



我们在2.2.2节曾讨论过元数据（metadata）以及它在程序集中的物理存储方式。本章将会看到它们是如何构成反射与attribute机制的基础的。

7.1 反射

反射机制代表了在执行期一个程序集的类型元数据的使用。通常情况下，该程序集是在另一个程序集执行的时候被显式载入的，不过它也可以被动态生成。

反射这个词用于表明我们使用了一个程序集的映像（就像镜子中的映像）。该映像由程序集的类型元数据构成。我们有时候也会使用内省（introspection）这个术语来表示反射。

7.1.1 何时需要反射

我们收集了一些反射机制的使用分类，在本章接下来的小节中将对它们展开更详细的讨论。反射机制可以在下述场景中用到。

- 在应用程序执行时，我们可以使用类型元数据的动态分析来探索程序集中的类型。例如，ildasm.exe与Reflector工具会显式地装载一个程序集中的模块并分析它们的内容（参见2.3节）。
- 使用后期绑定的期间。该技术需要使用程序集中的一个类，而该类在编译期是未知的。后期绑定技术通常用于解释型语言，如脚本语言。
- 当我们希望使用attribute中的信息的时候。
- 当我们希望从类外部访问类中的非公开成员的时候。当然，这种行为应该尽量避免，不过有时候还是有必要使用的。例如，在编写没有非公开成员就无法完成的单元测试的时候。
- 在动态构造程序集的期间。为了使用一个动态构造的程序集中的类，我们必须显式地使用后期绑定技术。

CLR与Framework在某些情况下会使用反射机制。例如，在值类型的Equals()方法的默认实现中，使用反射逐一比较两个实例中的字段。

CLR在序列化对象期间也使用反射，以确定哪个字段需要被序列化。甚至垃圾收集器也会在回收过程中使用它来构造引用树。

7.1.2 .NET 反射有何新意

反射机制的底层原理并不是什么新概念。很早以前我们就能够动态地分析一个可执行程序了，尤其是通过使用自描述信息。TLB格式（参见8.4节）就是为了这个目的构想出来的。而TLB格式中的数据正是来自于IDL（Interface Definition Language，接口定义语言）格式。IDL语言也可以被视为一种自描述语言。.NET中的反射机制则比TLB与IDL格式更进了一步。

- 在某些基类的帮助下，它更易于使用。
- 它比TLB与IDL语言更抽象。例如，它不使用物理地址，这意味着它在32位与64位机器上都能

发挥作用。

- ❑ 相比TBL元数据，.NET元数据总是包含在它所描述的模块中。
- ❑ 它描述数据的详细程度远胜于TLB格式。具体来说，我们能够获得一个程序集中声明的任何类型的所有可能信息（例如：类中方法的某个参数的类型）。

.NET反射之所以能描述如此详细的内容，归功于.NET Framework 中众多的基类，通过它们可以从一个包含在AppDomain中的程序集中抽取出各种类型元数据（type metadata）并使用它们。这些类大多可以在System.Reflection命名空间中找到，而且程序集中每个类型的元素都有一个类与之对应。

- ❑ 有一个类的实例代表了程序集（System.Reflection.Assembly）；
- ❑ 有一个类的实例代表了类与结构（System.Type）；
- ❑ 有一个类的实例代表了方法（System.Reflection.MethodInfo）；
- ❑ 有一个类的实例代表了字段（System.Reflection.FieldInfo）；
- ❑ 有一个类的实例代表了方法参数（System.Reflection.ParameterInfo）。

.....

最后要提醒大家，这些类仅提供了一种从逻辑上查看全体类型元数据的方法。但是我们看到的内容与物理的全体类型元数据并不会完全吻合，其中某些用于程序集内部组织的元素并没有被展现出来。

System.Reflection命名空间下的所有类以一种逻辑方式相互联系。例如，从一个System.Reflection.Assembly的实例中，可以获得一个System.Type实例的列表。从一个System.Type的实例中，可以获得一个System.Reflection.MethodInfo实例表。而从一个System.Reflection.MethodInfo实例中，又可以获得一个System.Reflection.ParameterInfo实例表。所有这些如图7-1所示。

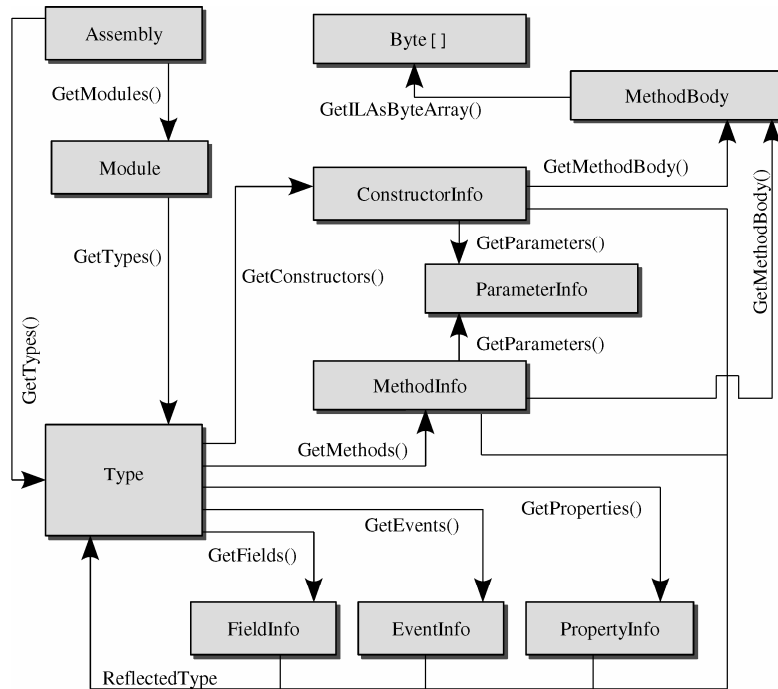


图7-1 反射类之间的交互

你可能会发现我们无法深入到IL指令层面，而只能到达字节表的层面，在字节表中包含一个方法的IL体。因此，为了查看IL指令，你可能想使用某些库，比如Cecil（由Jean-Baptiste Evain开发）、ILReader（由Lutz Roeder开发）或Rail（由Coimbra大学开发）。应该了解，在.NET 2下，反射知道如何处理泛型类型、泛型方法以及对参数类型的约束（参见13.10.2节）。

7.1.3 对载入 AppDomain 的程序集的反射

下面的例子展现了如何使用System.Reflection命名空间下的类分析类型元数据。事实上，这里我们展现了一个分析它自身的类型元数据的程序集。为了获得一个代表该程序集的Assembly类的实例，我们使用了静态方法Assembly.GetExecutingAssembly()。

例7-1

```
using System;
using System.Reflection;
class Program{
    public static void Main(){
        Assembly assembly = Assembly.GetExecutingAssembly();
        foreach ( Type type in assembly.GetTypes() ){
            Console.WriteLine( "Class: " + type );
            foreach ( MethodInfo method in type.GetMethods() ){
                Console.WriteLine( "  Method: " + method );
                foreach ( ParameterInfo param in method.GetParameters() )
                    Console.WriteLine( "    Param: " + param.GetType() );
            }
        }
    }
}
```

程序输出如下：

```
Class: Program
Method: Void Main()
Method: System.Type GetType()
Method: System.String ToString()
Method: Boolean Equals(System.Object)
Param: System.Reflection.ParameterInfo
Method: Int32 GetHashCode()
```

