Visual Studio Magazine

Generate Assemblies With Reflection Emit

Leverage reflection emit in the .NET Framework to generate and execute code dynamically in your applications.

by Randy Holloway

October 2002 Issue

Technology Toolbox: VB.NET, C#

The .NET Framework's Reflection namespace lets you view assembly metadata and create assemblies at run time. Reflection lets your code do type discovery, view assembly metadata, invoke assembly code dynamically at run time, and more.

Reflection emit takes these features to the next level, including building assemblies and creating new types at run time. You can define assemblies to run dynamically or be saved to disk. You can also define the modules and types within the assembly, along with their methods.

Most developers reserve reflection emit for building compilers and other complex tasks. However, you can use it for less exotic purposes, such as building flexible application features that leverage code generation at run time. You can also use it for apps that require extensive calculations based on user inputs or other variables defined at run time. Reflection emit lets you optimize emitted Microsoft Intermediate Language (MSIL) instructions for the specific calculation at hand. The emit technique provides the flexibility you need to generate the necessary code dynamically in an optimized manner.

You can combine the features of reflection and reflection emit to build an assembly at run time, derive a type reference from within the assembly, determine what methods of that type are available, and execute one of those methods. This technique lets you generate types at run time to perform functions or calculations that benefit from optimized code generation. You can also apply this technique to the runtime generation of code for performance monitoring and code profiling.

You need to understand the System.Reflection and System.Reflection.Emit namespaces to apply this technique effectively, along with the fundamentals of using the reflection emit technique for generating MSIL code. I'll show you how this all works by exploring the assembly creation process with reflection emit. Then I'll demonstrate how to use reflection for dynamic invocation.

Use the System.Reflection.Emit namespace's AssemblyBuilder class to dynamically construct assemblies containing different types comprising different modules. You can then derive type references from these assemblies and create objects. For example, you can emit a summation calculator using the SummationEmit function (see Listing 1). You instantiate the AssemblyBuilder class through the execution of the DefineDynamicAssembly method of the AppDomain class. This creates an assembly with a defined name and an AssemblyBuilderAccess enumeration value. The value determines whether the assembly runs, is saved, or both. You can create dynamic transient assemblies, as well as assemblies that are persisted to disk. This distinction matters for some applications, depending on the overhead required to create the assembly and future uses you might have in mind for it.

Once you create an assembly instance, you can call the DefineDynamicModule method to create a ModuleBuilder instance. The instance defines and represents an assembly module. Defining this module enables you to execute the DefineType method, which creates an instance of the TypeBuilder class. You must create the assembly, module definition, and type creation before you can define the functions in your code.

Define Your Type's Methods

Once you complete these steps, you can define the methods exposed by your types. Create a method in C# using the TypeBuilder's DefineMethod to create an instance of the MethodBuilder class:

```
Type[] paramTypes = new Type[0];
Type returnType = typeof(int);
MethodBuilder mthbSummation =
   typbSummation.DefineMethod
   ("CalculateSummation",
   MethodAttributes.Public |
   MethodAttributes.Virtual,
   returnType, paramTypes);
```

This code creates a method, assigns a method name, assigns method attributes based on values in the MethodAttributes enumeration, and defines the input and output parameters based on the types assigned. Continue with the summation calculator example by defining a method named CalculateSummation, with MethodAttributes assignments of public and virtual. Next, define the input and output parameter types. You use no input parameters for the function; assign the parameter inputs as a Type array with no elements. Now that you've built a method for this type, you can generate supporting code.

Generate code for this method using the ILGenerator class to emit MSIL instructions (also known as OpCodes). Create an instance of ILGenerator using the MethodBuilder class's GetILGenerator method. The ILGenerator lets you use the Emit method for instructions your .NET-supported language's compiler would generate ordinarily. This helps you in developing a high-performance custom class based on values determined at run time. In the example, emit the code for a simple summation of integers from 1 to 5. Define the summation as the sum of n over the range n = 1 to n = 5:

```
ilgSummation.Emit(OpCodes.Ldc_I4,
   0);
for (int i = 1; i <=
   iSummationSeed; i++)
{
ilgSummation.Emit(OpCodes.Ldc_I4,
   i);
ilgSummation.Emit(OpCodes.Add);
}
ilgSummation.Emit(OpCodes.Ret);
```

This code demonstrates the use of a loop to emit a series of Ldc_I4 instructions with an integer parameter, along with instructions for the Add OpCode. Use the Ldc_I4 OpCode to push a supplied int32 value onto an evaluation stack. Use the subsequent Add OpCode to add the values pushed onto the stack. Then use the Ret OpCode for a return instruction from the current method. Once you build a method and its instructions are emitted, you can create an instance of this type, then save it to disk along with its containing assembly. As you might expect, you use the TypeBuilder class's CreateType method to create the type. Then you use the AssemblyBuilder's Save method to save the assembly.

