沙漠之鹰

开发笔记，理解和思考



# .NET插件系统（三） 插件间通信问题——设计可自组织和注入的组装程序

**一. 问题的背景**

动态系统的要求之一，是不同模块可以根据自身需求自动组装，这往往通过配置文件或用户选择进行。 这个基本问题在前面的文章中已经讲述过了。

但新的问题来了，我们定义了不同的插件A,B,C，那么，不同插件之间的通信如何进行？

如果系统本身的框架非常明晰而且不易更改，那么面向固定接口的方法是最简单方便的。 这也是大部分插件系统在"主结构"上使用的做法。

但是，如果系统框架本身非常易变，**连他们之间交互的接口都会随着问题的不同而不同。**这就好像，系统中包含不同种类的插座和插头，我们需要自动将符合要求的插座和插头安装好，实现自动组网。如何实现这种自组织的组装程序呢？

**二. 具体的案例**

为了便于更好的说明问题，以一个我实际面对的设计问题进行分析，实在抱歉由于时间所限不能提供Demo.

我需要开发一个数据挖掘和处理平台，不同的功能通过插件的形式接入系统，并形成如左边的可执行列表：


用户可以很简单的通过拖拽，将左边的处理拖到后边的算法执行列表中，当点击"运行"按钮时，列表中的算法模块会被顺序或并行执行。

这些算法是多样的，比如数据挖掘中常用的 数据筛选，分词，聚类，数据分类显示等功能。你可以不必了解这些算法本身是什么，但本文中，您可能需要了解他们之间的结果可能相互依赖。这些算法的共同特征，是必须依赖于前一个或多个算法的结果，同时本身还可以输出给其他算法。 例如，数据分类显示必须依赖于聚类的结果，聚类则必须依赖于前期分词和数据筛选的功能。

这种结构很像顺序流动的数据流，或者像电网或自来水网的结构。那么问题来了，系统执行前不知道这些算法到底是什么，那怎么能提供插件间交互的需求？ 一种做法是，读写数据库，只要上一个算法告诉下一个算法数据的位置在哪里就可以了，但这种做法很不"环保"，试想，好好的存在内存中的数据，干嘛要写到硬盘中再读出来呢？这会造成无谓的开销。

另外，算法执行列表（右边）的顺序应该与组装顺序无关，意思是处在数据流上游的模块不一定就在执行列表的上游。

我们必须设计一套方法，能实现这些算法的相互通信。

### 三．声明可提供接口和注入接口需求

首先，为了保证重用，算法模块之间的通信方式只能是接口或抽象类。不论如何，**算法应该告诉管理器，它必须依赖什么，它可以提供什么。**

**如果一个算法模块可以提供某接口的结果，那么它必须实现该接口。**

**如果算法必须依赖某接口，那么它应该最少包含一个该接口的内部成员，或者，也实现之（本文没有考虑这种情况）。**

下面我们简单实现两个类：

计算方法A可以输出接口B和C,但计算方法B必须得到两个接口B和C的结果。

```
[SelfBuildClassAttribute(new string[] { }, new string[] { "IB", "IC" })]
   [XFrmWorkAttribute("计算方法A", "IDataProcess", "可输出接口B和C", "123")]
   public class Test1 : AbstractProcessMethod, IC, IB
   {
       public string outputC
       {
           get;
           set;
       }

       public string outputB
       {
           get;
           set;
       }
       public override bool DataProcess()
       {
           outputC = "已经正确赋值C";
           outputB = "已经正确赋值B";
           return true;
       }

   }




   [SelfBuildClassAttribute(new string[] { "IB", "IC" }, new string[] { })]
   [XFrmWorkAttribute("计算方法B", "IDataProcess", "必须通过外界提供B和C的接口", "123")]
   public class Test2 : AbstractProcessMethod
   {
       [SelfBuildMemberAttribute("IB")]
       public IB calledIB { get; set; }

       [SelfBuildMemberAttribute("IC")]
       public IC callIC { get; set; }
       public override bool DataProcess()
       {

           XLogSys.Print.Debug(calledIB.outputB);
           XLogSys.Print.Debug(callIC.outputC);
           return true;
       }
   }
```

