# 第50章 MAF

本章内容如下：

- MAF 的体系结构
- 定义协定
- 实现管道
- 创建插件
- 保存插件

本章详细介绍 Managed Add-In Framework(MAF)的体系结构，以及如何使用 MAF 创建和驻留插件。我们将学习 MAF 的体系结构，它通过驻留插件解决什么问题，如版本问题、发现、激活和隔离。接着讨论创建 MAF 管道所需的所有步骤，MAF 管道可用于连接插件和宿主；如何创建插件；如何创建使用插件的宿主应用程序。

## 50.1 MAF 体系结构

.NET 4 包含两个用于创建和使用插件的技术。第 28 章介绍了如何使用 Managed Extensibility Framework(MEF)创建插件。使用 MEF 的应用程序在运行期间在目录或程序集中查找插件，并使用属性连接它们。MAF 与 MEF 的区别是，MEF 没有使用应用程序域或不同的进程把插件与宿主应用程序分隔开。要把插件和宿主应用程序分隔开，应使用 MAF。但为了达到这个目的，MAF 的复杂性提高了不少。如果希望利用 MEF 和 MAF 的优点，就可以合并这两个技术。当然，这也会增加复杂性。

创建允许在运行期间添加插件的应用程序时，需要处理一些问题。例如，如何找到插件，如何解决版本问题，以便宿主应用程序和插件可以独立地升级。本节讨论插件的问题和 MAF 的体系结构如何解决这些问题。

要创建一个宿主应用程序，动态加载以后添加的程序集，必须解决几个问题，如表 50-1 所示。

表 50-1

| 插件问题 | 说明 |
| --- | --- |
| 发现 | 如何为宿主应用程序查找新插件？这有几个选项。一个选项是在配置文件中添加插件的信息。其缺点是安装新插件时，需要修改已有的配置文件。另一个选项是把包含插件的程序集复制到预定义的目录中，通过反射读取程序集的信息。<br>反射的更多内容可参见第 14 章 |

(续表)

| 插 件 问 题 | 说　　明 |
| --- | --- |
| 激活 | 程序集动态加载后，还不能使用 new 运算符创建实例。但可以用 Activator 类创建这类程序集。另外，如果插件加载到另一个应用程序域中或一个新进程中，那么还需要使用不同的激活选项。程序集和应用程序域的更多内容可参见第 18 章 |
| 隔离 | 插件可能会使宿主应用程序崩溃，读者可能见过 IE 因各种插件而崩溃的情况。根据宿主应用程序的类型和插件的集成方式，插件可以加载到另一个应用程序域或另一个进程中 |
| 生命周期 | 清理对象是垃圾回收器的一个工作。但是，垃圾回收器在这里没有任何帮助，因为插件可能在另一个应用程序域中或另一个进程中激活。把对象保存在内存中的其他方式有引用计数、租借和主办机制 |
| 版本 | 版本问题是插件的一个大问题。通常宿主的一个新版本仍可以加载旧插件，而旧宿主应有加载新插件的选项 |

下面探讨 MAF 的体系结构，说明这个架构如何解决这些问题。影响 MAF 的设计目标如下：

- 应易于开发插件
- 在运行期间查找插件应很高效
- 开发宿主应用程序应是一个很简单的过程，但不像开发插件那么容易
- 插件和宿主应用程序应独立地升级

MAF 使用管道解决了这些问题，管道在其核心使用协定。我们将讨论 MAF 如何使用发现功能来查找插件，如何激活插件，如何使插件保持激活状态，如何处理版本问题。

### 50.1.1　管道

MAF 体系结构基于一个包含 7 个程序集的管道。这个管道解决了插件的版本问题。因为管道中的程序集之间的依赖性很弱，所以协定、宿主程序和插件应用程序升级到新版本可以完全互不干扰。

图 50-1 显示了 MAF 体系结构的管道。其中心是协定程序集。这个程序集包含一个协定接口，其中列出了插件必须实现并且可以由宿主调用的方法和属性。协定的左边是宿主端，右边是插件端。图 50-1 还显示了程序集之间的依赖关系。最左端的宿主程序集与协定程序集没有依赖关系，插件程序集与协定程序集也没有依赖关系，这两个程序集实际上都没有实现协定定义的接口，只是有一个对视图程序集的引用。宿主应用程序引用宿主视图；插件引用插件视图。视图包含抽象的视图类，该类定义的方法和属性与协定相同。

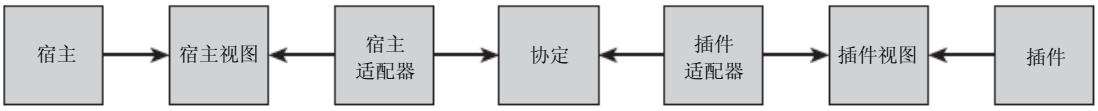


图　50-1

图 50-2 显示了管道中类的关系。宿主类与抽象的宿主视图类有一个关联，并调用其方法。抽象的宿主视图类由宿主适配器实现。适配器在视图和协定之间建立连接。插件适配器实现协定的方法和属性。这个适配器包含对插件视图的引用，把来自宿主端的调用转发给插件视图。宿主适配器类定义了一个具体的类，它派生自宿主视图的抽象基类，用于实现方法和属性。这个适配器包含对协定的一个引用，用于把来自视图的调用转发给协定。

