[基于AppDomain的"插件式"开发](http://www.cnblogs.com/kongyiyun/archive/2011/08/01/2123459.html)

很多时候,我们都想使用(开发)USB式(热插拔)的应用,例如,开发一个WinForm应用,并且这个WinForm应用能允许开发人员定制扩展插件,又例如,我们可能维护着一个WinService管理系统,这个WinService系统管理的形形色色各种各样的服务,这些服务也是各个"插件式"的类库,例如:

public interface IJob

{

void Run(DateTime time);

}

public class CollectUserInfo : IJob

{

public void Run(DateTime time)

{

//doing some thing...

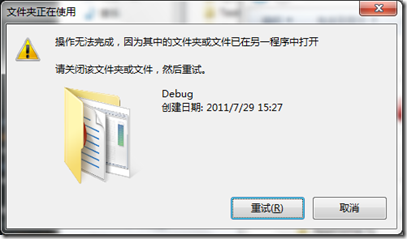
}

}

我们提供了一个IJob接口,所有"服务"都继承该接口,然后做相关的配置,在服务启动时,就可以根据配置,反射加载程序集,执行我们预期的任务.

# 更新程序集(dll/exe)

服务/插件程序(后面只称为服务,虽然两者应用不同,但是在此处他们所运用的原理和作用是相同的 :-) )很健稳的运行着.但在服务/插件程序运行一段时间之后,某些"插件"的业务需求发生的变化,或者版本升级等种种外部原因,导致我们对原本的"插件"程序集进行了升级(可能从v1.0升级至v2.0).当我们想像Asp.net应用一样.把新的dll替换旧dll的时候,错误发生了.

[](http://images.cnblogs.com/cnblogs_com/kongyiyun/201108/201108010957381943.png)

发生该错误的原因很简单,因为我们的程序中已经调用了该dll,那么在CLR加载该dll到文件流中也给其加了锁,所以,当我们要进行覆盖,修改,删除的时候自然就无法操作该文件了.那我们该怎么做?为什么Asp.net可以直接覆盖?

# AppDomain登场

我们知道,AppDomain是.Net平台里一个很重要的特性,在.Net以前,每个程序是"封装"在不同的进程中的,这样导致的结果就造就占用资源大,可复用性低等缺点.而AppDomain在同一个进程内划分出多个"域",一个进程可以运行多个应用,提高了资源的复用性,数据通信等.详见[应用程序域](http://msdn.microsoft.com/library/ZH-CN/113A8BBF-6875-4A72-A49D-CA2D92E19CC8(VS.100))

CLR在启动的时候会创建系统域(System Domain),共享域(Shared Domain)和默认域(Default Domain),系统域与共享域对于用户是不可见的,默认域也可以说是当前域,它承载了当前应用程序的各类信息(堆栈),所以,我们的一切操作都是在这个默认域上进行."插件式"开发很大程度上就是依靠AppDomain来进行.

# "热插拔"实现说明

当加载了一个程序集之后,该程序集就会被加入到指定AppDomain中,按照原来的想法,要实现"热插拔",只要在需要使用该"插件"的时候,加载该"插件"的程序集(dll),使用结束后,卸载掉该程序集便可达到我们预期的效果.加载程序集很简单,.C#提供一个Assembly类,方便又快捷.

var \_assembly = Assembly.LoadFrom(assemblyFile);

Assembly提供了数个加载方法详见Assembly类.

然后,C#却没有提供卸载程序集的方法,唯一能卸载程序集的方法只有卸载该程序集所在的AppDomain,这样,该AppDomain下的程序集都会被释放.知道这一点,我们便可以利用AppDomain来达到我们预期的效果.

# AppDomain实现"热插拔"

首先,我们需要先实例化一个新AppDomain作为"插件"的宿主.在实例化一个Domain之前,先声明该Domain的一些基本配置信息

AppDomainSetup setup = new AppDomainSetup();

setup.ApplicationName = "ApplicationLoader";

setup.ApplicationBase = AppDomain.CurrentDomain.BaseDirectory;

setup.PrivateBinPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "private");

setup.CachePath = setup.ApplicationBase;

setup.ShadowCopyFiles = "true"; //启用影像复制程序集

setup.ShadowCopyDirectories = setup.ApplicationBase;

AppDomain.CurrentDomain.SetShadowCopyFiles();

setup.ShadowCopyFiles = "true";这句很重要,其作用就是启用影像复制程序集,什么是[影像复制程序集](http://msdn.microsoft.com/library/ZH-CN/DE8B8759-FCA7-4260-896B-5A4973157672(VS.100)),复制程序集是保证"热插拔"