两个方法方法非常简单，继承于AbstractProcessMethod类，你不需要关心这个类的具体内容，只需注意 Test1实现了两个接口IB和IC，这两个接口都能提供两个字符串。Test2类则必须获得IB和IC两个接口的字符串成员。

**我们可以通过自定义Attribute实现可提供和依赖的接口的标识。** 本系统中使用了两个自定义的attribute：

- [SelfBuildClassAttribute(new string[] { }, new string[] { "IB", "IC" })]

两个必选形参，即需求的接口字符串列表 和 提供的接口字符串列表。

- [SelfBuildMemberAttribute("IC")]

要求被注入的需求者成员变量标识

（XFrmWorkAttribute是插件的标记，详情可见我上一篇关于插件的文章）

```
/// <summary>
/// 实现自组织算法的特性，它一般标记在模块的类名之前
/// </summary>
   public class SelfBuildClassAttribute:Attribute
    {
       /// <summary>
/// 要求的依赖项接口
/// </summary>
       public ICollection<string> dependInterfaceCollection
        {
            get;
            set;
        }
       /// <summary>
/// 可以输出的接口
/// </summary>
       public ICollection<string> outputInterfaceCollection
       {
           get;
           set;
       }
       public SelfBuildClassAttribute(string[] dependInterfaceName, string[] outputInterfaceName)
       {
           dependInterfaceCollection = new List<string>();
           outputInterfaceCollection = new List<string>();
           foreach (string rc in dependInterfaceName)
           {
               dependInterfaceCollection.Add(rc);
           }
           foreach (string rc in outputInterfaceName)
           {
               outputInterfaceCollection.Add(rc);
           }
       }
       public SelfBuildClassAttribute(ICollection<string> dependInterface, ICollection<string> outputInterfaceN
ame)
       {
           dependInterfaceCollection = dependInterface;
           outputInterfaceCollection = outputInterfaceName;
       }
    }


    /// <summary>
/// 自组织成员特性，一般放置在类的  要求注入的成员名上
/// </summary>
    public  class SelfBuildMemberAttribute:Attribute
    {
        public string invokeName{get;set;}  //需要被注入的依赖项接口
        public SelfBuildMemberAttribute(string myInvokeName)
        {
            invokeName = myInvokeName;
        }
    }
```

这两个类的作用已经在注释上写清楚了，您可以结合Test1和Test2两个类的具体实现来理解： Test1不需要依赖任何接口，但可以输出两个接口IB,IC。 Test2方法需要依赖IB和IC两个接口，因此它有两个成员变量，并加上了标记，标记的内容是该接口的名称。

下面，我们要做的工作，就是在运行时，自动将test1的方法注入到Test2的内部接口上。

### 四． 实现内部组装

当用户点击运行时，系统会自动实现接口装配，并按照执行策略执行列表当中的算法模块。

```
/// <summary>
   /// 可实现自组织的算法模块设计
```

```
/// </summary>
/// <typeparam name="T"></typeparam>
 public class SelfBuildProcessMethodCollection<T>:ProcessMethodCollection<T> where T:IProcess
{


     SelfBuildManager mySelfBuildManager = new SelfBuildManager();
     /// <summary>
     /// 是否设置自动装配算法模块
     /// </summary>
     public bool isSelftBuild = true;
     protected override bool BeginProcess()
     {

         mySelfBuildManager.BuildModule(this);  //实现接口的自动装配
         base.BeginProcess();


         return true;


     }
}
```

在我的系统中，所有算法的抽象接口都是Iprocess，但在这篇文章中，自组织并不一定需要该接口。系统保存的算法保存在了ICollection<IProcess>接口中。而具体装配的方法，则定义在SelfBuildManager中。