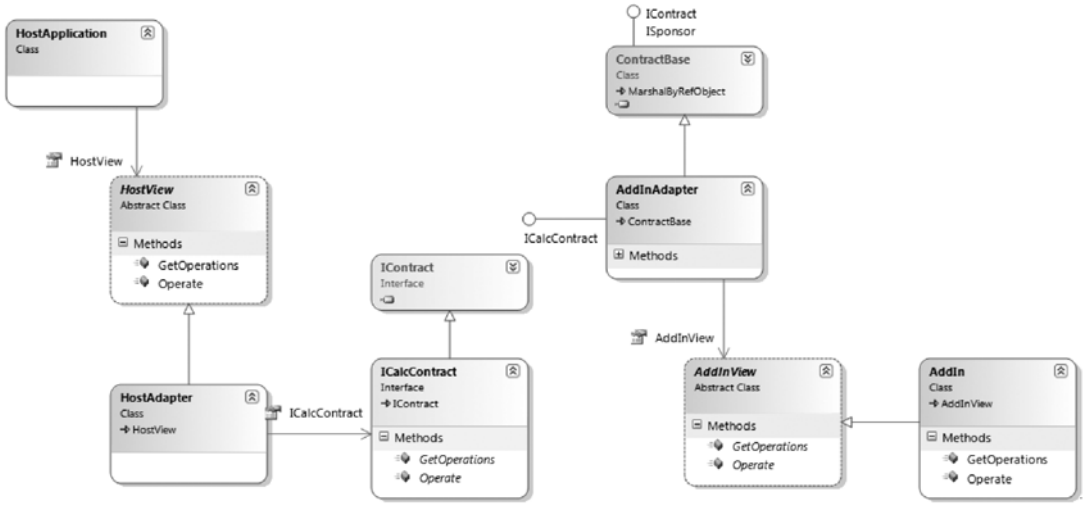


图　50-2

通过这个模型，插件端和宿主端可以完全独立地升级，只是需要调整映射层。例如，如果宿主的一个新版本使用全新的方法和属性，协定就仍可以保持不变，只有适配器需要更改。也可以定义新的协定。适配器可以更改，也可以同时使用几个协定。

### 50.1.2　发现

如何为宿主应用程序查找新插件？MAF 的体系结构使用一个预定义的目录结构来查找插件和管道的其他程序集。管道的组成部分必须存储在这些子目录中：

- HostSideAdapters
- Contracts
- AddInSideAdapters
- AddInViews
- AddIns

除了 AddIns 目录之外，其他目录都直接包含管道特定部分的程序集。AddIns 目录为每个插件程序集包含一个子目录。插件也可以存储在完全独立于其他管道组件的目录中。

管道的程序集需要使用反射来动态地加载，才能获得插件的所有信息。而且，对于许多插件，这还会增加宿主应用程序的启动时间。因此，MAF 使用一个缓存，来保存管道组件的信息。该缓存由安装插件的程序创建，如果宿主应用程序有管道目录的写入权限，该缓存就由宿主应用程序或安装插件的应用程序创建。

调用 AddInStore 类的方法来创建管道组件的缓存信息。Update()方法查找还没有列在存储文件中的新插件。Rebuild()方法用插件的信息重新生成整个二进制存储文件。

表 50-2 列出了 AddInStore 类的成员。

表　50-2

| AddInStore 类的成员 | 说　明 |
| --- | --- |
| Rebuild()、RebuildAddIns() | Rebuild()方法为管道的所有组件重新生成缓存。如果插件存储在另一个目录下，就可以使用 RebuildAddIns()方法重新生成插件的缓存 |

(续表)

| AddInStore 成员 | 说　明 |
| --- | --- |
| Update()、UpdateAddIns() | Rebuild()方法重建管道的完整缓存，Update()方法只用新管道组件的信息更新缓存。UpdateAddIns()方法只更新插件的缓存 |
| FindAddIn()、FindAddIns() | 这些方法都使用缓存查找插件。FindAddIns()方法返回匹配宿主视图的所有插件集合。FindAddIn()方法返回一个特定的插件 |

### 50.1.3　激活和隔离

AddInStore 类的 FindAddIns()方法返回表示插件的 AddInToken 对象集合。使用 AddInToken 类可以访问插件的信息，如名称、描述、发布者和版本。使用 Activate()方法可以激活插件。表 50-3 列出了 AddInToken 类的特性和方法。

表　50-3

| AddInToken 类的成员 | 说　明 |
| --- | --- |
| Name、Publisher、Version、Description | AddInToken 类的 Name、Publisher、Version 和 Description 属性返回用 AddInAttribute 特性赋予插件的信息 |
| AssemblyName | AssemblyName 返回包含插件的程序集的名称 |
| EnableDirectConnect | 使用 EnableDirectConnect 属性可以设置一个值，指出宿主应直接连接到插件上，而不使用管道的组件。只有插件和宿主运行在同一个应用程序域中，插件视图和宿主视图的类型相同时，才能使用这个属性。该属性仍要求管道的所有组件都存在 |
| QualificationData | 插件可以用 QualificationDataAttribute 特性标记应用程序域和安全需求。插件可以列出安全需求和隔离需求。例如，[QualificationData (“Isolation”, ”NewAppDomain”)]表示插件必须驻留在新进程中。可以从 AddInToken 类中读取这些信息，激活有特定需求的插件。除了应用程序域和安全需求之外，还可以使用这个属性通过管道传递自定义信息 |
| Activate() | 插件用 Activate()方法激活，利用这个方法的参数，可以定义插件是否加载到新应用程序域或新进程中。还可以定义插件获得的权限 |