7.1.4 从元数据获取信息

本节目的在于展示一个小程序，该程序使用反射机制显示了包含在System与mscorlib程序集中的异常类集合。我们是建立在所有异常类都继承自System.Exception这一事实的基础上的。不是直接继承自System.Exception的异常类型会用一个星号标记。

我们可以依据所有attribute类都继承自System.Attribute类这一事实，很容易地修改该程序来显示框架中所有的attribute类。

例7-2

```
using System;
using System.Reflection;
class Program {
    static void Main() {
        // Build the strong name of the 'System' assembly.
        // Its version number is the same as the one of the assembly
    }
}
```

```

// 'mscorlib' which contains the System.Object class.
string systemAsmStrongName = "System, Culture = neutral, " +
    "PublicKeyToken=b77a5c561934e089, Version=" +
    typeof(System.Object).Assembly.GetName().Version.ToString();

// Explicitly load the 'System' assembly for reflecting on it.
// There is no need to load the 'mscorlib' assembly since
// it is automatically and implicitly loaded by the CLR.
Assembly.ReflectionOnlyLoad( systemAsmStrongName );

// For each assembly in the current AppDomain...
foreach (Assembly a in AppDomain.CurrentDomain.GetAssemblies()) {
    Console.WriteLine("\nAssembly:" + a.GetName().Name);

    // For each type in the assembly 'a'...
    foreach (Type t in a.GetTypes()) {
        // Treat only public classes...
        if (!t.IsClass || !t.IsPublic) continue;
        bool bDerivedException = false;
        bool bDirectInherit = true;

        // Is System.Exception a base type of 't'?
        Type baseType = t.BaseType;
        while ( baseType != null && !bDerivedException ) {
            // To find attribute classes replace the following line
            // by: if( baseType == typeof(System.Attribute))
            if ( baseType == typeof(System.Exception) )
                bDerivedException = true;
            else bDirectInherit = false;
            baseType = baseType.BaseType;
        } // end while
        // Display the name of the class if it is an exception
        // class. Put a star if the class doesn't inherit directly
        // from System.Exception.
        if ( bDerivedException )
            if ( bDirectInherit )
                Console.WriteLine( " " + t );
            else
                Console.WriteLine( " *" + t );
    } // end foreach(Type...)
} // end foreach(Assembly...)
}

```

注意前一个例子中使用的`Assembly.ReflectionOnlyLoad()`方法。通过该方法可以告诉CLR，载入的程序集将仅用于反射。因此，CLR将不允许以这种方法载入的程序集的代码执行。同样，以`Reflection Only`模式装载程序集会稍微快些，因为CLR不需要完成任何安全验证工作。`Assembly`类提供`bool ReflectionOnly{get;}`属性用于判断一个程序集是否是通过该方法载入的。

7.2 后期绑定



在开始本节之前，建议对面向对象编程的基本内容有着很好的理解，尤其是有关多态的概念。该主题请参见第12章。

7.2.1 “绑定一个类”的含义

首先，我们需要在“绑定一个类”的含义上达成共识。我们将使用“软件层”这个术语而不是“程

序集”，因为后者在其他技术中也被用到。

在使用类（实例化类并使用类的实例）的软件层与定义类的软件层之间会建立起一个类的联系。具体来说，这个联系是使用类的软件层中对类中方法的调用与这些方法在定义类的软件层中的物理地址之间的对应关系。正是通过这个联系，当类方法被调用时线程才能继续执行下去。

一般来说，我们将一个类的联系分为三类：完全在编译期创建的早期绑定，在编译期创建了部分联系的动态绑定以及在执行期创建的后期绑定。

7.2.2 早期绑定与动态绑定

早期绑定联系由编译器在根据.NET源代码创建程序集的期间创建。我们无法为一个虚方法或抽象方法创建一个早期绑定。事实上，当一个虚方法或抽象方法被调用时，多态机制会在执行期间根据被调用方法的实际对象确定将要执行的代码。这种情况下，该联系被视为动态绑定。在其他资料中，动态绑定有时被称为隐式的后期绑定，因为它们是由多态机制隐式创建的并且是在执行期完成的。

现在让我们进一步观察早期绑定，它们是为静态方法或类中那些不是虚方法或抽象方法的方法创建的。如果严格遵循前一节中对“类的联系”的定义，.NET中是不存在早期绑定的。事实上，我们必须先等待JIT编译器将方法体转换为机器语言，才能知道它在进程地址空间中的物理地址。创建程序集的编译器并不知道这个方法的地址信息^①。我们在4.6节看到，为了解决这个问题，创建程序集的编译器在IL代码中方法将被调用的位置插入了与被调用的方法相对应的元数据符号（metadata token）。当方法体被即时（JIT）编译的时候，CLR在内部保存了方法与机器语言下的方法体的物理地址的对应联系。这段被称为存根的信息被物理保存到一个与方法相关的内存地址中。

以上的认识很重要，因为在像C++这样的语言中，当一个方法不是虚方法或抽象方法（即C++中的纯虚函数）时，编译器就可以计算出该方法体在机器语言下的物理地址。然后，编译器在每个调用该方法的位置插入一个指向该内存地址的指针。这个区别给了.NET很大的优势，因为编译器不需要再考虑诸如内存表现之类的技术细节。IL代码完全独立于它所运行的物理层。

而在动态绑定中，其中几乎所有的事物都是以与早期绑定中相同的方式工作的。编译器在IL代码中方法被调用的位置插入了与被调用的虚（或抽象）方法相对应的元数据符号。这里，我们提到的元数据符号是属于定义在引用类型中的方法的，而该引用类型就是将发生方法调用的那个类型。然后就是CLR的工作，它将在执行期间根据引用对象的具体实现确定跳转到哪个方法。

插入一个类型元数据符号，编译器使用这项技术创建动态绑定与早期绑定，它主要用于以下三种情况。

- 当包含在模块中的代码调用了—个处在同一模块中的方法时。
- 当包含在模块中的代码调用了—个处在同一程序集的不同模块中的方法时。
- 当包含在程序集的一个模块中的代码调用了定义在另一个在编译期引用进来的程序集中的方法时。在运行时，如果即时编译方法调用的时候该程序集尚未载入，那么CLR将隐式地装载它。

7.2.3 后期绑定

程序集A的代码可以实例化并使用一个定义在程序集B中的类型，而该类型可能在A编译的时候并没有被引用。我们把这种类型的联系描述为后期绑定。我们之所以在句中使用“后期”这个词是因为绑定是在代码执行期完成的而非编译期。这种类型的绑定同样是显式的，因为被调用的方法的名称必须使用一个字符串显式地指定。

^① 因为在程序运行时才发生JIT编译。——译者注

后期绑定在微软的开发世界中并不是什么新的概念。COM技术中的自动化机制就是一个例子，它使用IDispatch接口作为变通方法，允许脚本语言或者弱类型的语言如VB使用后期绑定。后期绑定的概念同样存在于Java中。

后期绑定是习惯于C++的开发者很难理解的概念之一。事实上，在C++中只存在早期与动态绑定。难于理解的原因来自以下事实：我们知道创建一个绑定所需的必要信息（即元数据符号）处在将被调用的类所在的程序集B之中，但是我们不能理解为什么开发者不能利用编译器的能力，在编译A的期间，通过引用程序集B创建早期与动态绑定。对此，存在以下解释：

