加载，我们的真正的插件式的应用程序框架里会有专门的 PluginService 来负责插件的加载，卸载。

我们的插件位于一个 DLL 里，所以我们首先创建一个 Class library 工程。创建一个 FirstPlugin 类让它派生于 IPlugin 接口，并实现接口的方法和属性，由于本文的目的是演示动态加载，所以 IPlugin 接口的 Loading 事件我们就不提供默认的实现了，虽然编译的时候会给出一个警告，我们不必理会它。这个插件的功能就是在应用程序里创建一个停靠在主窗体底部的 ToolStrip，这个 ToolStrip 上有一个按钮，点击这个按钮，会弹出一个 MessageBox 显示

“The first plugin”。下面是代码：

我把完整源代码也附上，方便大家使用：源代码下载

```
using System;
using System.Collections.Generic;
using System.Text;
using PluginFramework;
using System.Windows.Forms;

namespace FirstPlugin
{
    public class FirstPlugin:IPlugin
    {
        private IApplication application = null;
        private String name="";
        private String description = "";

        IPlugin Members
    }
}
```

接下来我们创建一个 Windows Application 工程让主窗体派生于 IApplication 接口并实现 IApplication 接口的方法和属性，下来我们声明 1 个 MenuStrip 和 1 个 StatusStrip，让他们分别停靠在窗口的顶部和底端，接下来我们声明 4 个 ToolStripPanel，分别人他们停靠在上下左右四个边，最后我们创建一个 ToolStrip,在上边添加一个按钮，当点击这个按钮的时候，我们动态的加载插件。

为了方便演示，我们把生成的 Assembly 放置到固定的位置，以方便主程序加载，在本例里，我们在应用程序所在的文件夹里创建一个子文件夹 Plugins（E:\Practise\PluginSample\PluginSample\bin\Debug\Plugins），将插件工程产生的 Assembly（FirstPlugin.dll）放置在这个子文件夹。下面是动态加载的代码：

```
private void toolStripButton1_Click(object sender, EventArgs e)
        {
            //动态加载插件，为了方便起见，我直接给出插件所在的位置
            String pluginFilePath = Path.GetDirectoryName(Application.ExecutablePath) +
"\\plugins\\FirstPlugin.dll";
            Assembly assembly = Assembly.LoadFile(pluginFilePath);

            //得到 Assembly 中的所有类型
            Type[] types = assembly.GetTypes();

            //遍历所有的类型，找到插件类型，并创建插件实例并加载
```

```
foreach (Type type in types)
{
    if (type.GetInterface("IPlugin") != null)//判断类型是否派生自 IPlugin 接口
    {
        IPlugin plugin = (IPlugin)Activator.CreateInstance(type);//创建插件实例
        plugin.Application = this;
        plugin.Load();
    }
}
```

记住父窗体指针。

构建插件式的应用程序框架(四)－－－－－服务容器

在构建插件式的应用程序框架(二)－－－－－订立契约一文中，可以看到我们的 IApplication 接口是派生于 IServiceContainer 接口的。为什么要派生于 IServiceContainer 呢？我们来看看 IServiceContainer 的定义，它有几个 AddService 方法和 RemoveService 方法以及从 IserviceProvider 继承过来的 GetService 方法。Service 本身是.NET 设计时架构的基础，Service 提供设计时对象访问某项功能的方法实现，说起来还真拗口。就我看来，ServiceContainer 机制的本质就是解耦合，就是将类型的设计时功能从类型本身剥离出来。如果你把类型的设计时功能也封装到类型里，这样的类型包含了很多只有开发人员才会用到而最终用户根本不需要的功能，使得类型既臃肿有不便于扩展。而将设计时功能剥离出来，这样类型就可以不依赖于特定的设计环境，之所以现在有这么多非官方的.NET 设计环境可能就是这个原因吧。

我们的插件式的应用程序框架正好也需要这样一个松散的架构，我就移花接木把它应用到我们的框架中。

ServiceContainer 是.NET 提供的 IserviceContainer 的实现，如果没有特殊的需要我们不必扩展它，而是直接的利用它。在上一篇文章中我们在实现 IApplication 接口的时候就直接使用的 ServiceContainer。我们在使用 Service 架构的时候，总是倾向于有一个根容器，各个 Service 容器构成了一个 Service 容器树，每一个节点的服务都可以一直向上传递，直到根部，而每一个节点请求 Service 的时候，我们总是可以从根节点获得。我把这个根节点比喻成一个服务中心，它汇总了所有可提供的服务，当某个对象要请求服务（GetService）只需要向根结点发送要获得的服务，根结点就可以把服务的对象传递给它。