一个插件可能使整个应用程序崩溃。例如，IE 可能因一个失败的插件而崩溃。根据应用程序类型和插件类型，可以让插件运行在另一个应用程序域或另一个进程中，来避免这个问题。这里 MAF 给出了几个选项。可以在新应用程序域或新进程中激活插件。新应用程序域还可以有有限的权限。

AddInToken 类的 Activate()方法有几个重载版本，在这些版本中，可以传递应加载的插件的环境参数。表 50-4 列出了不同的选项。

表　50-4

| AddInToken.Activate()方法的参数 | 说　明 |
| --- | --- |
| AppDomain | 可以传递应加载的插件的一个新应用程序域，这样可以使插件独立于宿主应用程序，还可以从应用程序域中卸载插件 |
| AddInSecurityLevel | 如果插件应使用不同的安全级别来运行，就传递 AddInSecurityLevel 枚举的一个值，其值可以是 Internet、Intranet、FullTrust 和 Host |

(续表)

| AddInToken.Activate()方法的参数 | 说　明 |
| --- | --- |
| PermissionSet | 如果预定义的安全级别不够专用，那么还可以给插件的应用程序域赋予 PermissionSet |
| AddInProcess | 插件还可以运行在与宿主应用程序不同的进程中。可以给 Activate()方法传递一个新的 AddInProcess。如果卸载所有插件，新进程就可以关闭；否则新进程继续运行。这个选项可以用 KeepAlive 属性设置 |
| AddInEnvironment | 传递 AddInEnvironment 对象是定义在哪里加载插件的应用程序域的另一个选项。在 AddInEnvironment 的构造函数中，可以传递一个 AppDomain 对象。还可以用 AddInController 类的 AddInEnvironment 属性获得插件的已有 AddInEnvironment |

应用程序域详见第 18 章。

应用程序的类型也会限制可以使用的选项。WPF 插件目前不支持跨进程。Windows 窗体不能在不同的应用程序域之间连接 Windows 控件。

下面列出调用 AddInToken 类的 Activate()方法时管道执行的步骤：

(1) 用指定的权限创建应用程序域。

(2) 用 Assembly.LoadFrom()方法把插件的程序集加载到新的应用程序域中。

(3) 用反射调用插件的默认构造函数。因为插件派生自在插件视图中定义的基类，所以也加载视图的程序集。

(4) 构造插件端适配器的一个实例。因为把插件的这个实例传递给适配器的构造函数，所以适配器能连接协定和插件。因为插件适配器派生自基类 MarshalByRefObject，所以可以在应用程序域之间调用它。

(5) 激活代码给宿主应用程序的应用程序域返回插件端适配器的一个代理。因为插件适配器实现协定接口，所以该代理包含协定接口的方法和属性。

(6) 宿主端适配器的实例在宿主应用程序的应用程序域中构造。把插件端适配器的代理传递给该构造函数。激活代码会从插件令牌中查找宿主端适配器的类型。

宿主端适配器返回宿主应用程序。

### 50.1.4　协定

协定定义 MAF 体系结构中宿主端和插件端之间的边界。协定用一个接口来定义，该接口必须派生自基接口 IContract。协定必须仔细考虑，因为它根据需要支持灵活的插件场景。

因为协定没有版本冲突，不能改变，以便插件以前的实现代码仍可以在新的宿主程序中运行。新版本应通过定义新协定来创建。

可以使用的协定类型有一些限制，其原因是版本问题，而且应用程序域要从宿主应用程序跨越到插件上。类型必须是安全的，且没有版本冲突，能在边界(应用程序域或跨进程)之间传递，也能在宿主和插件之间传递。

可以用协定传递的类型可以是：

- 基元类型

- 其他协定
- 可序列化的系统类型
- 简单的可序列化的自定义类型，包括基本类型、协定，以及没有实现代码的类型

IContract 接口的成员如表 50-5 所示。

表 50-5

| Icontract 接口的成员 | 说　明 |
|---|---|
| QueryContract() | 使用 QueryContract()方法可以查询协定，验证它是否也实现了另一个协定。一个插件可以支持几个协定 |
| RemoteToString() | QueryContract()方法的参数需要协定的字符串表示。RemoteToString()方法返回当前协定的字符串表示 |
| AcquireLifetimeToken()<br>RevokeLifetimeToken() | 客户端调用 AcquireLifetimeToken()方法来保存对协定的引用。AcquireLifetimeToken()方法会递增引用计数。RevokeLifetimeToken()方法递减引用计数 |
| RemoteEquals() | RemoteEquals()方法可用于比较两个协定引用 |

协定接口在 System.AddIn.Contract、System.AddIn.Contract.Collections 和 System.AddIn. Contract.-Automation 名称空间中定义。表 50-6 列出了可以用于协定的协定接口。

表 50-6

| 协　定 | 说　明 |
|---|---|
| IListContract<T> | IListContract<T>可用于返回一个协定列表 |
| IEnumeratorContract<T> | IEnumeratorContract<T>用于枚举 IListContract<T>的元素 |
| IServiceProviderContract | 一个插件可以为其他插件提供服务。提供服务并实现 IserviceProviderContract 接口的插件称为服务提供程序。通过 QueryService()方法，可以查询实现该接口的插件提供什么服务 |
| IProfferServiceContract | IProfferServiceContract 是服务提供程序和 IserviceProviderContract 协定提供的接口。IprofferServiceContract 协定定义 ProfferService()和 Revoke Service()方法。ProfferService()方法给所提供的服务添加一个 IserviceProviderContract 协定，而 RevokeService()方法删除它 |
| INativeHandleContract | 这个接口允许使用 GetHandle()方法访问本地 Windows 句柄。这个协定由 WPF 宿主程序用于使用 WPF 插件 |