- 最常见的原因在于某些语言根本就没有编译器！在一个脚本语言中，指令是被一条一条解释的。在这种情况下，只存在后期绑定。通过使用后期绑定，可以使用由解释型语言编译的程序集中的类。在.NET中可以方便地使用后期绑定技术这一事实，使得创建一个专有的解释/动态语言变得相对容易（比如IronPython语言<http://www.ironpython.com/>）。
- 我们可能希望在由编译型语言如C#写成的程序中使用后期绑定技术。原因在于，使用后期绑定可以为应用程序的通用架构带来某种程度的灵活性。该技术实际上是一种最近很流行的被称为插件的设计模式，我们将在本章对它做进一步介绍。
- 某些应用程序需要调用尚未获得的程序集中的代码。一个典型的例子就是开源工具NUnit，该工具可以通过调用任意一个程序集的方法来测试其代码。我们将在稍后构造一个自定义attribute的时候，再进一步接触这个话题。
- 如果在程序集A编译期间程序集B尚不存在，我们就必须在A中的代码与B中的类之间使用后期绑定。这种情况我们将在稍后谈到动态构造程序集的时候介绍。

一些人喜欢使用后期绑定来代替多态。事实上，因为在调用期间，只考虑方法的名称与签名式，而与被调用方法所处对象的类型无关，所以只需要在实现对象的时候提供具有合适名称和签名式的方法即可。但是，我个人不推荐这种做法，因为它的约束性太差，并且无法促使应用程序的开发者去做恰当的设计以及使用抽象接口。

除了刚提到的原因外，还有就是不需要显式地使用后期绑定。不要为了好玩而在应用程序中使用后期绑定，因为：

- 会失去由编译器完成的语法验证的好处。
- 后期绑定的性能远不如早期或动态绑定方法。（即使使用了后面提到的优化方法。）
- 无法为被混淆的类创建后期绑定。事实上，在混淆过程中，程序集中包含的类的名称会被改变。因此，后期绑定机制无法正确地找到合适的类。

7.2.4 在C#编译到IL期间如何实例化一个未知的类

如果一个类或结构在编译期是未知的，那么就无法使用new操作符对它实例化。幸运的是，.NET Framework确实提供了一些类，使用这些类可以创建那些在编译期间未知的类的实例。

1. 精确化一个类型

现在让我们看一下，在指定一个类型时，可以采取的各种不同的技术。

- 某些类的某些方法接受一个包含类型完整名称（包括命名空间）的字符串。
- 其他方法接受一个System.Type类的实例。在一个AppDomain中，每个System.Type类的实例代表一种类型，而且不会有二个实例同时代表该类型。

获得一个System.Type类的实例的几种方式：

- 在C#中，我们通常使用typeof()关键字，它接受一个类型作为参数并返回相应的System.Type实例。

- 也可以使用 `System.Type` 类中 `GetType()` 静态方法的一个重载版本。
- 如果一个类型被封装在另一个类中，可以使用 `System.Type` 类的非静态方法 `GetNestedType()` 或 `GetNestedTypes()`。还可以使用 `System.Reflection.Assembly` 类的非静态方法 `GetType()`、`GetTypes()` 或 `GetExportedTypes()`。
- 也可以使用 `System.Reflection.Module` 类的非静态方法 `GetType()`、`GetTypes()` 或 `FindTypes()`。

现在假设以下程序被编译为 `Foo.dll` 程序集。我们将要展示几种创建一个 `NMFoo.Calc` 类的实例的方法，这些方法允许在一个没有引用 `Foo.dll` 的程序集中完成创建。

例7-3 Foo.dll程序集的代码

```
using System;
namespace NMFoo {
    public class Calc {
        public Calc() {
            Console.WriteLine("Calc.Constructor called!");
        }
        public int Sum(int a, int b) {
            Console.WriteLine("Method Calc.Sum() called!");
            return a + b;
        }
    }
}
```

2. 使用 `System.Activator` 类

`System.Activator` 类提供了两个静态方法 `CreateInstance()` 与 `CreateInstanceFrom()`，通过它们可以创建一个在编译期间未知的类的实例。例如：

例7-4

```
using System;
using System.Reflection;
class Program {
    static void Main() {
        Assembly assembly = Assembly.Load("Foo.dll");
        Type type = assembly.GetType("NMFoo.Calc");
        object obj = Activator.CreateInstance(type);
        // 'obj' is a reference toward an instance of NMFoo.Calc.
    }
}
```

这两个方法中都提供了一些重载版本，甚至还有泛型版本，其中使用了以下参数。

- 用于代表一个字符串的类或 `System.Type` 的一个实例；
- 包含类的程序集的名称，该参数是可选的；
- 构造参数列表，该参数是可选的。

如果包含类的程序集并未出现在 `AppDomain` 中，调用 `CreateInstance()` 或 `CreateInstanceFrom()` 方法会导致该程序集被载入。取决于我们调用的是 `CreateInstance()` 方法还是 `CreateInstanceFrom()` 方法，在内部会调用 `System.AppDomain.Load()` 或 `System.AppDomain.LoadFrom()` 方法来载入程序集。CLR 会根据提供的参数选择类的一个构造函数，并返回一个包含了一个封送对象的 `ObjectHandle` 类的实例。在介绍 .NET remoting 的第22章中，我们会在分布式应用的环境下展示这些方法的另一种用法。

使用System.Type类的一个实例来指定类型的CreateInstance()重载版本会直接返回对象的一个实例。

System.Activator还有一个CreateComInstanceFrom()方法，该方法用于创建一个COM对象的实例，以及一个用于创建远程对象的GetObject()方法。

3. 使用System.AppDomain类

System.AppDomain类拥有CreateInstance()、CreateInstanceAndUnwrap()、CreateInstanceFrom()与CreateInstanceFromAndUnwrap()这四个非静态方法，通过它们可以创建一个在编译期间未知的类的实例，例如：

例7-5

```
using System;
using System.Reflection;
class Program {
    static void Main() {
        object obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
            "Foo.dll", "NMFoo.Calc");
        // 'obj' is a reference toward an instance of NMFoo.Calc.
    }
}
```

这些方法与之前谈到的System.Activator中的方法类似。不过，通过它们可以选择将对象创建在哪个AppDomain中。此外，“AndUnwrap()”版本会返回一个对对象的直接引用，该引用是从一个ObjectHandle类的实例中获得的。

4. 使用System.Reflection.ConstructorInfo类

System.Reflection.ConstructorInfo类的实例引用一个构造函数。该类的Invoke()方法在内部为构造函数创建了一个后期绑定，并通过这个绑定调用构造函数。因此，通过它们可以创建一个该构造函数所属类型的实例。例如：

例7-6

```
using System;
using System.Reflection;
class Program {
    static void Main() {
        Assembly assembly = Assembly.Load ("Foo.dll");
        Type type = assembly.GetType("NMFoo.Calc");
        ConstructorInfo constructorInfo = type.GetConstructor( new Type[0] );
        object obj = constructorInfo.Invoke( new object[0] );
        // 'obj' is a reference toward an instance of NMFoo.Calc.
    }
}
```

5. 使用System.Type类

通过System.Type类的非静态方法InvokeMember()可以创建一个在编译期间未知的类的实例，只需要在调用的时候使用BindingFlags枚举量中的CreateInstance值即可。例如：

