How to dynamically call a method in C#? [duplicate]

Asked 15 years, 1 month ago Modified 5 years, 6 months ago Viewed 76k times

41

This question already has answers here:

Calling a function from a string in C# (5 answers)

Closed 4 years ago.

I have a method:

```
add(int x,int y)
```

I also have:

```
int a = 5;
int b = 6;
string s = "add";
```

Is it possible to call `add(a,b)` using the string `s` ?

c#

Share Edit Follow Flag

edited Feb 21, 2020 at 23:47
kingsfoil
3,895 7 37 60

asked Jul 15, 2010 at 10:49
scatman
14.6k 23 72 94

8 `if (s == "add") { add(a,b); }` This? – dtb Jul 15, 2010 at 10:50

5 Answers

Sorted by: Reset to default Trending (recent votes count more)

78

how can i do this in c#?

Using reflection.

`add` has to be a member of some type, so (cutting out a lot of detail):

```
typeof(MyType).GetMethod("add").Invoke(null, new [] {arg1, arg2})
```

This assumes `add` is static (otherwise first argument to `Invoke` is the object) and I don't need extra parameters to uniquely identify the method in the `GetMethod` call.

Share Edit Follow Flag

answered Jul 15, 2010 at 10:54

Richard
110k 21 214 279

3 Comments

Add a comment

Richard Over a year ago

@DavidStratton It will work with `private` members: use one of the overloads of `GetMethod` that takes a `BindingFlags` argument with `BindingFlags.NonPublic` .

2 Reply •••

Sreekumar P Over a year ago

Can we use "Func<>" instead of Refection ?

2 Reply •••

Richard Over a year ago

@Sreekumar No, because to create a lambda you either need to fix at compile time or build expression trees. The latter done dynamically will need to use reflection.

1 Reply •••

Add a comment

28

Use reflection - try the `Type.GetMethod` Method

Something like

```
MethodInfo addMethod = this.GetType().GetMethod("add");
object result = addMethod.Invoke(this, new object[] { x, y } );
```

You lose strong typing and compile-time checking - invoke doesn't know how many parameters the method expects, and what their types are and what the actual type of the return value is. So things could fail at runtime if you don't get it right.

It's also slower.

Share Edit Follow Flag

edited Jul 15, 2010 at 11:04

answered Jul 15, 2010 at 10:53

Anthony
5,226 6 68 87

2 Comments

Add a comment

Joker_37 Over a year ago

any idea how can I make this call as await

0 Reply ...

tblev Over a year ago

Only problem I had was `this` was `null`

0 Reply ...

17

If the functions are known at compile time and you just want to avoid writing a switch statement.

Setup:

```
Dictionary<string, Func<int, int, int>> functions =
  new Dictionary<string, Func<int, int, int>>();

functions["add"] = this.add;
functions["subtract"] = this.subtract;
```

Called by:

```
string functionName = "add";
int x = 1;
int y = 2;

int z = functions[functionName](x, y);
```

Share Edit Follow Flag

answered Jul 15, 2010 at 11:52

Amy B

110k 21 139 191

Comments

Add a comment

11

You can use reflection.

```
using System;
using System.Reflection;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Program p = new Program();
            Type t = p.GetType();
```

```
            MethodInfo mi = t.GetMethod("add", BindingFlags.NonPublic |
BindingFlags.Instance);
            string result = mi.Invoke(p, new object[] {4, 5}).ToString();
            Console.WriteLine("Result = " + result);
            Console.ReadLine();
        }

        private int add(int x, int y)
        {
            return x + y;
        }
    }
}
```

Share Edit Follow Flag

answered Jul 15, 2010 at 10:57

Itay Karo
18.4k 4 43 59

Comments

Add a comment

Vote

@Richard's answer is great. Just to expand it a bit:

This can be useful in a situation where you dynamically created an object of unknown type and need to call its method:

```
var do = xs.Deserialize(new XmlTextReader(ms)); // example - XML deserialization
do.GetType().GetMethod("myMethodName").Invoke(do, new [] {arg1, arg2});
```

becasue at compile time `do` is just an `Object`.