### 50.1.5 生命周期

插件需要加载多长时间？它使用多少时间？何时可以卸载应用程序域？解决上述问题有几个选项。一个选项是使用引用计数。每次使用插件都会递增引用计数。如果引用计数递减到 0，就可以卸载插件。另一个选项是使用垃圾回收器。如果垃圾回收器在运行，且没有再引用对象，该对象就是垃圾回收器的目标。.NET 远程处理使用租约机制，和使对象保持激活状态的主办方。只要租期到了，就询问主办方该对象是否应继续保持激活状态。

应用程序域详见第 18 章。

对于插件，卸载插件还有一个特殊的问题，因为插件运行在不同的应用程序域中和不同的进程中。但垃圾回收器不能跨不同进程工作。MAF 使用一个混合的模型来管理生命周期。在单个应用程序域中，使用垃圾回收机制。在管道内部，虽然使用一个隐式的承办机制，但引用计数可用于从外部控制主办方。

下面考虑一种情况：把插件加载到另一个应用程序域中。在宿主应用程序中，当不再需要引用时，垃圾回收器清理了宿主视图和宿主端适配器。而在插件端，协定定义 AcquireLifetimeToken()和 RevokeLifetimeToken()方法，来递增和递减主办方的引用计数。这两个版本不仅递增和递减一个值，还可以在某一方频繁调用 RevokeLifetimeToken()方法时，提早释放对象。而 AcquireLifetimeToken()方法返回一个表示生命周期令牌的标识符，这个标识符必须用于调用 RevokeLifetimeToken()方法。所以这两个方法总是成对调用。

通常不必处理 AcquireLifetimeToken()和 RevokeLifetimeToken()方法的调用。然而，可以使用 ContractHandle 类，在构造函数中调用 AcquireLifetimeToken()方法，在终结器中调用 RevokeLifetimeToken()方法。

终结器详见第 13 章。

在把插件加载到新应用程序域的情形中，当不再需要插件时，可以删除加载的代码。如果不再需要这个插件，MAF 就使用一个简单的模型把一个插件指定为应用程序域的拥有者，来卸载应用程序域。如果在激活插件时创建应用程序域，这个插件就是应用程序域的拥有者。如果应用程序域是以前创建的，就不会自动卸载它。

在宿主端适配器中 ContractHandle 类用来增加插件的引用计数。这个类的成员如表 50-7 所示。

表　50-7

| ContractHandle 类的成员 | 说　明 |
|---|---|
| Contract | 在 ContractHandle 类的构造过程中，可以指定一个实现 Icontract 接口的对象，来保存对它的引用。Contract 属性返回这个对象 |
| Dispose() | 可以调用 Dispose()方法，而不是等待垃圾回收器执行终结操作，来调用生命周期令牌 |
| AppDomainOwner() | AppDomainOwner()是 ContractHandle 类的一个静态方法，如果插件拥有通过该方法传递的应用程序域，该方法就返回插件适配器 |
| ContractOwnsAppDomain() | 使用静态方法 ContractOwnsAppDomain()，可以验证指定的协定是否是应用程序域的拥有者。如果是，在删除协定时，就会卸载应用程序域 |

### 50.1.6　版本问题

版本问题是插件的一个大问题。宿主应用程序可以利用插件进一步开发。插件的一个要求是宿主应用程序的新版本仍可以加载插件的旧版本。旧宿主程序仍可以运行插件的新版本。那么，协定该如何修改呢？

System.AddIn 完全独立于宿主应用程序和插件的实现，这通过包含 7 部分的管道概念实现。

## 50.2 插件示例

下面是一个宿主应用程序的简单示例，它可以加载计算器插件。插件支持其他插件提供的不同计算操作。

我们需要创建一个解决方案，它包含 6 个库项目和一个控制台应用程序。示例应用程序的项目如表 50-8 所示。这个表列出了需要引用的程序集。在解决方案中引用其他项目，还需要把 Copy Local 属性设置为 False，这样程序集就不会复制。但 HostApp 控制台项目例外，它需要引用 HostView 项目。必须复制这个程序集，这样才能在宿主应用程序中找到它。另外，还需要更改所生成的程序集的输出路径，以便程序集复制到管道的正确目录下。

表 50-8

| 项　目 | 引　用 | 输出路径 | 说　明 |
|---|---|---|---|
| CalcContract | System.AddIn.Contract | .\Pipeline\Contracts\ | 这个程序集包含与插件通信的协定。协定用接口定义 |
| CalcView | System.AddIn | .\Pipeline\AddInViews\ | CalcView 程序集包含插件引用的一个抽象类，这是协定的插件端 |
| CalcAddIn | System.AddIn.CalcView | .\Pipeline\AddIns\CalcAddIn\ | CalcAddIn 是引用插件视图程序集的插件项目。这个程序集包含插件的实现代码 |
| CalcAddInAdapter | System.AddIn、System.AddIn.Contract CalcView、CalcContract、 | .\Pipeline\AddInSideAdapters\ | CalcAddInAdapter 连接插件视图和协定程序集，并把协定映射到插件视图上 |
| HostView | | | 包含宿主视图的抽象类的程序集不需要引用任何插件程序集，也不引用解决方案中的其他项目 |
| HostAdapter | System.AddIn、System.AddIn.Contract、HostView、CalcContract、 | .\Pipeline\HostSideAdapters\ | 宿主适配器把宿主视图映射到协定上。因此需要引用这些项目 |
| HostApp | System.AddIn、HostView | | 宿主应用程序激活插件 |