例7-7

```
using System;
using System.Reflection;
class Program {
    static void Main() {
```

```

Assembly assembly = Assembly.Load("Foo.dll");
Type type = assembly.GetType("NMFoo.Calc");
Object obj = type.InvokeMember(
    null, // Don't need to provide a name for calling a constructor.
    BindingFlags.CreateInstance,
    null, // Don't need a binder.
    null, // Don't need a target object since we build it.
    new Object[0]); // No parameters.
// Here, 'obj' is a reference toward an instance of NMFoo.Calc.
}
}

```

6. 特殊情况

通过以上介绍的方法，几乎可以创建任何一种类或结构的实例。下面是两种特殊情况。

- 为了创建一个数组，必须调用System.Array类中的静态方法CreateInstance()。
- 为了创建一个委托对象，必须调用System.Delegate类中的CreateDelegate()方法。

7.2.5 使用后期绑定

现在我们知道如何创建在编译期间未知的类的实例，为了使用这些实例，让我们来看一下这些类型的成员之间的后期绑定的创建过程。同样有几种方法可以实现。

1. Type.InvokeMember()方法

让我们回到Type.InvokeMember()方法，之前我们使用它通过调用未知的类型的一个构造函数创建了一个在编译期间该未知的类型的实例。在内部实现中，该方法完成下面3个任务。

- 它在它被调用的类型上寻找与所提供的信息相对应的成员。
- 如果该成员被找到，为它创建一个后期绑定。
- 使用该成员（是方法就调用，是构造函数就创建一个对象的实例，是字段就读取或设值，是属性就执行set或get访问器，等等）。

下面的例子展示了如何调用NMFoo.Calc类的实例上的Sum()方法（注意在调试期间，调试器能够进入使用后期绑定的方法体中）。

例7-8

```

using System;
using System.Reflection;
class Program {
    static void Main() {
        object obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
            "Foo.dll", "NMFoo.Calc");

        Type type = obj.GetType();
        object[] parameters = new object[2];
        parameters[0] = 7;
        parameters[1] = 8;
        int result = (int) type.InvokeMember(
            "Sum", // Name of the method.
            BindingFlags.InvokeMethod,
            null, // Don't need a binder.
            obj, // The target object.
            parameters); // Parameters.

        // Here, the value of 'result' is 15.
    }
}

```

Type.InvokeMember()方法最常用的重载版本是：

```

public object InvokeMember(
    string      name,           // The name of the member.
    BindingFlags invokeAttr,   // Characteristics supported by the
                                // member to found. (private/public,
                                // instance/static, ignore case ...)
    Binder      binder,        // Some rules to bind with the member.
    object      target,        // The target object on which
                                // the member is invoked.
    object[]    args           // Arguments of the call.
);

```

`invokeAttr`参数是一个二进制标志位，它指示了搜索何种类型的成员。为了搜索方法，我们使用 `BindingFlags.InvokeMethod` 标志位。各种标志位的介绍请参见MSDN上名为“`BindingFlags Enumeration`”的文章。

`binder`参数是一个 `Binder` 类型的对象，它会指示 `InvokeMember()` 方法如何搜索。大多数情况下，该参数可以设为 `null` 以表示希望使用默认值，也就是 `System.Type.DefaultBinder`。 `Binder` 类型的对象提供了以下类型的信息：

- 它指示了参数会接受何种类型的转换。在上一个例子中，我们可以提供两个 `double` 类型的参数。由于 `DefaultBinder` 支持从 `double` 到 `int` 的转换，所以仍然能够成功地调用方法。
- 它指示了我们是否在参数列表中使用了可选参数。

所有这些（尤其是类型转换表）在MSDN上一篇名为 *Type.DefaultBinder Property* 的文章中有更详细的介绍。我们还可以通过继承 `Binder` 类，创建自己的 `binder` 对象。不过在大多数情况下使用一个 `DefaultBinder` 的实例就足够了。

如果在后期绑定成员的调用时引发了异常，`InvokeMember()` 会截获异常并重新抛出一个 `System.Reflection.TargetInvocationException` 类型的异常。自然地，在方法中引发的异常会被重新抛出的异常中的 `InnerException` 属性所引用。

最后注意，在创建一个后期绑定时，无法访问非公有成员。否则，通常会抛出 `System.Security.SecurityException` 异常。不过，如果 `System.Security.Permissions.ReflectionPermissionFlags` 的 `TypeInfo` 标志位（可以通过 `System.Security.Permissions.ReflectionPermission` 类的实例访问）被设为真，就可以访问非公有成员。如果 `MemberAccess` 标志位被设为真，就可以访问非可见类型（即以非公有方式封装在其他类型中）^①。

2. 一次绑定，多次调用

我们看到，通过一个 `ConstructorInfo` 实例可以创建一个后期绑定以调用一个构造函数，以同样的方式，通过一个 `System.Reflection.MethodInfo` 类的实例也可以创建一个后期绑定并调用任意一个的方法。使用 `MethodInfo` 类而不是 `Type.InvokeMember()` 方法的优势在于可以节省每次调用时搜索成员的时间，因此会带来一些性能上的优化。如下例所示。

例7-9

```

using System;
using System.Reflection;
class Program {
    static void Main() {
        object obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
            "Foo.dll", "NMFoo.Calc");
    }
}

```

① 这段内容似乎有误，`TypeInfo` 标志位表示是否可以绑定非可见成员，而 `MemberAccess` 标志位表示是否可以调用非可见成员，有待进一步查证。——译者注

```

Type type = obj.GetType();
// Create a late bind with the 'Sum' method.
MethodInfo methodInfo = type.GetMethod("Sum");
object[] parameters = new object[2];
parameters[0] = 7;
parameters[1] = 8;
int result;
// 10 calls to 'Sum'.
for (int i = 0; i < 10; i++)
    result = (int)methodInfo.Invoke( obj, parameters );
}
}

```

3. VB.NET如何背着你创建后期绑定

让我们为VB.NET做一些旁注，并观察一下当**Strict**选项被设为**off**之后，该语言如何背着你秘密地使用后期绑定。例如，下面的VB.NET程序……

例7-10 VB.NET与后期绑定

```

Option Strict Off
Module Module1
    Sub Main()
        Dim obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
            "Foo.dll", "NMFoo.Calc")

        Dim result As Integer = obj.Sum(7, 8)
    End Sub
End Module

```

……等同于下面的C#程序：

例7-11

```

using System;
using System.Reflection;
using Microsoft.VisualBasic.CompilerServices;
class Program {
    static void Main() {
        object obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
            "Foo.dll", "NMFoo.Calc");

        object[] parameters = new object[2];
        parameters[0] = 7;
        parameters[1] = 8;
        int result = (int) LateBinding.LateGet(obj, null, "Sum",
            parameters, null, null);
    }
}

```

7.2.6 利用接口：使用后期绑定的正确方法

为了使用一个在编译期间未知的类，除了我们介绍过的通过使用后期绑定的方法外，还有另一个完全不同的方法。该方法具有较大优势，因为与使用早期或动态绑定相比，它在性能上几乎没有损失。不过，为了使用该“秘诀”，你必须迫使自己遵循某种规范（实际上就是名为插件的设计模式）。

我们的想法是确保在编译时未知的类型实现了一个接口，而该接口是编译器所知的。为此，我们不得不创建第三个程序集，用于承载该接口。让我们用三个程序集重写calc的例子：