从另外一个角度看，ServiceContainer 为我们的插件是应用程序提供了有力的支持，利用 ServiceContainer，你不但可以获得应用程序所提供的所有的功能，而且你还可以通过插件向应用程序添加 Service，而你添加的 Service 又可以服务另外的 Service，这样我们的应用程序框架就更加的灵活了。但是任何东西都是有两面性的，带来灵活的同时也为开发人员的工作增加了复杂度，所以使用 ServcieContianer 开发的应用程序必须提供足够详细的文档，否则开发人员可能根本不知道你到底有多少 Service 可以用，因为很多的 Service 是通过插件提供的，可能应用程序的作者都不会知道程序发布以后会出现多少 Service。

写了这么多，可能接触过 ServiceContainer 的朋友已经觉得罗唆了，没接触过的还是觉得说得莫明其妙。有空接着写，我会创建几个简单的服务演练演练，增强一下感性认识，呵呵。

构建插件式的应用程序框架(五)－－－－管理插件

我们现在已经搭建了插件式的应用程序框架，接下来的工作就是要充实框架的内容，提供基本的服务，也就是 Service。我想首要的任务就是提供插件的管理服务，我在前面的文章也提到了，要实现动态加载必须要知道插件寄宿在哪里，哪些要加载，哪些不加载，这些就是这篇文章要讨论的问题。

首先解决的就是插件放在什么地方，我采取的传统的方法，将插件放到应用程序所在目录下的制定目录，我会在应用程序所在的目录下创建一个文件夹，命名为 Plugins。接下来的工作就是要通知哪些插件是要加载的，哪些是不需要加载的，我会将这些信息放到应用程序的配置文件中的制定配置块中，当应用程序运行的时候，就会读取配置文件，并根据获得的信息加载插件。另外我们的应用程序框架是建立在 Service 基础之上，所以我需要创建一个管理插件的 service。

我们现在定义一个插件管理的 Service 接口。

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

namespace PluginFramework
{
    public interface IPluginService
    {
        IApplication Application { get;set;}
        void AddPlugin(String pluginName, String pluginType, String Assembly, String pluginDescription);    添加插件到list中。
        void RemovePlugin(String pluginName);
        String[] GetAllPluginNames();
        Boolean Contains(String pluginName);
        Boolean LoadPlugin(String pluginName);
        Boolean UnLoadPlugin(String pluginName);
        IPlugin GetPluginInstance(String pluginName);   获取插件实例
        void LoadAllPlugin();
    }
}
```

PluginService 要实现的目标首先是在配置文件中添加/删除要加载的插件以及相关的信息，接下来就是动态的加载插件。我们要定义几个类型：Plugin 配置区块类型，Plugin 元素类型，plugin 元素集合类型，以便我们能够读取插件的信息。

最后我们实现 PluginService：

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

```
using System.Text;
using System.Xml;
using System.Configuration;
using System.Reflection;
using System.Windows.Forms;
using System.IO;
using System.Collections;

namespace PluginFramework
{
    public class PluginService : IPluginService
    {
        private IApplication application = null;
        private PluginConfigurationSection config = null;
        private Dictionary<String, IPlugin> plugins = new Dictionary<string, IPlugin>();
        private XmlDocument doc = new XmlDocument();

        public PluginService()
        {

        }

        public PluginService(IApplication application)
        {
            this.application = application;

        }

        IPluginService Members
    }
}
```

由于代码比较多，我也就不一一列举了，只把比较重要的代码列出来，其余的我会提供源代码的下载。在实现了 **PluginService** 以后，我们需要有一个地方能够使用这个 **Service** 来管理插件，我的做法是在一个菜单里添加一个项目，当用户点击这个项目的时候弹出插件管理的对话框，用户在这个对话框中选择使用那些插件，当插件被选中的时候，插件会被立即加载进来，并且记录到配置文件里，当用户下次运行应用程序的时候，插件默认会被自动的加载。

另外从现在开始我们就需要使用配置文件了，所以，我们需要给应用程序添加一个 **app.config** 文件，文件内容如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
```

```
        <section
            name="PluginSection"
            type="PluginFramework.PluginConfigurationSection, PluginFramework"
            />
    </configSections>
    <PluginSection>

    </PluginSection>
</configuration>
```

样子，总体来说我们就为 **Plugin** 的管理提供了一个基本的实现，如果大家还有什么不明白的地方，可以参考我提供的源代码或者通过 **e-mail** 和我联系。