Share Edit Follow Flag

answered Jul 4, 2016 at 14:23

ajeh
2,794 2 37 66

3 Comments

Add a comment

Joker_37 Over a year ago

How can I make this method awaitable?

0 Reply •••

ajeh Over a year ago

What issue are you facing, please elaborate.

0 Reply •••

Joker_37 Over a year ago

I wanted to call this method asynchronously. I found solution. I type casted it as task like this. `await (Task)do.GetType().GetMethod("myMethodName").Invoke(do, new [] {arg1, arg2});`

1 Reply •••

Add a comment

Call dynamic method from string

Asked 10 years, 3 months ago Modified 6 years, 5 months ago Viewed 5k times

4

I'm trying to call a method from a dynamic without knowing its name. I have difficulties to explain this in english so there's the code:

```
public void CallMethod(dynamic d, string n)
{
    // Here I want to call the method named n in the dynamic d
}
```

I want something like: `d.n()` but with *n* replaced by the string.

I want this :

```
Type thisType = this.GetType();
MethodInfo theMethod = thisType.GetMethod(TheCommandString);
theMethod.Invoke(this, userParameters);
```

but with **dynamic**.

If you need the context to help you: I'm make an application that's support "mods", you put DLLs in the mod folder and it loads it and execute it. It works with **dynamic** (I have a dictionnary like this : `Dictionnary<string, dynamic> instances;`). I want the application to get the methods name from the library (with `instances["topkek"].GetMethods();` , I've already made this method) but then call the method with the string it returns. I don't know if what I said mean something (I'm french :/)...

I'm using VS 2013 Express with the .Net framework 4.5, if you need more information to help me ask me.

c# dynamic methods reflection

Share Edit Follow Flag

edited Mar 14, 2016 at 21:54

asked May 17, 2015 at 16:02

SwagLordKnight
53 1 8

How will you know which method to call? You can use `Type.GetMethods` on a 'normal' `Type` - not sure how it'll work with `dynamic` . – Wai Ha Lee May 17, 2015 at 16:05

Your question is too broad and provides a very little detail. Please try to put in some effort and explain yourself. – Yuval Itzchakov May 17, 2015 at 16:08

The method is specified by the user, I want something like: "d.n()", ofc if the method doesn't exists it will crash. So this isn't possible with dynamic ? – SwagLordKnight May 17, 2015 at 16:09

1 Why not pass a delegate instead? – Yuval Itzchakov May 17, 2015 at 16:10

Look into reflection, here is a tutorial tutorialspoint.com/csharp/csharp_reflection.htm – faljbour May 17, 2015 at 16:11

3 Answers

Sorted by: Reset to default Trending (recent votes count more)

8

you can write your method as follows -

```
public void CallMethod(dynamic d, string n)
    {
        d.GetType().GetMethod(n).Invoke(d, null);
    }
```

Share Edit Follow Flag

answered May 17, 2015 at 23:14

Sanjay Singh
714 2 8 15

Comments

Add a comment

1

I want to add another approach as solution:

In your case, the caller (developer of the mod) knows the method to call. Thus, this might helpful:

```
// In the main application:
public dynamic PerformMethodCall(dynamic obj, Func<dynamic, dynamic> method)
{
    return method(obj);
{


 // In a mod:
 mainProgram.PerformMethodCall(myDynamicObj, n => n.myDynamicMethod());

 // In another mod:
 mainProgram.PerformMethodCall(myDynamicObj, n => n.anotherMethod());
```

This is a further development of the idea of Yuval Itzchakov in his commentary. He had suggested using a delegate.