例7-12 包含接口的程序集的代码（InterfaceAsm.cs）

```

namespace NMFoo {
    public interface ICalc {

```

```

        int Sum(int a, int b);
    }
}

```

例7-13 包含目标类的程序集的代码 (ClassAsm.cs)

```

using System;
namespace NMFoo {
    public class CalcWithInterface : ICalc {
        public CalcWithInterface() {
            Console.WriteLine("Calc.Constructor called!");
        }
        public int Sum(int a, int b) {
            Console.WriteLine("Method Calc.Sum() called!");
            return a + b;
        }
    }
}

```

例7-14 在编译期目标类未知的客户程序集的代码 (ProgramAsm.cs)

```

using System;
using System.Reflection;
using NMFoo;
class Program {
    static void Main() {
        ICalc obj = AppDomain.CurrentDomain.CreateInstanceAndUnwrap(
            "ClassAsm.dll", "NMFoo.CalcWithInterface") as ICalc;
        int result = obj.Sum(7, 8);
    }
}

```

注意，通过显式地将 `CreateInstanceAndUnwrap()` 方法返回的对象强制转换为 `ICalc` 类型，就可以通过动态链接方式调用 `Sum()` 方法。我们还可以使用 `Activator.CreateInstance<ICalc>()` 这个泛型重载版本来避免类型转化。

图7-2对3个程序集的组织结构以及它们之间的联系做了总结。

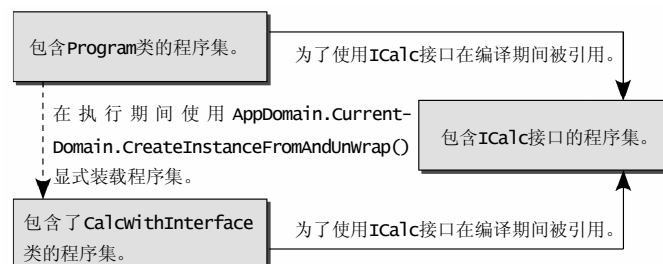


图7-2 插件设计模式与程序集组织结构

根据插件设计模式背后的思想，那些在 `CreateInstanceAndUnwrap()` 方法中用于创建实例所必需的数据（这里是 `"Foo.dll"` 与 `"NMFoo.CalcWithInterface"` 两个字符串）通常保存配置文件中。这样，就可以通过修改配置文件来选择新的实现而无需重新编译。

插件设计模式的一个变种就是使用抽象类代替接口。

最后，应该知道还可以使用委托来创建一个方法的后期绑定。尽管该方法比使用 `MethodInfo` 更加

高效，但是一般来说我们更喜欢使用插件设计模式。

7.3 attribute

7.3.1 attribute 是什么

一个attribute是一份标记代码中的元素的信息，这个元素可以是一个类或一个方法。例如，.NET Framework 提供了System.ObsoleteAttribute，该attribute可用于标记一个方法，如下所示（注意使用[]括号的语法）：

```
[System.ObsoleteAttribute()]
void Fct() { }
```

Fct()方法被标记上了System.ObsoleteAttribute信息，该信息会在编译期间被插入到程序集中，以后可以被C#编译器使用。当调用该方法时，编译器会发出警告，提示最好避免调用废弃方法，因为此类方法可能在将来的版本中消失。如果没有attribute，就不得不通过合适的文档表述出Fct()方法现在处于废弃状态这一事实；而且这种做法的缺点在于，无法保证客户会阅读文档从而知道该方法现在是废弃的。

7.3.2 何时需要 attribute

使用attribute的优势在于它所包含的信息会被插入到程序集中，而这些信息可以在不同的时间用于各种不同目的：

- 一些专门针对编译器的标准attribute不会保存到程序集中。例如，SerializationAttribute并不会直接为一个类型加上特定的标记，而只是告诉编译器该类型可以被序列化。因此，编译器在将被CLR在执行期间使用的具体类型上设置某些标志。像SerializationAttribute这样的attribute，也被称为伪attribute。
- attribute可以在CLR执行期间使用。例如，.NET Framework提供了System.ThreadStaticAttribute，当一个静态字段被标记上该attribute时，CLR将确保在执行期间每个线程中只有该字段的一个版本。
- attribute可以在调试器执行期间使用。因此，通过System.Diagnostics.DebuggerDisplayAttribute可以在调试期间定制代码中某个元素的显示内容（例如某个对象的状态）。
- attribute可以被一个工具使用。例如，.NET Framework提供了System.Runtime.InteropServices.ComVisibleAttribute，当一个类被标记上该attribute，tlbexp.exe 工具会为该类产生一个文件，使得该类可以被当作一个COM对象使用。
- attribute可以在用户代码执行期间使用，此时需要使用反射机制去访问attribute信息。例如，使用attribute验证类中字段的值就是一件很有趣的事。一个字段必须处于某个范围内，一个引用字段必须非空，一个字符串字段最多包含100个字符，……由于存在反射机制，使得编写代码以验证任何一个被标记字段的状态变得很容易。稍后，我们将展示一个在代码中使用attribute的例子。
- attribute可以在用户通过诸如ildasm.exe或Reflector这样的工具分析程序集的时候使用。因此，可以想象一个attribute将会为代码中的一个元素赋予一个说明其特性的字符串。由于该字符串被包含在程序集中，所以我们可以直接查阅这些注释而无需访问源代码。

7.3.3 关于 attribute 应该知道的事

- 一个attribute必须由一个继承于System.Attribute的类定义。

- attribute类的一个实例只有在被反射机制访问时才会被实例化。根据它的使用情况，一个attribute类不一定会被实例化（像System.ObsoleteAttribute那样的attribute就不需要被反射机制使用）。
- .NET Framework 内置了一些attribute以供使用。某些attribute是专门为CLR提供的，其他的则被编译器或微软提供的工具所使用。
- 可以创建自己的attribute类，不过它们只能被你的程序使用，因为你无法修改编译器或CLR。
- 习惯上，一个attribute类的名称会以Attribute为后缀。不过，在C#中，一个名为XXXAttribute的attribute，在标记代码中元素的时候既可以使用XXXAttribute表示也可以简单地用XXX表示。在专门介绍泛型的那一章中，我们在13.10.3节中讨论了attribute概念与泛型概念之间相互交叠的规则。

7.3.4 可以应用 attribute 的代码元素

attribute将被应用于源代码中的各种元素。下面是可以使用attribute标记的所有元素。它们是由AttributeTargets枚举量的值定义的。

元素类型	attribute的作用范围
All	所有的代码元素：程序集本身、类、类成员、委托、事件、字段、接口、方法、模块、参数、属性、返回值与结构
Assembly	程序集本身
Class	类
Constructor	构造函数
Delegate	委托
Enum	枚举量
Event	事件
Field	字段
GenericParameter	泛型参数
Interface	接口
Method	方法
Module	模块
Parameter	参数
Property	属性
ReturnValue	返回值
Struct	结构

7.3.5 .NET Framework 中的一些标准 attribute

要想很好地理解一个attribute，首先要很好地了解它的应用场景。因此，每个标准attribute将在专门提到它们的章节中介绍。

- 一些与安全管理相关的attribute参见6.6节。
- 一些与P/Invoke机制相关的attribute参见8.1节。
- 一些与在.NET应用程序中使用COM相关的attribute参见原书8.4.3节。
- 一些与序列化机制相关的attribute参见22.3节。
- 一些与XML序列化相关的attribute参见原书21.9.2节。