构建插件式的应用程序框架(六)－－－－通讯机制

前天发了构建插件式的应用程序框架(五)－－－－管理插件这篇文章，有几个朋友在回复中希望了解插件之间是如何通讯的。

这个系列的文章写到这里，也该谈谈这个问题了，毕竟已经有了插件管理。不知道大家有没有注意到我在第四篇文章里谈到的服务容器（**Service Container**），**Service** 是我所提到的插件式的应用程序框架的基础，我们也可以回头看看 **IApplication** 的接口定义,**IApplication** 是派生于 **IServiceContainer**。我把应用程序提供的相关的功能抽象成一个一个的 **Service**，比如文档管理的，我们就抽象成 **IDocumentService**，停靠工具栏管理功能抽象成 **IDockBarService**，菜单管理的功能抽象成 **IMenuService**，等等。我在第四篇文章里也提到了“我们在使用 **Service** 架构的时候，总是倾向于有一个根容器，各个 **Service** 容器构成了一个 **Service** 容器树，每一个节点的服务都可以一直向上传递，直到根部，而每一个节点请求 **Service** 的时候，我们总是可以从根节点获得。我把这个根节点比喻成一个服务中心，它汇总了所有可提供的服务，当某个对象要请求服务（**GetService**）只需要向根结点发送要获得的服务，根结点就可以把服务的对象传递给它。”

IApplication 是从 **IServiceContainer** 接口派生出来的，而我们的应用程序主窗口又是从 **IApplication** 接口派生出来的，所以，我们的应用程序主窗口就是一个 **ServiceContainer**。从 **IPlugin** 的定义来看，它有一个 **IApplication** 接口属性，这个 **IApplication** 属性是什么时候指定的呢，在第五篇文章的源代码里我们看到，当每一个 **Plugin** 被实例化的时候，由 **PluginService** 指定的，所以在每一个 **Plugin** 被 **Load** 之前，**IApplication** 已经被指定，而代表这个 **IApplication** 接口的实例正是我们的应用程序主窗口，而它正是我们所需要的服务容器。一旦我们能够获得 **IApplication** 实例，我们就可以获得整个应用程序所提供的所有的服务。假设我们要获得文档服务，就可以使用 **Plugin** 的 **Application.GetService(typeof(IdocumentService))**来得到文档服务的实例，接着我们就可以使用这个实例来完成某项功能，比如添加一个新文档等等，其实在第五篇文章的源代码就有这样代码：

```
private void CheckExistedPlugin()
        {
            IPluginService                              pluginService                =
(IPluginService)application.GetService(typeof(IPluginService));
            if (pluginService != null)
```

```
{
        List<String> nameList=new List<string>();
        String[] pluginNames = pluginService.GetAllPluginNames();
        nameList.AddRange(pluginNames);
        foreach (ListViewItem item in listView1.Items)
        {
            if (nameList.Contains(item.Text))
            {
                item.Checked = true;
            }
        }
    }
```

当然，要在插件中获得实例，你必须在应用程序里或者其他插件里实例化服务对象，然后添加到服务容器里，还拿上边的例子，我们在应用程序里实例化了 **PluginService**，然后添加到了容器里，代码如下：

```
public MainForm()
{
    InitializeComponent();
    pluginService = new PluginService(this);
    serviceContainer.AddService(typeof(IPluginService), pluginService);
}
```

稍后，我会继续完善这个例子，做一个简单的多文档编辑器来做演示，并提供一些基础的服务，以便大家阅读。

构建插件式的应用程序框架(七)－－－－－基本服务

既然做好了框架，我们就希望为某个目标服务，我们要提供一些基本的服务，方便用户继续扩展他的功能。首先想到的功能就是，菜单，工具栏的管理，接下来我们要实现一些更流行的功能，比如停靠工具栏等等。

如何实现这些服务呢？我们希望我们的插件在运行时可以获得应用程序本身的菜单，工具条，停靠工具栏等等，然后向他们添加项目，比如加入一个菜单项，添加一个工具栏按钮。为了在运行时获得某个菜单或者工具栏，我们要为每一个菜单后者工具栏分配一个 **Key**，然后放到一个词典中，当需要的时候，我们通过这个 **key** 来获得实例。对于这个 **Key** 呢，在我的例子比较简单就是他的名字，我们来看看 **ToolStripService** 的代码：

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

```
namespace PluginFramework
{
    public class ToolStripService:IToolStripService
    {
        private IApplication application = null;
        private Dictionary<String, ToolStrip> toolStrips = new Dictionary<string, ToolStrip>();

        public ToolStripService(IApplication application)
        {
            this.application = application;
        }

        IToolStripService Members
    }
}
```

对于视图或者是停靠工具栏来说，最好是不要直接在词典中放入实例，而是应该将对象的类型放入到词典中，因为，视图和停靠工具栏本身都是从 Form 派生而来，所以，当视图或者是停靠工具栏被关闭的时候，对象就被销毁了，而对象的创建在是插件的 Load 方法里完成的，我们不可能再去调用插件的 Load 方法，这样给我们的使用带来了不便，所以我们应该注册类型，然后在 Service 中实现一个 Show 方法是比较合理的，这里为了演示方便，我就直接在 Load 里面实例化了，并把实例放到了词典里。