In the example, you define a single method and override; however, you can use the TypeBuilder class to build constructors, define events, and perform virtually all functions related to type definition. The ReflectionEmit driver application has references to both the RESummation (C#) and

RESummationVB components. When you invoke these components through the driver application, it emits an assembly in the application directory containing the code emitted using the ILGenerator (see Figure 1 for a view of the assembly generated, using the ILDASM utility with the .NET Framework).

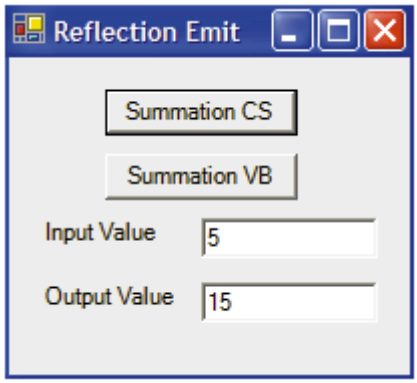

Figure 1. Drive the Reflection Emit Classes.

The assembly generation process produces an assembly containing defined types and their methods, all based on MSIL code (see Listing 2). The MSIL defines the CalculateSummation method, including the OpCodes defined for the stack push and the add functions performed. This method also defines the method override using the ISummation interface. You should now agree that the reflection emit API provides a rich set of instructions for generating assemblies.

Invoke Dynamically With Reflection

The System.Reflection namespace hosts a number of classes supporting reflection functionality—notably, the Assembly class. Use this class for implementing type discovery, invoking assembly code dynamically, and viewing assembly metadata at run time. The Assembly class exposes various methods for loading assemblies, enabling you to reference a new assembly in your app at run time. You can use an instance of the Assembly object (once it's loaded) to query for assembly manifest data, determine available types for reference, and view metadata within that unit of app code. You'll use this in application development, especially when all the components aren't available at compile time (see the sidebar, "Load an Assembly at Run Time").

Create an instance of the GenerateSummation type created at run time by calling the asmSummation object's CreateInstance method:

```
int i;
ISummation SummationCalculator =
   null;
Assembly asmSummation =
   SummationEmit(iSummationSeed);
SummationCalculator = (ISummation)asmSummation.
CreateInstance("GenerateSummation");
i = SummationCalculator.
   CalculateSummation();
```

The assembly returned from the SummationEmit function contains the GenerateSummation type with the CalculateSummation method. The CalculateSummation code is a summation based on the input integer iSummationSeed. The SummationCalculator is an instance of the GenerateSummation type implementing the ISummation interface, returning a value from the CalculateSummation method. Using reflection to derive a type reference dynamically lets your code execute at run time.

You can build functionality into your apps to support emitting custom performance monitoring code:

```
PerformanceCounter pcFreeMemory =
   new PerformanceCounter("Memory",
   "Available MBytes");
```

This code creates an instance of the PerformanceCounter class using its constructor, associating the instance with the free memory performance counter. Here's the associated code in MSIL:

```
IL_0000:   ldstr   "Memory"
IL_0005:   ldstr   "Available MBytes"
IL_000a: newobj   instance void
[System]System.Diagnostics.
   PerformanceCounter::.ctor(string,
```

```
    string)
```

You could also implement this kind of performance monitoring code support at design time using C#, or at run time using C# and the CodeDOM. However, using the emit technique adds flexibility in optimizing the code based on user input or environmental variables. Also, you can boost performance with calculation-intensive code. The emit techniques I've demonstrated let you emit the MSIL code needed to implement the PerformanceCounter class (or another class exposed through the .NET Framework) to leverage that functionality dynamically in your apps.

Dynamic generation and execution of code at run time using reflection emit can be a powerful technique. I admit that its applications are highly specialized and might not be valuable to most developers, but if you're up to the challenge, you can apply dynamic code generation to more mainstream uses than generating mathematical algorithms—especially if they require recursion and the code exhibits performance reduction as the number of iterations increase. Using these techniques can create more elegant solutions that boost performance and increase flexibility.

About the Author

Randy Holloway is the founder of Winformation Systems, a consulting and training initiative specializing in Web services and Windows-based enterprise system development. Randy also has expertise in digital imaging. He speaks at development conferences and writes for *VSM*, *XML & Web Services Magazine*, and more. Contact him at randyholloway@winformationsystems.com.