允许实现一个同步机制的`System.Runtime.Remoting.Contexts.SynchronizationAttribute`参见5.9节。

允许提示编译器对某些方法进行有条件编译的`ConditionalAttribute`参见9.3.2节。

用于针对静态字段修改线程行为的`ThreadStaticAttribute`参见原书5.13.1节。

用于提示编译器是否必须做某些验证的`CLSCompliantAttribute`参见4.9.2节。

用于实现`params` C#关键字的`ParamArrayAttribute`参见4.9.2节。

`CategoryAttribute`参见18.5节。

7.3.6 自定义的 attribute 的示例

一个自定义的attribute就是你通过定义一个继承于`System.Attribute`的类而为自己创建的attribute。和我们在本节一开始谈到的字段验证attribute类似，可以想象，在很多情况下我们都可以从使用自定义的attribute中受益。我们将要展示的例子来自于NUnit开源工具的启发。

通过NUnit工具可以执行任何程序集中的任何方法从而测试它们。由于没有必要测试一个程序集中的每一个方法，NUnit仅执行那些被标记了`TestAttribute`的方法。

为了实现该特性的一个简化版，我们将做出以下约束。

- 如果没有抛出任何未捕获的异常则认为该方法通过测试。
- 我们定义了一个只能应用于方法的`TestAttribute`。该attribute可以配置方法必须被执行的次数（通过`int`类型的`TestAttribute.nTime`属性）。该attribute还可以用于忽略一个被标记的方法（通过`bool`类型的`TestAttribute.Ignore`属性）。
- `Program.TestAssembly(Assembly)`方法允许执行包含在程序集中所有被标记了`TestAttribute`的方法，而程序集是作为参数传入方法的。出于简单化的考虑，我们假设这些方法都是公有的、非静态的而且不接受任何参数。我们还必须使用后期绑定来访问这些被标记的方法。

下面的程序满足了这些约束。

例7-15

```
using System;
using System.Reflection;

[AttributeUsage(AttributeTargets.Method, AllowMultiple=false)]
public class TestAttribute : System.Attribute {
    public TestAttribute() {
        Console.WriteLine("TestAttribute.ctor() Default ctor.");
    }
    public TestAttribute(int nTime) {
        Console.WriteLine("TestAttribute.ctor(int)");
        m_nTime = nTime;
    }
    private int m_nTime = 1;
    private bool m_Ignore = false;
    public bool Ignore { get { return m_Ignore; } set { m_Ignore = value; } }
    public int nTime { get { return m_nTime; } set { m_nTime = value; } }
}

class Program {
    static void Main() {
        // The programs reflects on itself.
        TestAssembly( Assembly.GetExecutingAssembly() );
    }
}
```

```

}
static void TestAssembly(Assembly assembly) {
    // For each methods of each type declared in 'assembly'.
    foreach( Type type in assembly.GetTypes() ) {
        foreach( MethodInfo method in type.GetMethods() ) {
            // Get attributes of type 'TestAttribute' which mark
            // the method. Trigger the call to 'TestAttribute.ctor()'.
            object[] attributes = method.GetCustomAttributes(
                typeof(TestAttribute),false);
            if( attributes.Length == 1 ) {
                // Get an instance of 'TestAttribute'.
                TestAttribute testAttribute =attributes[0] as TestAttribute;
                // If we shouldn't ignore the method.
                if( ! testAttribute.Ignore ) {
                    object [] parameters = new object[0];
                    object instance = Activator.CreateInstance(type);
                    // Call the method 'nTime' times.
                    for(int i=0;i< testAttribute.nTime ; i++) {
                        try {
                            //Invoke the method with a late binds.
                            method.Invoke(instance,parameters);
                        } catch( TargetInvocationException ex ) {
                            Console.WriteLine(
                                "The method {" + type.FullName + "." +
                                method.Name +
                                "} threw an exception of type " +
                                ex.InnerException.GetType() +
                                " during run #" + (i+1) + "." );
                        } // end catch(...)
                    } // end for(...)
                } // end if( ! attribute.Ignore )
            } // end if( attributes.Length == 1 )
        } // end foreach(MethodInfo...)
    } // end foreach(Type...)
}

}

class Foo {
    [Test()]
    public void Crash() {
        Console.WriteLine("Crash()");
        throw new ApplicationException();
    }
    int state = 0;
    [Test(4)]
    public void CrashTheSecondTime() {
        Console.WriteLine("CrashTheSecondTime()");
        state++;
        if (state == 2) throw new ApplicationException();
    }
    [Test()]
    public void DontCrash () {
        Console.WriteLine("DontCrash");
    }
    [Test(Ignore = true)]
    public void CrashButIgnored() {
        Console.WriteLine("CrashButIngored()");
        throw new ApplicationException();
    }
}

```

```

    }
}

```

该程序输出：

```

TestAttribute.ctor() Default ctor.
Crash()
The method {Foo.Crash} threw an exception of type System.ApplicationExce
ption during run #1.
TestAttribute.ctor(int)
CrashTheSecondTime()
CrashTheSecondTime()
The method {Foo.CrashTheSecondTime} threw an exception of type System.Ap
plicationException during run #2.
CrashTheSecondTime()
CrashTheSecondTime()
TestAttribute.ctor() Default ctor.
DontCrash
TestAttribute.ctor() Default ctor.

```

让我们做一些注解。

- 我们为 `TestAttribute` 类标记了一个 `AttributeUsage` 类型的 `attribute`。我们使用 `AttributeTarget` 枚举量的 `Method` 值告诉编译器 `TestAttribute` 只能被应用在方法上。
- 我们将 `AttributeUsage` 类的 `AllowMultiple` 属性设为 `false` 以表示一个方法不会接受多个 `TestAttribute` 类型的 `attribute`。注意用于初始化 `AllowMultiple` 属性的特殊语法，我们把 `AllowMultiple` 称为有名参数。
- 在为 `Foo.CrashButIgnore()` 方法标记 `TestAttribute` 的时候，我们也使用了有名参数这一语法。
- 当一个异常被引发并且没有在某方法执行期间被捕获，但由于该方法是通过后期绑定调用的，所以此时调用该方法的方法会产生一个 `TargetInvocationException` 类型的异常将原始异常覆盖，而原始异常则被覆盖它的异常的 `InnerException` 属性所引用。
- 为了避免将代码分散到多个程序集中，该程序将进行自我测试（事实上，只有 `Foo` 类中的方法被测试，因为只有它们被标记了 `TestAttribute`）。图7-3是将代码分散后将会出现的程序集组织结构。

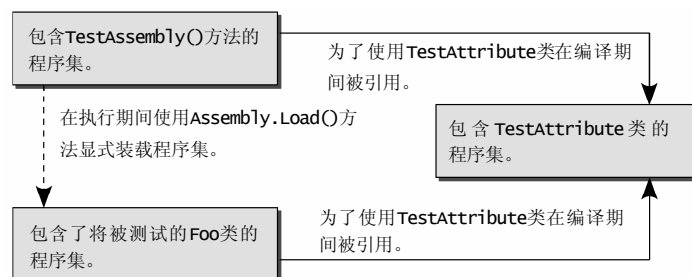


图7-3 程序集组织结构

图7-3与图7-2相似不是偶然的。在两种情况中，都是通过一个在编译期间两者都知道的中介（本例中是一个 `attribute`，而前一个例子中是一个接口）来使用一个在编译期未知的元素。