### 50.2.1 插件协定

下面实现协定程序集。协定程序集包含一个协定接口，该接口定义了在宿主和插件之间的通信协议。

下面的代码是为计算器示例应用程序定义的协定。应用程序给协定定义 GetOperations()和 Operate()方法。GetOperations()方法返回计算器插件支持的一个数学操作列表。数学操作由

IOperationContract 接口定义，IOperationContract 接口本身就是一个协定。IOperationContract 接口定义只读特性 Name 和 NumberOperands。

Operate()方法调用插件中的操作，它需要 IOperation 接口定义的一个操作和通过 double 数组提供的操作数。通过这个接口，插件就可以支持需要任意多个 double 操作数的操作，且返回一个 double。AddInStore 类使用 AddInContract 属性生成缓存。AddInContract 属性把类标记为插件协定接口。

这里使用的协定的功能与第 28 章中插件的协定类似。但 MEF 协定没有 MAF 协定的限制。MAF 协定的限制很严，因为宿主和插件之间有应用程序域边界和进程边界。而 MEF 体系结构没有使用不同的应用程序域。

可从 wrox.com 下载源代码

```
using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
   [AddInContract]
   public interface ICalculatorContract: IContract
   {
      IListContract<IOperationContract> GetOperations();
      double Operate(IOperationContract operation, double[] operands);
   }
   public interface IOperationContract: IContract
   {
      string Name { get; }
      int NumberOperands { get; }
   }
}
```

代码段 CalcContract/ICalculatorContract.cs

## 50.2.2　计算器插件视图

插件视图重新定义了插件可以识别的协定。该协定定义 ICalculatorContract 和 IOperationContract 接口。为此，插件视图定义了 Calculator 抽象类和 Operation 具体类。

在 Operation 中，没有每个插件都需要的特定实现代码，因为该类已经用插件视图程序集实现了。这个类用 Name 和 NumberOperands 属性描述数学计算的一个操作。

Calculator 抽象类定义需要由插件实现的方法。虽然协定定义了需要在应用程序域边界和进程边界之间传递的参数和返回类型，但插件视图不需要。这里可以使用类型，以便于插件开发人员编写插件。GetOperations()方法返回 IList<Operation>，而不是 IListOperation<IOperationContract>，这与协定程序集不同。

AddInBase 属性把类标识为插件视图，便于存储。

可从 wrox.com 下载源代码

```
using System.AddIn.Pipeline;
using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
   [AddInBase]
```

```
    public abstract class Calculator
    {
        public abstract IList<Operation> GetOperations();
        public abstract double Operate(Operation operation, double[] operand);
    }
    public class Operation
    {
        public string Name { get; set; }
        public int NumberOperands { get; set; }
    }
}
```

代码段 CalcView/CalculatorView.cs

### 50.2.3 计算器插件适配器

插件适配器把协定映射到插件视图上。这个程序集引用协定和插件视图程序集。适配器的实现代码需要把协定中的IListContract<IOperationContract>集合的GetOperations()方法映射到IList<Operation>集合的 GetOperations()视图方法上。

该程序集包含OperationViewToContractAddInAdapter类和CalculatorViewToContractAddInAdapter类。这两个类实现 IOperationContract 和 ICalculatorContract 接口。IContract 基接口的方法可以通过派生自ContractBase 基类来实现。这个基类提供默认的实现代码。OperationViewToContractAddInAdapter 类实现IOperationContract 接口的其他成员，并把调用转发给在构造函数中指定的 Operation View。

OperationViewToContractAddInAdapter 类还包含静态辅助方法 ViewToContractAdapter() 和ContractToViewAdapter()，前者把 Operation 映射到 IOperationContract 上，后者把 IOperationContract 映射到 Operation 上。


可从
wrox.com
下载源代码

```
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    internal class OperationViewToContractAddInAdapter: ContractBase,
          IOperationContract
    {
        private Operation view;
        public OperationViewToContractAddInAdapter(Operation view)
        {
            this.view = view;
        }
        public string Name
        {
            get { return view.Name; }
        }
        public int NumberOperands
        {
            get { return view.NumberOperands; }
        }
        public static IOperationContract ViewToContractAdapter(Operation view)
        {
            return new OperationViewToContractAddInAdapter(view);
        }
        public static Operation ContractToViewAdapter(
             IOperationContract contract)
```

```
        {
            return (contract as OperationViewToContractAddInAdapter).view;
        }
    }
}
```

代码段 CalcAddInAdapter/OperationViewToContractAddInAdapter.cs

CalculatorViewToContractAddInAdapter 类非常类似于 OperationViewToContractAddInAdapter 类：它派生自 ContractBase 基类，来继承 IContract 接口的默认实现代码，它还实现一个协定接口。但这里的 ICalculatorContract 接口用 GetOperations()和 Operate()方法实现。

适配器的 Operate()方法调用 Calculator 视图类的 Operate()方法，其中 IOperationContract 需要转换为 Operation。这是使用 OperationViewToContractAddInAdapter 类定义的静态辅助方法 ContractViewToAdapter()完成。