实现的主要工作.AppDomain加载程序集的时候,如果没有ShadowCopyFiles,那就直接加载程序集,结果就是程序集被锁定,相反,如果启用了ShadowCopyFiles,则CLR会将准备加载的程序集拷贝一份至CachePath,再加载CachePath的这一份程序集,这样原程序集也就不会被锁定了. AppDomain.CurrentDomain.SetShadowCopyFiles();的作用就是当前AppDomain也启用ShadowCopyFiles,在此,当前AppDomain也就是前面我们说过的那个默认域(Default Domain),为什么当前域也要启用ShadowCopyFiles呢?

主AppDomian在调用子AppDomain提供过来的类型,方法,属性的时候,也会将该程序集添加到自身程序集引用当中去,所以,"插件"程序集就被主AppDomain锁定,这也是为什么创建了单独的AppDomain程序集也不能删除,替换(释放)的根本原因

利用SOS,可以很清楚的看到这一点

0:018> !dumpdomain

--------------------------------------

System Domain: 5b912478

LowFrequencyHeap: 5b912784

HighFrequencyHeap: 5b9127d0

StubHeap: 5b91281c

Stage: OPEN

Name: None

--------------------------------------

Shared Domain: 5b912140

LowFrequencyHeap: 5b912784

HighFrequencyHeap: 5b9127d0

StubHeap: 5b91281c

Stage: OPEN

Name: None

Assembly: 00109de0 [C:\Windows\Microsoft.Net\assembly\GAC\_32\mscorlib\v4.0\_4.0.0.0\_\_b77a5c561934e089\mscorlib.dll]

ClassLoader: 00110f68

Module Name

58631000 C:\Windows\Microsoft.Net\assembly\GAC\_32\mscorlib\v4.0\_4.0.0.0\_\_b77a5c561934e089\mscorlib.dll

--------------------------------------

Domain 1: 000f4598

LowFrequencyHeap: 000f4914

HighFrequencyHeap: 000f4960

StubHeap: 000f49ac

Stage: OPEN

SecurityDescriptor: 000f5568

Name: AppDomainTest.exe

Assembly: 00109de0 [C:\Windows\Microsoft.Net\assembly\GAC\_32\mscorlib\v4.0\_4.0.0.0\_\_b77a5c561934e089\mscorlib.dll]

ClassLoader: 00110f68

SecurityDescriptor: 001097b0

Module Name

58631000 C:\Windows\Microsoft.Net\assembly\GAC\_32\mscorlib\v4.0\_4.0.0.0\_\_b77a5c561934e089\mscorlib.dll

Assembly: 0011d448 [E:\Test\AppDomainTest\AppDomainTest\bin\Debug\AppDomainTest.exe]

ClassLoader: 00117fd0

SecurityDescriptor: 0011d3c0

Module Name

001c2e9c E:\Test\AppDomainTest\AppDomainTest\bin\Debug\AppDomainTest.exe

Assembly: 00131370 [C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Windows.Forms\v4.0\_4.0.0.0\_\_b77a5c561934e089\System.Windows.Forms.dll]

ClassLoader: 0011fa00

SecurityDescriptor: 001299a0

Module Name

579c1000 C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Windows.Forms\v4.0\_4.0.0.0\_\_b77a5c561934e089\System.Windows.Forms.dll

Assembly: 00131400 [C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Drawing\v4.0\_4.0.0.0\_\_b03f5f7f11d50a3a\System.Drawing.dll]

ClassLoader: 00131490

SecurityDescriptor: 0012e9c0

Module Name

62661000 C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Drawing\v4.0\_4.0.0.0\_\_b03f5f7f11d50a3a\System.Drawing.dll

Assembly: 00131d20 [C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System\v4.0\_4.0.0.0\_\_b77a5c561934e089\System.dll]

ClassLoader: 00133d08

SecurityDescriptor: 0012f078

Module Name

5aa81000 C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System\v4.0\_4.0.0.0\_\_b77a5c561934e089\System.dll

Assembly: 00131ed0 [C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Configuration\v4.0\_4.0.0.0\_\_b03f5f7f11d50a3a\System.Configuration.dll]

ClassLoader: 001415a8

SecurityDescriptor: 0012f430

Module Name

5a981000 C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Configuration\v4.0\_4.0.0.0\_\_b03f5f7f11d50a3a\System.Configuration.dll