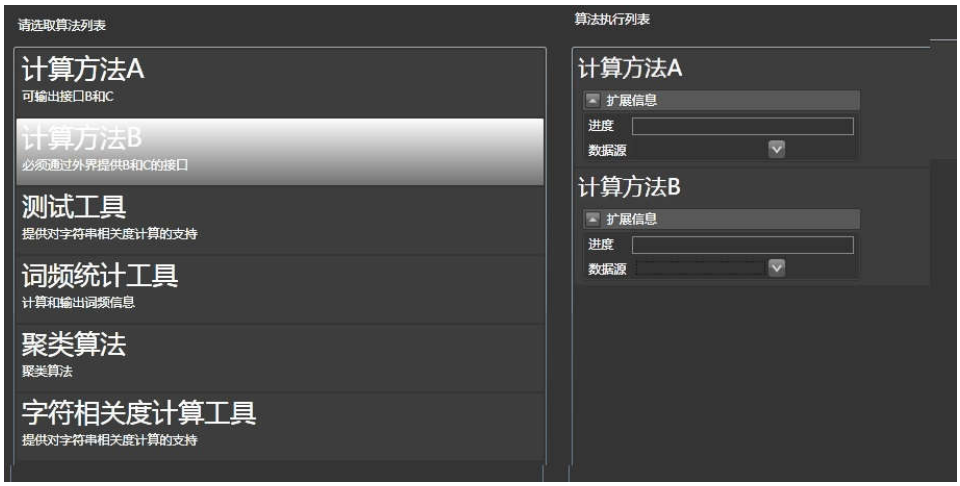
```
public class SelfBuildManager
{
    /// <summary>
    /// 保存所有依赖接口的字典
    /// </summary>
    Dictionary<string, IProcess> dependDictinary = new Dictionary<string, IProcess>();
    /// <summary>
    /// 可提供接口的集合字典
    /// </summary>
    Dictionary<string, IProcess> outputDictinary = new Dictionary<string, IProcess>();
    public void BuildModule(ICollection<IProcess> processCollection)
    {
        myProcessCollection = processCollection;
        GetAllDependExportDictionary();  //获取所有依赖和能提供的接口字典
        BuildAttribute();   //实现接口自动组装的方法
    }

    ICollection<IProcess> myProcessCollection;
    private void GetAllDependExportDictionary( )
    {
        foreach (IProcess rc in myProcessCollection)
        {
            Type type = rc.GetType();
            // Iterate through all the Attributes for each method.
            foreach (Attribute attr in
                type.GetCustomAttributes(typeof(SelfBuildClassAttribute), false))
            {
                SelfBuildClassAttribute attr2 = attr as SelfBuildClassAttribute;
                foreach (string outputString in attr2.outputInterfaceCollection)
                {
                    outputDictinary.Add(outputString, rc);
                }
                foreach (string dependString in attr2.dependInterfaceCollection)
                {
                    dependDictinary.Add(dependString, rc);
                }
            }
        }
    }


    private void BuildAttribute()
    {
        foreach (KeyValuePair<string, IProcess> dependNeeder in dependDictinary)
```

```
            {
                IProcess outputProvider;
                outputDictinary.TryGetValue(dependNeeder.Key,out  outputProvider);       //在输出字典中找到满足该依赖项
的接口
                if (outputProvider != null)
                {
                    PropertyInfo[] PropertyInfoArray=dependNeeder.Value.GetType().GetProperties();    ///获取该类的
所有属性列表
                    foreach (PropertyInfo fl in PropertyInfoArray)
                    {
                        foreach (Attribute attr in fl.GetCustomAttributes(typeof(SelfBuildMemberAttribute), false)
)
                        {
                            SelfBuildMemberAttribute attr2 = attr as SelfBuildMemberAttribute;   //找到自装配成员的标
记
                            if (attr2 != null && attr2.invokeName == dependNeeder.Key)
                            {
                                try
                                {

                                    fl.SetValue(dependNeeder.Value, outputProvider, null);   //通过反射，将提供者注入
到需求者的变量中
                                }
                                catch (System.Exception ex)
                                {
                                    XLogSys.Print.Error(ex.Message+"无法进行组装");
                                }

                                break;

                            }
                        }

                    }

                }



            }

        }
    }
```