GetOperations()方法的实现需要把 IListContract<IOperationContract>集合转换为 IList<Operation>集合。对于这个集合转换，CollectionAdapters 类定义了转换方法 ToIList()和 ToIListContract()。其中 ToIListContract()方法用于这个转换。

AddInAdapter 属性把类标识为插件端适配器，用于插件存储器。


```
using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    [AddInAdapter]
    internal class CalculatorViewToContractAddInAdapter: ContractBase,
          ICalculatorContract
    {
        private Calculator view;
        public CalculatorViewToContractAddInAdapter(Calculator view)
        {
            this.view = view;
        }
        public IListContract<IOperationContract>GetOperations()
        {
            return CollectionAdapters.ToIListContract<Operation,
                 IOperationContract>(view.GetOperations(),
                 OperationViewToContractAddInAdapter.ViewToContractAdapter,
                 OperationViewToContractAddInAdapter.ContractToViewAdapter);
        }
        public double Operate(IOperationContract operation, double[] operands)
        {
            return view.Operate(
                 OperationViewToContractAddInAdapter.ContractToViewAdapter(
                      operation), operands);
        }
    }
}
```

代码段 CalcAddInAdapter/OperationViewToContractAddInAdapter.cs

因为适配器类由.NET 反射调用，所以这些类可以使用内部的访问修饰符。因为这些类要具体实现，所以最好使用 internal 访问修饰符。

### 50.2.4 计算器插件

插件现在包含具体功能的实现代码。它用 CalculatorV1 类实现。插件程序集依赖于插件视图程序集，因为它需要实现 Calculator 抽象类。

AddIn 属性把类标记为插件，用于插件存储器，并添加发布者、版本和描述信息。在宿主端，这些信息可以从 AddInToken 类中访问。

CalculatorV1 在 GetOperations()方法中返回一个支持的操作列表。Operate()方法根据操作计算操作数。

可从 wrox.com 下载源代码

```
using System;
using System.AddIn;
using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
   [AddIn("CalculatorAddIn", Publisher="Wrox Press", Version="1.0.0.0",
         Description="Sample AddIn")]
   public class CalculatorV1: Calculator
   {
      private List<Operation> operations;
      public CalculatorV1()
      {
         operations = new List<Operation>();
         operations.Add(new Operation() { Name = "+", NumberOperands = 2 });
         operations.Add(new Operation() { Name = "-", NumberOperands = 2 });
         operations.Add(new Operation() { Name = "/", NumberOperands = 2 });
         operations.Add(new Operation() { Name = "*", NumberOperands = 2 });
      }
      public override IList<Operation> GetOperations()
      {
         return operations;
      }
      public override double Operate(Operation operation, double[] operand)
      {
         switch (operation.Name)
         {
            case "+":
               return operand[0] + operand[1];
            case " - ":
               return operand[0] - operand[1];
            case "/":
               return operand[0] / operand[1];
            case "*":
               return operand[0] * operand[1];
            default:
               throw new InvalidOperationException(
                     String.Format("invalid operation {0}", operation.Name));
         }
```

```
        }
    }
}
```

代码段 CalcAddIn/Calculator.cs

### 50.2.5　计算器宿主视图

下面看看宿主端的宿主视图。与插件视图类似，宿主视图也定义一个抽象类，其方法类似于协定。但是，这里定义的方法由宿主应用程序调用。

Calculator 和 Operation 类都是抽象类，因为其成员由宿主适配器实现。这两个类只需要定义宿主应用程序使用的接口：


```
using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
    public abstract class Calculator
    {
        public abstract IList<Operation> GetOperations();
        public abstract double Operate(Operation operation,
             params double[] operand);
    }
    public abstract class Operation
    {
        public abstract string Name { get; }
        public abstract int NumberOperands { get; }
    }
}
```

代码段 HostView/CalculatorHostView.cs

### 50.2.6　计算机宿主适配器

宿主适配器程序集引用宿主视图和协定，从而把视图映射到协定上。OperationContractToViewHostAdapter 类实现 Operation 抽象类的成员。CalculatorContractToViewHostAdapter 类实现 Calculator 抽象类的成员。

在 OperationContractToViewHostAdapter 类中，在构造函数中指定对协定的引用。适配器类还包含一个 ContractHandle 实例，该实例添加了对协定生命周期的引用，这样只要宿主应用程序需要，插件就保持加载状态。


```
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    internal class OperationContractToViewHostAdapter: Operation
    {
        private ContractHandle handle;
        public IOperationContract Contract { get; private set; }
        public OperationContractToViewHostAdapter(IOperationContract contract)
        {
            this.Contract = contract;
            handle = new ContractHandle(contract);
        }
```

```
        public override string Name
        {
            get
            {
                return Contract.Name;
            }
        }
        public override int NumberOperands
        {
            get
            {
                return Contract.NumberOperands;
            }
        }
    }
    internal static class OperationHostAdapters
    {
        internal static IOperationContract ViewToContractAdapter(Operation view)
        {
            return ((OperationContractToViewHostAdapter)view).Contract;
        }
        internal static Operation ContractToViewAdapter(
                IOperationContract contract)
        {
            return new OperationContractToViewHostAdapter(contract);
        }
    }
```

代码段 HostAdapter/OperationContractToViewHostAdapter.cs

CalculatorContractToViewHostAdapter 类实现抽象宿主视图类 Calculator 的方法，并把调用转发给协定。同样，该类还有一个 ContractHandle 实例，它包含对协定的引用，这类似于插件端类型转换的适配器。但这次仅需要从插件适配器向宿主端的类型转换。