Assembly: 00132080 [C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Xml\v4.0\_4.0.0.0\_\_b77a5c561934e089\System.Xml.dll]

ClassLoader: 00141620

SecurityDescriptor: 0012f5c8

Module Name

546e1000 C:\Windows\Microsoft.Net\assembly\GAC\_MSIL\System.Xml\v4.0\_4.0.0.0\_\_b77a5c561934e089\System.Xml.dll

Assembly: 00132ce0 [E:\Test\AppDomainTest\AppDomainTest\bin\Debug\CrossDomainController.dll]

ClassLoader: 001b3450

SecurityDescriptor: 06f94560

Module Name

001c7428 E:\Test\AppDomainTest\AppDomainTest\bin\Debug\CrossDomainController.dll

Assembly: 00132350 [C:\Users\kong\AppData\Local\assembly\dl3\6ZYK3XE9.86Q\2AQ35O7C.VHE\1f704bbb\b7cca5cf\_8c4fcc01\ShowHelloPlug.DLL]

ClassLoader: 001b32e8

SecurityDescriptor: 070a8620

Module Name

001c7d78 C:\Users\kong\AppData\Local\assembly\dl3\6ZYK3XE9.86Q\2AQ35O7C.VHE\1f704bbb\b7cca5cf\_8c4fcc01\ShowHelloPlug.DLL

--------------------------------------

Domain 2: 06fd0238

LowFrequencyHeap: 06fd05b4

HighFrequencyHeap: 06fd0600

StubHeap: 06fd064c

Stage: OPEN

SecurityDescriptor: 06724510

Name: ApplicationLoaderDomain

Assembly: 00109de0 [C:\Windows\Microsoft.Net\assembly\GAC\_32\mscorlib\v4.0\_4.0.0.0\_\_b77a5c561934e089\mscorlib.dll]

ClassLoader: 00110f68

SecurityDescriptor: 06f93bd0

Module Name

58631000 C:\Windows\Microsoft.Net\assembly\GAC\_32\mscorlib\v4.0\_4.0.0.0\_\_b77a5c561934e089\mscorlib.dll

Assembly: 00132e90 [E:\Test\AppDomainTest\AppDomainTest\bin\Debug\ApplicationLoader\assembly\dl3\c91a2898\f6f7f865\_9a4fcc01\CrossDomainController.DLL]

ClassLoader: 001b3540

SecurityDescriptor: 06f92be0

Module Name

00a833c4 E:\Test\AppDomainTest\AppDomainTest\bin\Debug\ApplicationLoader\assembly\dl3\c91a2898\f6f7f865\_9a4fcc01\CrossDomainController.DLL

Assembly: 001330d0 [E:\Test\AppDomainTest\AppDomainTest\bin\Debug\ApplicationLoader\assembly\dl3\32519346\b7cca5cf\_8c4fcc01\ShowHelloPlug.DLL]

ClassLoader: 001b39f0

SecurityDescriptor: 06f92f98

Module Name

00a83adc E:\Test\AppDomainTest\AppDomainTest\bin\Debug\ApplicationLoader\assembly\dl3\32519346\b7cca5cf\_8c4fcc01\ShowHelloPlug.DLL

除了新建的AppDomain(Domain2)中的Module引用了ShowHelloPlug.dll,默认域(Domian1)也有ShowHelloPlug.dll的

程序集引用.

# 应用程序域之间的通信

每个AppDomain都有自己的堆栈,内存块,也就是说它们之间的数据并非共享了.若想共享数据,则涉及到应用程序域之间的通信.C#提供了[MarshalByRefObject类](http://msdn.microsoft.com/library/ZH-CN/SYSTEM.MARSHALBYREFOBJECT(VS.100))进行跨域通信,那么,我们必须提供自己的跨域访问器.

public class RemoteLoader : MarshalByRefObject

{

private Assembly \_assembly;

public void LoadAssembly(string assemblyFile)

{

try

{

\_assembly = Assembly.LoadFrom(assemblyFile);

//return \_assembly;

}

catch (Exception ex)

{

throw ex;

}

}

public T GetInstance<T>(string typeName) where T : class

{

if (\_assembly == null) return null;

var type = \_assembly.GetType(typeName);

if (type == null) return null;

return Activator.CreateInstance(type) as T;

}

public void ExecuteMothod(string typeName, string methodName)

{

if (\_assembly == null) return;

var type = \_assembly.GetType(typeName);

var obj = Activator.CreateInstance(type);