7.3.7 条件 attribute

C#2引入了条件attribute的概念。一个条件attribute只有在一个符号被定义后才会被编译器考虑到。下面的例子展示了在一个由3个文件生成的项目中使用条件attribute。

例7-16

```
[System.Diagnostics.Conditional("_TEST")]
public class TestAttribute : System.Attribute { }
```

例7-17

```
#define _TEST
[Test] // The compiler marks the Foo1 class with the 'Test' attribute.
class Foo1 { }
```

例7-18

```
#undef _TEST
[Test] // The compiler doesn't mark the Foo2 class with
// the 'Test' attribute.
class Foo2 { }
```

在上一节的示例中，条件attribute被用于从同一段代码产生debug与release版本。在9.2.2节，我们介绍了条件attribute的另一个用处。

7.4 动态生成程序集并在运行中使用

`System.Reflection.Emit`命名空间提供了一套类，通过它们可以在运行时执行另一个程序集期间构造一个新的程序集。除了生成的代码不是C#或C++的形式而是即将使用某个程序集的代码，事实上这就是代码生成。然后，就可以把这个程序集以持久化的方式保存下来。

7.4.1 为什么要考虑动态生成程序集

当使用`System.Reflection.Emit`命名空间中的类构造一个程序集时，代码是用IL语言直接发出的，关于该语言的介绍请参见2.7节。尽管IL语言相对来说比较直接，但是大多数开发者更习惯于C#和VB.NET语言。对于大多数开发者来说，直接产生IL代码远比产生C#源文件后再将其编译难得多。事实上，人们总是倾向于使用诸如C# 或VB.NET这样的结构化语言产生代码。

既然如此，为什么需要动态生成程序集呢？

事实上至少在3个场景下会使用到它，更详细的内容请参见MSDN上名为*Reflection Emits Application Scenarios*的文章。

- 在Web浏览器中执行脚本：主要思想是Web浏览器中的一个脚本动态构造了一个程序集并以持久化的方式保存到客户端。
- 在一个ASP.NET页面中执行脚本：主要思想是某个ASP.NET页面中的一个脚本动态构造了一个程序集并保存在服务器的缓存中。因此，只有第一个访问者会引发程序集的创作。
- 在执行期间编译一个正则表达式。执行正则表达式的问题是这种类型的问题的代表，该类问题有一个相对较慢的通用的解决方案，还有一个更加高效的专用的解决方案。解决这一问题的主要思想就是，当需要提供一个正则表达式时，在执行期间构造一个专用的解决方案，而不是为开发者提供一个较慢的通用的解决方案。.NET Framework 允许编译正则表达式以及以上这些内容，该内容请参见16.7节。在`XsltCompiledTransform`类的实现中能够发现同样的思想（参见21.7节）。

以上内容看上去很抽象，不过我们将展示一个实际的例子，让读者对此有一个更好的理解。

7.4.2 一个实际的问题

1. 简介

我们将在这里展示的例子是建立在对整数系数多项式 P 求值的基础上的。想象一下，一个应用程序在运行时接受了这样一个多项式（例如通过用户）。假设在此之后，应用程序必须在一个很大的取值范围内多次计算该多项式。我们将展示一个通过在运行时动态创建一个新的程序集来解决此类实际问题的优化方案。

现在，让我假设用户输入的多项式 P 如下所示。

$$P(x) = 66x^3 + 83x^2 - 13735x + 30139$$

介绍一点历史知识，以上多项式是数学家Dress和Landreau在1999年发现的。该多项式有一个特点就是，在接受-26~19（包含两者）的整数作为 x 的值输入时它只产生素数结果。当然，我们也可以使用其他任何系数是整数的多项式作为示例。

我们将使用每一个输入对 P 求值1千万次，用来测试性能。我们将只使用4字节的有符号整数（C#中的`int`或`System.Int32`类型）。让我们提醒一下那些对数学有些遗忘的人，最简单（以计算次数为衡量标准）的计算方法就是将 P 重写为以下形式：

$$P(x) = 30139 + x(-13735 + x(83 + 66x))$$

这个技巧被称为Hörner算法。对于一个确定的值 x ，只需要三次乘法与三次加法就可以计算出 P 的值。

2. 解决方案1：通用方法

在不知道该知识的情况下，出现在脑中的简单的解决方案是：

例7-19

```
using System;
using System.Diagnostics;
class Program {
    static int Eval ( int x, int[] Coefs ) {
        int tmp = 0;
        int degree = Coefs.GetLength(0);
        for(int i=degree-1 ; i>= 0 ; i--)
            tmp = Coefs[i]+x*tmp;
        return tmp;
    }
    static void Main() {
        Stopwatch sw = Stopwatch.StartNew();
        int [] Coefs = {30139,-13735,83,66};
        for( int x = -26 ; x<= 19 ; x++ )
            for( int i = 0 ; i<10000000 ; i++)
                Eval (x,Coefs);
        Console.WriteLine("Duration:" + sw.Elapsed );
    }
}
```

由于我们关注的是对性能的测试，就没有必要验证`Eval()`方法的返回结果是否是素数了。

3. 解决方案2: 优化方法

如果我们没有限制必须在`Eval()`方法外定义系数，我们可以将程序改成下面这样。

例7-20

```
using System;
using System.Diagnostics;
```

```

class Program {
    static int Eval( int x ) {
        return 30139-x*(13735-x*(83+x*66));
    }
    static void Main() {
        Stopwatch sw = Stopwatch.StartNew();
        for( int x = -26 ; x<= 19 ; x++ )
            for( int i = 0 ; i<10000000 ; i++)
                Eval(x);
        Console.WriteLine("Duration:" + sw.Elapsed );
    }
}

```

4. 两种解决方案之间的性能比较

第一个解决方案在参考机器上花费的时间是17.81秒。在同一台机器上，第二个解决方案（经过优化的）花费的时间是3.87秒，因此与第一个解决方案相比性能提升了大约4.6倍。一些机器指令，比如在表上传递引用，在表中查找系数以及管理循环，在第二个解决方案中都没有出现。这些结果表明针对特定多项式的优化算法比使用通用算法更加高效。

有两个问题可能最终导致这些性能结果失真。

- 潜在问题1：垃圾回收器可能在其中的某个测试中激活自己而在其他测试中并不活动。我们可以证实垃圾回收器没有触发自身，因为在这些程序只使用了很少的内存。
- 潜在问题2：JIT编译器可能足够的“智能”以至于注意到根本不值得调用Eval()方法，因为我们并不使用该方法的结果。我们在Eval()方法中加入了一段递增全局计数器的代码，并验证了修改后的性能结果与原来的类似。由此证实上述假设并不成立。

7.4.3 理想的第三种解决方案——动态创建程序集

优化方法和通用方法共同的问题在于，Eval()方法中的系数必须在编译期就已知。我们只能在编译期提供多项式，而不是在执行期。通过动态构造程序集，我们将不仅能够使用更加高效的优化方法，还能够在运行时接受多项式的系数。事实上，我们只需要根据优化方法产生Eval()方法的IL代码。