HostAdapter 属性把类标记为一个需要在 HostSideAdapters 目录下安装的适配器。

可从 wrox.com 下载源代码

```
using System.Collections.Generic;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    [HostAdapter]
    internal class CalculatorContractToViewHostAdapter: Calculator
    {
        private ICalculatorContract contract;
        private ContractHandle handle;
        public CalculatorContractToViewHostAdapter(ICalculatorContract contract)
        {
            this.contract = contract;
            handle = new ContractHandle(contract);
        }
        public override IList<Operation> GetOperations()
        {
            return CollectionAdapters.ToIList<IOperationContract, Operation>(
                contract.GetOperations(),
                OperationHostAdapters.ContractToViewAdapter,
```

```
                OperationHostAdapters.ViewToContractAdapter);
            }
            public override double Operate(Operation operation, double[] operands)
            {
                return contract.Operate(OperationHostAdapters.ViewToContractAdapter(
                    operation), operands);
            }
        }
    }
```

代码段 HostAdapter/OperationContractToViewHostAdapter.cs

### 50.2.7　计算器宿主

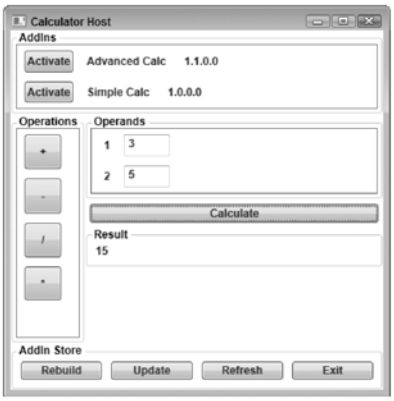


图　50-3

示例宿主应用程序使用 WPF 技术。这个应用程序的用户界面如图 50-3 所示。其顶部是可用的插件列表。左边是活动插件的操作。选择要调用的操作时，就会显示操作数。输入操作数的值后，就可以调用插件的操作。

底部一行的按钮用于重建和更新插件存储器，以及退出应用程序。

下面的 XAML 代码显示了用户界面的树型结构。在 ListBox 元素中，给项模板使用不同的样式，为插件列表、操作列表和操作数列表指定特定的表示方式。

项模板的内容可参见第 35 章。

可从
wrox.com
下载源代码

```
<DockPanel>
    <GroupBox Header="AddIn Store" DockPanel.Dock="Bottom">
        <UniformGrid Columns="4">
            <Button x:Name="rebuildStore" Click="RebuildStore"
                   Margin="5"> Rebuild</Button>
            <Button x:Name="updateStore" Click="UpdateStore"
                   Margin="5">Update</Button>
            <Button x:Name="refresh" Click="RefreshAddIns"
                   Margin="5"> Refresh</Button>
            <Button x:Name="exit" Click="App_Exit" Margin="5">Exit</Button>
        </UniformGrid>
    </GroupBox>
    <GroupBox Header="AddIns" DockPanel.Dock="Top">
        <ListBox x:Name="listAddIns" ItemsSource="{Binding}"
               Style="{StaticResource listAddInsStyle}"/>
    </GroupBox>
    <GroupBox DockPanel.Dock="Left" Header="Operations">
        <ListBox x:Name="listOperations" ItemsSource="{Binding}"
      Style="{StaticResource listOperationsStyle}"/>
        </GroupBox>
            <StackPanel DockPanel.Dock="Right" Orientation="Vertical">
                <GroupBox Header="Operands">
```

```
            <ListBox x:Name="listOperands" ItemsSource="{Binding}"
                  Style="{StaticResource listOperandsStyle}">
            </ListBox>
         </GroupBox>
         <Button x:Name="buttonCalculate" Click="Calculate" IsEnabled="False"
               Margin="5">Calculate</Button>
         <GroupBox DockPanel.Dock="Bottom" Header="Result">
            <Label x:Name="labelResult" />
         </GroupBox>
      </StackPanel>
   </DockPanel>
```

代码段 HostAppWPF/CalculatorHostWindow.xaml

在代码隐藏中，在 Window 的构造函数中调用 FindAddIns()方法。FindAddIns()方法使用 AddInStore 类获得 AddInToken 对象的集合，把它们传递给 listAddIns 列表框的 DataContext 属性，以显示它们。AddInStore.FindAddIns()方法的第一个参数传递宿主视图定义的 Calculator 抽象类，从存储器中查找应用于协定的所有插件。第二个参数传递从应用程序配置文件中读取的管道目录。运行 Wrox 下载网站(http://www.wrox.com)上的示例应用程序时，需要修改应用程序配置文件中的目录，以匹配自己的目录结构。


```
using System;
using System.AddIn.Hosting;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Windows;
using Wrox.ProCSharp.MAF.Properties;
namespace Wrox.ProCSharp.MAF
{
   public partial class CalculatorHostWindow: Window
   {
      private Calculator activeAddIn = null;
      private Operation currentOperation = null;
      public CalculatorHostWindow()
      {
         InitializeComponent();
         FindAddIns();
      }
      void FindAddIns()
      {
         try
         {
            this.listAddIns.DataContext =
               AddInStore.FindAddIns(typeof(Calculator),
               Settings.Default.PipelinePath);
         }
         catch (DirectoryNotFoundException ex)
         {
            MessageBox.Show("Verify the pipeline directory in the " +
                 "config file");
            Application.Current.Shutdown();
         }
```

```
    }
    //...
```