Expression<Action> lambda = Expression.Lambda<Action>(Expression.Call(Expression.Constant(obj), type.GetMethod(methodName)), null);

lambda.Compile()();

}

}

为了更好的操作这个跨域访问器,接下来我构建了一个名为AssemblyDynamicLoader的类,它内部封装了RemoteLoader类

的操作.

public class AssemblyDynamicLoader

{

private AppDomain appDomain;

private RemoteLoader remoteLoader;

public AssemblyDynamicLoader()

{

AppDomainSetup setup = new AppDomainSetup();

setup.ApplicationName = "ApplicationLoader";

setup.ApplicationBase = AppDomain.CurrentDomain.BaseDirectory;

setup.PrivateBinPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "private");

setup.CachePath = setup.ApplicationBase;

setup.ShadowCopyFiles = "true";

setup.ShadowCopyDirectories = setup.ApplicationBase;

AppDomain.CurrentDomain.SetShadowCopyFiles();

this.appDomain = AppDomain.CreateDomain("ApplicationLoaderDomain", null, setup);

String name = Assembly.GetExecutingAssembly().GetName().FullName;

this.remoteLoader = (RemoteLoader)this.appDomain.CreateInstanceAndUnwrap(name, typeof(RemoteLoader).FullName);

}

public void LoadAssembly(string assemblyFile)

{

remoteLoader.LoadAssembly(assemblyFile);

}

public T GetInstance<T>(string typeName) where T : class

{

if (remoteLoader == null) return null;

return remoteLoader.GetInstance<T>(typeName);

}

public void ExecuteMothod(string typeName, string methodName)

{

remoteLoader.ExecuteMothod(typeName, methodName);

}

public void Unload()

{

try

{

if (appDomain == null) return;

AppDomain.Unload(this.appDomain);

this.appDomain = null;

}

catch (CannotUnloadAppDomainException ex)

{

throw ex;

}

}

}

这样我们每次都是通过AssemblyDynamicLoader类进行跨域的访问.

AppDomain.CurrentDomain.SetShadowCopyFiles();

this.appDomain = AppDomain.CreateDomain("ApplicationLoaderDomain", null, setup);

String name = Assembly.GetExecutingAssembly().GetName().FullName;

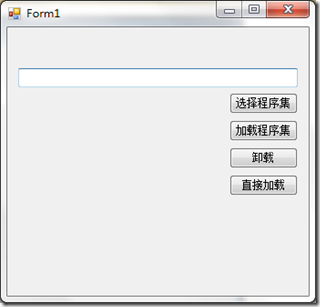
this.remoteLoader = (RemoteLoader)this.appDomain.CreateInstanceAndUnwrap(name, typeof(RemoteLoader).FullName);

通过我们前面构造的一个AppDomainSetup,构建了一个我们所需的AppDomain,并且在这个appDomain中构建了

一个RemoteLoader类的实例(此时该实例已具备跨域访问能力,也就是说我们在主域能获取子域内部的数据信息).目前RemoteLoader只提供了少数的几个方法.

# 跨域操作

下面,我们就模拟一次"插件式"的跨域操作.首先我们构造了一个窗体,其有以下元素.

[](http://images.cnblogs.com/cnblogs_com/kongyiyun/201108/201108010957436116.png)

选择程序集路径之后,加载程序集,然后就触发程序集指定类型(通过配置获取)的特定操作.这里我们定义了一个公共接口,它是所有"插件"操作的主要入口了.

public interface IPlug

{

void Run();

}

随后定义了一个实现该接口的类.

[Serializable]

public class ShowHelloPlug : IPlug

{

public void Run()

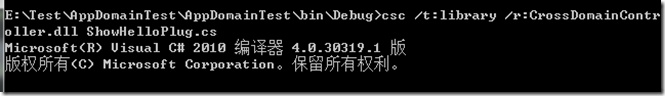
{

MessageBox.Show("Hello World...");

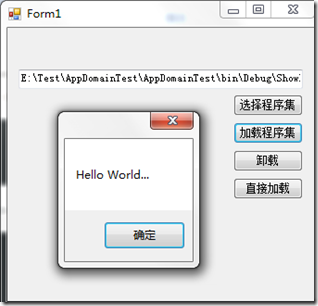
}

}

这个"插件"的工作很简单.仅仅弹出一个对话框,说声"Hello World…",接下来将其编译成一个dll.

[](http://images.cnblogs.com/cnblogs_com/kongyiyun/201108/201108010957459298.png)

回到界面,选择刚才编译的Dll,然后直接加载.