针对上节给出的例子中的多项式，C#编译器产生的IL代码如下所示。

```

.method private hidebysig static int32 Eval(int32 x) cil managed
{
    // Code size          28 (0x1c)
    .maxstack 7
    .locals init ([0] int32 CS$000000003$000000000)
    IL_0000: ldc.i4      0x75bb      // coef 30139 push on top of the stack
    IL_0005: ldarg.0          // x value push on top of the stack
    IL_0006: ldc.i4      0xffffca59  // coef -13735 push on top of the stack
    IL_000b: ldarg.0          // x value push on top of the stack
    IL_000c: ldc.i4.s     83          // coef 83 push on top of the stack
    IL_000e: ldarg.0          // x value push on top of the stack
    IL_000f: ldc.i4.s     66          // coef 66 push on top of the stack
    IL_0011: mul              //
    IL_0012: add              //      Computation of the polynomial
    IL_0013: mul              //      value for x with
    IL_0014: add              //      three additions
    IL_0015: mul              //      and three multiplications.
    IL_0016: add              //
    IL_0017: stloc.0
    IL_0018: br.s          IL_001a
    IL_001a: ldloc.0
    IL_001b: ret
}

```

```
} // end of method Program::Calc
```

现在我们只需能够使用 `System.Reflection.Emit` 命名空间中的类为任意的多项式产生这样的 IL 代码。



在本例中，`ldarg` IL 指令使用参数 0 装载第一个方法参数。如果该方法不是静态方法，`ldarg.0` 将代表 `this` 引用，并且我们需要使用 `ldarg.1` 来访问第一个方法参数。从这点来看，对于那些不再是静态方法的方法，我们将使用 `ldarg.1` 去访问第一个方法元素。

下面是代码：

例7-21

```
using System;
using System.Reflection;
using System.Reflection.Emit;
using System.Threading;
using System.Diagnostics;

public interface IPolynome {
    int Eval(int x);
}

class Polynome {
    public IPolynome polynome;
    public Polynome(int[] coefs) {
        Assembly asm = BuildCodeInternal(coefs);
        polynome = (IPolynome)asm.CreateInstance("PolynomeInternal");
    }
    private Assembly BuildCodeInternal(int[] coefs) {
        AssemblyName asmName = new AssemblyName();
        asmName.Name = "EvalPolAsm";

        // Build an assembly dynamically.
        AssemblyBuilder asmBuilder =
            Thread.GetDomain().DefineDynamicAssembly(
                asmName, AssemblyBuilderAccess.Run );

        // Add a module in the assembly.
        ModuleBuilder modBuilder =
            asmBuilder.DefineDynamicModule("MainMod");

        // Add the 'PolynomeInternal' class in the module.
        TypeBuilder typeBuilder = modBuilder.DefineType(
            "PolynomeInternal", TypeAttributes.Public);
        typeBuilder.AddInterfaceImplementation(typeof(IPolynome));

        // Implement the int Eval(int) method
        MethodBuilder methodBuilder = typeBuilder.DefineMethod(
            "Eval",
            MethodAttributes.Public | MethodAttributes.Virtual,
            typeof(int), // return type
            new Type[] { typeof(int) }); // argument

        // Generate the IL code from the table of coefs.
        ILGenerator ilGen = methodBuilder.GetILGenerator();
        int deg = coefs.GetLength(0);
        for (int i = 0; i < deg - 1; i++) {
            ilGen.Emit(OpCodes.Ldc_I4, coefs[i]);
```

```

        ilGen.Emit(OpCodes.Ldarg, 1);
    }
    ilGen.Emit(OpCodes.Ldc_I4, coeffs[deg - 1]);
    for (int i = 0; i < deg - 1; i++) {
        ilGen.Emit(OpCodes.Mul);
        ilGen.Emit(OpCodes.Add);
    }
    ilGen.Emit(OpCodes.Ret);

    // Indicate that the 'Eval()' method is an implementation...
    // ... of the interface method 'IPolynome.Eval()'.
    MethodInfo methodInfo = typeof(IPolynome).GetMethod("Eval");
    typeBuilder.DefineMethodOverride(methodBuilder, methodInfo);
    typeBuilder.CreateType();
    return asmBuilder;
}
}

class Program {
    static void Main() {
        Stopwatch sw = Stopwatch.StartNew();
        int[] coeffs = { 30139, -13735, 83, 66 };
        Polynome p = new Polynome(coeffs);
        for (int x = -26; x <= 19; x++)
            for (int i = 0; i < 10000000; i++)
                p.polynome.Eval(x);
        Console.WriteLine("Duration:" + sw.Elapsed);
    }
}

```

1. 将第三种解决方案与解决方案1、2进行比较

该解决方案在参考机器上花费的时间是4.36秒。这大概比第二个版本慢了1.12倍，而且创建程序集的时间并没有被计算进去，因为它需要花费200秒~400秒。我们可以猜测这个小的性能损失是由于使用了一个非静态方法，因为这会迫使我们在每次调用**Eval()**的时候都要传入一个实例的引用。此外，调用一个定义在接口中的方法比调用一个非虚方法或抽象方法的代价要稍微昂贵一些。事实上，**callvirt** IL指令用来调用接口中的方法，而**call**指令用来调用静态方法。因此，相比**callvirt**指令JIT编译器会为**call**指令产生消耗较少时钟周期的代码。这个小的性能损失也可以归咎于我们没有考虑到多项式的某些系数可能被保存在一个字节中（在这里是83与66）。因此，编译器会针对这个情况使用为操作那些保存整型数的字节而专门优化的指令。

尽管如此，在遵守同样的约束的情况下（即多项式的系数在**Eval()**方法之外定义），第三种方案仍然比第一种方案快了4.1倍。这是从动态创建程序集中直接获得的巨大收获。如果考虑到**null**系数，我们还可以进一步优化代码。

2. 对代码的技术描述

上例中值得注意的类型有：

```

AssemblyName
AssemblyBuilder
ModuleBuilder
TypeBuilder
MethodBuilder
ILGenerator
MethodInfo

```

它们通常按以上顺序使用，与我们在例子中所采用的方式一样。**MSDN**上还详细记载了一些其他类

型。这些其他类型主要负责管理异常，分支管理以及类中其他元素（属性、事件）的管理。

7.4.4 将程序集保存到磁盘上的能力

如果想将程序集保存到磁盘上，你可能需要：

- ❑ 为模块命名（例如MainMod.dll）。
- ❑ 使用AssemblyBuilderAccess.RunAndSave代替AssemblyBuilderAccess.Run。
- ❑ 在BuildCodeInternal()方法返回前，使用以下代码保存程序集。

```
TheAsm.Save("MainMod.dll");
```

我们可以比较C#编译器产生的IL代码与我们的程序产生的IL代码。

区别来自于以下事实。

- ❑ 我们动态生成了一个非静态方法。
- ❑ 与C#编译器产生的代码相比，我们没有使用本地变量。
- ❑ 我们没有使用ldc.i4.s优化指令，该指令在栈上装载了一个4字节的有符号整数（-128和127之间的整数）。我们可以修改程序以使用该指令。

7.4.5 结论

在这里我们看到了一个利用动态创建程序集进行编程的代表性的例子。我们可以很容易地改造这个方法，将其用于其他与向量计算一样有用的领域（矩阵组合，二次方程求值等）。常会遇到同一个事先未知的矩阵进行了数百万次组合（例如在3D场景中计算点的位置）。

大多数开发者可能从不需要去动态地创建一个程序集，但是某些项目可以从中受益很多。