下边这个图例里显示了插件加入的停靠工具栏，工具栏，一个新的菜单“View”和 View 菜单的子菜单：

最近实在是没有时间，文章发的很慢，也写的很错，说的不清楚的地方，可以参考一下源代码，望各位朋友见谅。

构建插件式的应用程序框架(八)－－－－视图服务的简单实现

我在前一篇文章里提到，对于停靠工具栏或者是视图最好是不要将实例放到词典中，而是将工具栏或者视图的类型放到词典中，因为视图类型会经常的被重用，并且会经常被关闭或者再打开。当实例被关闭后，资源就被释放了，对于实例的管理就会比较麻烦，所以我们分为两步走。在插件被加载的时候，我们只注册类型，在应用程序运行的时候，我们通过某种途径来实例化他。

我修改的以前的例子，主要突出本次演示的功能。这次的例子实现的功能是通过插件扩展应用程序处理不同文件的能力。在原始的应用程序中，我们可以通过 File 菜单的 Open，只能打开一种文件，就是文本文件，大家可以在例子中看到，当我们没有加载插件的情况下，在 OpenFileDialog 的 Filter 中只有"Text（*.txt）"。选择一个文本文件以后，将会出现文本文件视图。当我们加载插件以后，在点击 File->Open 菜单，我们观察 Filter，发现会多出两种文件："JPEG"和"BMP"，这是我们就可以打开图片文件，选中文件以后，将会出现 Picture 视图，并且在主菜单下边，还会出现一个工具栏，点击工具栏上的按钮，可以给图片加上水印，

并且工具栏会根据 PictureView 的状态（Active）显示和消失。比如你打开了一个文本视图和一个图片视图，当你切换到文本视图的时候，工具栏就会消失，再切换到图片视图的时候，工具栏又会出现。

我在框架里面添加了一个 IDocumentViewService 的接口，用以描述服务的功能：

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

namespace PluginFramework
{
    public interface IDocumentViewService
    {
        void RegisterView(String fileType,string fileFilter,Type viewType);
        void ShowView(String fileType, String filePath);
        void RemoveRegister(String fileType);
        String GetFileFilter(String fileType);
        String GetFileTypeByFileFilter(String fileFilter);

        StringCollection FileTypies { get;}
    }
}
```

下面是这个服务的实现：

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

namespace PluginFramework
{
    public class DocumentViewService:IDocumentViewService
    {
        private Dictionary<String, Type> docViewRegister = new Dictionary<string, Type>();
        private Dictionary<String, String> fileTypeToFileFilter = new Dictionary<string, string>();
        private Dictionary<String, String> fileFilterToFileType = new Dictionary<string, string>();
        private IApplication application = null;

        public DocumentViewService(IApplication app)
        {
            application = app;
        }
```

```
IDocumentViewService Members#region IDocumentViewService Members

public void RegisterView(string fileType, string fileFilter, Type viewType)
{
        docViewRegister[fileType] = viewType;
        fileTypeToFileFilter[fileType] = fileFilter.ToUpper();
        fileFilterToFileType[fileFilter.ToUpper()] = fileType;
}

        public void ShowView(string fileType, string filePath)
        {
            if(docViewRegister.ContainsKey(fileType))
            {
                IDocumentView docView = null;
                try
                {
                    docView                                                                    =
(IDocumentView)Activator.CreateInstance(docViewRegister[fileType]);
                    docView.Application = application;
                    docView.ShowView(filePath);
                }
                catch
                {

                }

            }
        }

public void RemoveRegister(string fileType)
{
        docViewRegister.Remove(fileType);
}

public StringCollection FileTypies
{
        get
        {
            StringCollection sc = new StringCollection();
            foreach (String key in docViewRegister.Keys)
            {
                sc.Add(key);
            }
```

```
            return sc;
        }
    }

    #endregion

    IDocumentViewService Members#region IDocumentViewService Members

    public string GetFileFilter(string fileType)
    {
        String result = "";
        if (fileTypeToFileFilter.ContainsKey(fileType))
        {
            result = fileTypeToFileFilter[fileType];
        }
        return result;
    }

    #endregion

    IDocumentViewService Members#region IDocumentViewService Members


    public string GetFileTypeByFileFilter(string fileFilter)
    {
        String result = "";
        if (fileFilterToFileType.ContainsKey(fileFilter))
        {
            result = fileFilterToFileType[fileFilter];
        }
        return result;
    }

    #endregion
    }
}
```

时间比较紧，写的比较粗糙。另外定义了 DocumentView 的基本功能，就是需要打开的文件的路径，以及显示的方法。再插件了，我实现的一个 PictureView，为两种文件注册了这个视图类型，大家可以根据自己的需要继续扩展。转眼又十一点多了，明天还要上班，就写到这里了，又说的不清楚的地方，大家可以参考一下源代码。