[](http://images.cnblogs.com/cnblogs_com/kongyiyun/201108/201108010957489524.png)

到这里,我们的工作完成了一半了.呼呼.OK.我们的需求发生了变化,不再是弹出Hello World了.而时候弹出Hi,I'm Kinsen,我们修改刚才的子类,并再编译一次.再将Dll替换刚才的Dll,这次,Dll没有没锁定(因为我们前面启用了ShadowCopyFiles.).再加载一下程序集,你会发现结果并不是"Hi,I'm Kinsen",而是"Hello World.."为什么会这样呢?这时候,借助SOS的力量(前面有SOS结果).

我们发现Domain1(Default Domain)和Domain2(新创建Domain)都引用了程序集ShowHelloPlug.DLL,但是两个引用的Dll地址却不相同,这是因为启用了ShadowCopyFiles,它们加载的都是各自程序集的备份,我们根据Domain2的Assembly地址查看ShowHelloPlug的编译代码.

0:011> !dumpmt 00fc40ac

00fc40ac is not a MethodTable

0:011> !dumpmd 00fc40ac

Method Name: Plug.ShowHelloPlug.Run()

Class: 046812b4

MethodTable: 00fc40bc

mdToken: 06000001

Module: 00fc3adc

IsJitted: no

CodeAddr: ffffffff

Transparency: Critical

从IsJitted为no可以看出,该程序集并没有被调用,那调用的是谁?我们再次查看Domain1(Default Domain

)中的ShowHelloPlug.

0:011> !dumpmd 001f8240

Method Name: Plug.ShowHelloPlug.Run()

Class: 004446e4

MethodTable: 001f8250

mdToken: 06000001

Module: 001f7d78

IsJitted: yes

CodeAddr: 00430de0

Transparency: Critical

已知每个AppDomain都有自己的堆栈信息,各自不互相影响,所以,当我们在主域中获取到了子域中的数据,并非新建一个指向该实例的引用,而是在自己的堆栈上开辟出一块空间"深度拷贝"该实例,那么必然就达不到我们我需的结果.

# 子域内部调用

那么为了达到我们预期的效果,我们必须在子域内部执行我们所需的操作(调用),所以在RemoteLoader类中增加了一个Execute方法

public void ExecuteMothod(string typeName, string methodName)

{

if (\_assembly == null) return;

var type = \_assembly.GetType(typeName);

var obj = Activator.CreateInstance(type);

Expression<Action> lambda = Expression.Lambda<Action>(Expression.Call(Expression.Constant(obj), type.GetMethod(methodName)), null);

lambda.Compile()();

}

此处我暂时只想到了利用反射调用,这样的代价就是调用所需消耗的资源更多,效率低下.目前还没有

想出较好的解决方案,有经验的童鞋欢迎交流.

这样外部的调用就变成以下

loader = new AssemblyDynamicLoader();

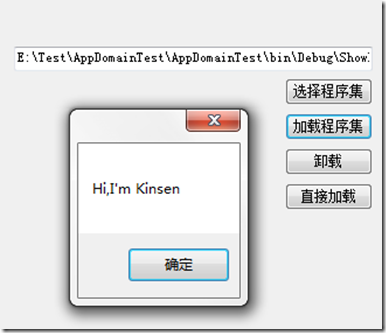
loader.LoadAssembly(txt\_dllName.Text);

//var obj = loader.GetInstance<IPlug>("Plug.ShowHelloPlug");

//obj.Run();

loader.ExecuteMothod("Plug.ShowHelloPlug", "Run");

现在在将Dll替换,结果正常.

[](http://images.cnblogs.com/cnblogs_com/kongyiyun/201108/201108010957514244.png)

# 尾声

做"插件式"开发,除了利用AppDomain之外,也有童鞋给出了另一种解决方案,也就是在加载Dll的时候,先将Dll在内存中复制一份,这样原来的Dll也就不会被锁定了.详见[插件的“动态替换”](http://www.cnblogs.com/zhuweisky/archive/2005/12/30/308218.html).

以上实例本人皆做过实验,但可能还存在一定不足或概念错误,若有不当之处,欢迎各位童鞋批评指点.

# 更多

[通过应用程序域AppDomain加载和卸载程序集](http://www.cnblogs.com/wayfarer/archive/2004/09/29/47896.html)

[什么是的AppDomain](http://zh-cn.w3support.net/index.php?db=so&id=574708)