具体的方法请参考代码的注释部分。由于代码注释已经很详细了，因此不做更多解释。

### 五. 实现和验证

我们将计算方法A和计算方法B都拖入算法执行列表中：




并单击执行按钮：

```
2012-03-25 17:56:16 INFO 已经成功添加计算方法A到当前列表
2012-03-25 17:56:17 INFO 已经成功添加计算方法B到当前列表
2012-03-25 17:56:20 INFO 计算方法A的计算已经开始
2012-03-25 17:56:20 INFO 计算方法B的计算已经开始
2012-03-25 17:56:20 INFO 成功获取注入变量IB已经正确赋值B
2012-03-25 17:56:20 INFO 成功获取注入变量IC已经正确赋值C
2012-03-25 17:56:20 INFO 计算方法B已经处理完成
```

可以看到，接口确实被正确赋值了。设计成功。

**六. 必须考虑的问题和扩展点**

虽然设计成功，但系统有一些不可避免的问题：

1. 如果一个需求者发现有不止一个满足该需求的提供者，那么如何选择？目前系统未作此区分，仅仅在找到第一个适配对象后停止搜索。合适的方法是提供用户介入的控制方案，即用户可以用线将不同算法的需求和提供联系起来，当然，该需求暂时有些复杂，如果作者实现了它，一定会公开其方法。
2. 性能和灵活性： 通过反射实现的方法必须讨论性能，好在系统只执行一次装配过程，并尽可能的通过标记简化搜索条件。 但应该研究更好的搜索方法。
3. 该功能的易用性： 作者本人认为该系统是足够易用的，你可以简单地将需求和提供接口的字符串列表标记在类前，并将需求的接口标记在需求方的成员变量前，暂时没有想到更好的做法。
4. 相互依赖问题：一种可能的情况是算法A依赖算法B的结果，算法B依赖A的结果，这种情况一定是不允许的吗？不一定，但若能处理这种需求，就可能实现更强的灵活性，同时带来更复杂的组装逻辑。

有任何问题，欢迎讨论！



**标签：** .NET， 插件， 反射， 自组装



FerventDesert
关注 - 6
粉丝 - 905
+加关注



« 上一篇：.NET插件系统之二——不实例化获取插件信息和可视化方法
» 下一篇：NET插件系统之四——提升系统搜索插件和启动速度的思考

posted @ 2012-03-25 18:24 FerventDesert 阅读(3174) 评论(4) 编辑 收藏

评论列表

#1楼 2012-03-25 20:29 SpeakHero

好文章

支持(0) 反对(0)

#2楼 2012-03-26 09:07 小小松

不错.....

支持(0) 反对(0)

#3楼 2012-03-26 11:27 SpeakHero

没有完整的demo 感觉就是空话

支持(0) 反对(0)

#4楼[楼主] 2012-03-26 12:56 FerventDesert

@ SpeakHero
不是所有的文章都有demo啊，主要是它依附的东西比较多，提供一个单独的DEMO有点困难，不过我觉得如果真需要解决这个问题的话，看我的文章应该会得到启发吧

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

**注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。**




**最新IT新闻**:
· 中国发射一枚超级卫星：飞机高铁上将实现高速上网
· 赶在厄玛飓风之前登陆之前 谷歌地图提供实时封闭路段信息
· 号称打败谷歌翻译的DeepL究竟靠不靠谱？
· 开发者发现新证据 iTunes商店即将提供HDR格式内容
· 信用服务公司Equifax数据泄露 涉及1.43亿用户
» 更多新闻...


**最新知识库文章**:
· 做到这一点，你也可以成为优秀的程序员
· 写给立志做码农的大学生
· 架构腐化之谜
· 学会思考，而不只是编程
· 编写Shell脚本的最佳实践
» 更多知识库文章...