代码段 HostAppWPF/CalculatorHostWindow.xaml

要更新插件存储器的缓存，UpdateStore()和 RebuildStore()方法应映射到 Update 和 Rebuild 按钮的 Click 事件上。在这些方法的实现代码中，使用 AddInStore 类的 Update()或 Rebuild()方法。如果程序集存储在错误的目录下，这些方法就返回一个警告字符串数组。由于管道结构比较复杂，因此第一次把程序集复制到正确的目录下时，其项目配置不太可能完全正确。阅读这些方法返回的信息，可以清楚地了解出错原因。例如，“在程序集\Pipeline\AddInSideAdapters\CalcView.dll 中没有找到可用的 AddInAdapter 部件”消息表示，CalcView 程序集存储在错误的目录下。

```
private void UpdateStore(object sender, RoutedEventArgs e)
{
   string[] messages = AddInStore.Update(Settings.Default.PipelinePath);
   if (messages.Length != 0)
   {
      MessageBox.Show(string.Join("\n", messages),
           "AddInStore Warnings", MessageBoxButton.OK,
           MessageBoxImage.Warning);
   }
}
private void RebuildStore(object sender, RoutedEventArgs e)
{
   string[] messages =
        AddInStore.Rebuild(Settings.Default.PipelinePath);
   if (messages.Length != 0)
   {
      MessageBox.Show(string.Join("\n", messages),
            "AddInStore Warnings", MessageBoxButton.OK,
            MessageBoxImage.Warning);
   }
}
```

在图 50-3 中，可看到插件的旁边有一个 Activate 按钮。单击这个按钮会调用处理程序方法 ActivateAddIn()。在这个方法的实现代码中，使用 AddInToken 类的 Activate()方法激活插件。其中插件加载到用 AddInProcess 类创建的一个新进程中。AddInProcess 类启动 AddInProcess32.exe 进程。把该进程的 KeepAlive 属性设置为 false，只要最后一个插件引用被垃圾回收了，该进程就停止。AddInSecurityLevel.Internet 参数使插件在有限的权限下运行。ActivateAddIn()方法的最后一条语句调用 ListOperations()方法，ListOperations()方法又调用插件的 GetOperations()方法。GetOperations()方法把返回的列表赋予 listOperations 列表框的 DataContext 属性，用于显示所有操作。

```
private void ActivateAddIn(object sender, RoutedEventArgs e)
{
   FrameworkElement el = sender as FrameworkElement;
   Trace.Assert(el != null, "ActivateAddIn invoked from the wrong " +
         "control type");

   AddInToken addIn = el.Tag as AddInToken;
   Trace.Assert(el.Tag != null, String.Format(
         "An AddInToken must be assigned to the Tag property " +
         "of the control {0}", el.Name);
```

```
    AddInProcess process = new AddInProcess();
    process.KeepAlive = false;

    activeAddIn = addIn.Activate<Calculator> (process,
          AddInSecurityLevel.Internet);
    ListOperations();
}
void ListOperations()
{
    this.listOperations.DataContext = activeAddIn.GetOperations();
}
```

激活插件并在 UI 上显示操作列表后，用户就可以选择操作。在 Operations 类别中 Button 的 Click 事件赋予处理程序方法 OperationSelected()。在这个方法的实现代码中，检索赋予 Button 的 Tag 属性的 Operation 对象，获得操作所需要的操作数个数。为了允许用户给操作数添加值，把一个 OperandUI 对象数组绑定到 listOperands 列表框上。

```
private void OperationSelected(object sender, RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "OperationSelected invoked from " +
          "the wrong control type");
    Operation op = el.Tag as Operation;
    Trace.Assert(el.Tag != null, String.Format(
          "An AddInToken must be assigned to the Tag property " +
          "of the control {0}", el.Name);
    currentOperation = op;
    ListOperands(new double[op.NumberOperands]);
}
private class OperandUI
{
    public int Index { get; set; }
    public double Value { get; set; }
}
void ListOperands(double[] operands)
{
    this.listOperands.DataContext =
       operands.Select((operand, index) =>
             new OperandUI()
             { Index = index + 1, Value = operand }).ToArray();
}
```

通过 Calculate 按钮的 Click 事件调用 Calculate()方法。这里从 UI 中检索操作数，把操作和操作数传递给插件的 Operate()方法，结果显示为标签的内容：

```
private void Calculate(object sender, RoutedEventArgs e)
{
    OperandUI[] operandsUI = (OperandUI[])this.listOperands.DataContext;
    double[] operands = operandsUI.Select(opui = > opui.Value).ToArray();
    labelResult.Content = activeAddIn.Operate(currentOperation,
         operands);
}
```

### 50.2.8　其他插件

现在完成了艰苦的工作。创建了管道组件和宿主应用程序。管道现在可以正常工作，还可以轻松地在宿主应用程序中添加其他插件，例如，下面的 Advanced Calculator 插件：


可从
wrox.com
下载源代码

```
[AddIn("Advanced Calc", Publisher = "Wrox Press", Version = "1.1.0.0",
      Description = "Another AddIn Sample")]
public class AdvancedCalculatorV1: Calculator
```

代码段 AdvancedCalcAddIn/AdvancedCalculator.cs

## 50.3　小结

本章学习了新颖的.NET 3.5 技术的概念：Managed Add-In Framework。

MAF 使用管道概念使宿主程序集和插件程序集的创建完全独立。清楚定义的协定把宿主视图和插件视图分开。适配器可以使宿主端和插件端独立地修改。

下一章介绍如何通过.NET Enterprise Services 使用.NET 组件。