Share Edit Follow Flag

answered Mar 18, 2019 at 12:31

SommerEngineering
1,736 1 22 28

Comments

Add a comment

2

If all methods are void, this could work. Otherwise you need to change it a bit.

```
    public void CallMethod(string className, string methodName)
    {
        object dynamicObject;
        // Here I want to call the method named n in the dynamic d
        string objectClass = "yourNamespace.yourFolder." + className;
        Type objectType = Type.GetType(objectClass);
        if (objectType == null)
        {
            // Handle here unknown dynamic objects
        }
        else
        {
            // Call here the desired method
            dynamicObject = Activator.CreateInstance(objectType);
            System.Reflection.MethodInfo method =
objectType.GetMethod(methodName);
            if (method == null)
            {
                // Handle here unknown method for the known dynamic object
            }
            else
            {
                object[] parameters = new object[] { };   // No parameters
                method.Invoke(dynamicObject, parameters);
            }
        }
    }
```

Share Edit Follow Flag

edited May 18, 2015 at 8:43

answered May 17, 2015 at 16:27

Blas Soriano
566 3 16

6 Comments

Add a comment

SwagLordKnight Over a year ago

Thanks but I don't want to create a new instance of my type, the instance is already the variable "d".

Blas Soriano Over a year ago

@SwagLordKnight Then it is even easier, you can skip the part to build the objectClass. Check link

user57508 Over a year ago

could you please name your parameters in a more descriptive way?

1 Reply •••

Blas Soriano Over a year ago

@AndreasNiedermair parameters is an array of object, to hold the parameters to be passed to the method. For example, if the method expect a string and an int, inside the brackets { } you need to add these 2 parameters: parameters = new object[] { str, number } where str and number must be defined somewhere previously.

0 Reply •••

Blas Soriano Over a year ago

@AndreasNiedermair I agree. I should keep good practices instead of copying names from the original question. Thanks for pointing it; updated.

1 Reply •••

Add a comment | Show 1 more comment

Start asking to get answers

Find the answer to your question by asking.

Ask question

Explore related questions

c# dynamic methods reflection

See similar questions with these tags.

How to: Define and execute dynamic methods

03/30/2024

The following procedures show how to define and execute a simple dynamic method and a dynamic method bound to an instance of a class. For more information on dynamic methods, see the DynamicMethod class.

1. Declare a delegate type to execute the method. Consider using a generic delegate to minimize the number of delegate types you need to declare. The following code declares two delegate types that could be used for the `SquareIt` method, and one of them is generic.

   C#

   ```
   private delegate long SquareItInvoker(int input);

   private delegate TReturn OneParameter<TReturn, TParameter0>
       (TParameter0 p0);
   ```

2. Create an array that specifies the parameter types for the dynamic method. In this example, the only parameter is an `int` (`Integer` in Visual Basic), so the array has only one element.

   C#

   ```
   Type[] methodArgs = {typeof(int)};
   ```

3. Create a DynamicMethod. In this example the method is named `SquareIt`.

   > **Note**
   >
   > It is not necessary to give dynamic methods names, and they cannot be invoked by name. Multiple dynamic methods can have the same name. However, the name appears in call stacks and can be useful for debugging.

   The type of the return value is specified as `long`. The method is associated with the module that contains the `Example` class, which contains the example code. Any loaded module could be specified. The dynamic method acts like a module-level `static` method (`Shared` in Visual Basic).

C#

```
DynamicMethod squareIt = new DynamicMethod(
    "SquareIt",
    typeof(long),
    methodArgs,
    typeof(Example).Module);
```

4. Emit the method body. In this example, an ILGenerator object is used to emit the common intermediate language (CIL). Alternatively, a DynamicILInfo object can be used in conjunction with unmanaged code generators to emit the method body for a DynamicMethod.

   The CIL in this example loads the argument, which is an `int`, onto the stack, converts it to a `long`, duplicates the `long`, and multiplies the two numbers. This leaves the squared result on the stack, and all the method has to do is return.

   C#

   ```
   ILGenerator il = squareIt.GetILGenerator();
   il.Emit(OpCodes.Ldarg_0);
   il.Emit(OpCodes.Conv_I8);
   il.Emit(OpCodes.Dup);
   il.Emit(OpCodes.Mul);
   il.Emit(OpCodes.Ret);
   ```

5. Create an instance of the delegate (declared in step 1) that represents the dynamic method by calling the CreateDelegate method. Creating the delegate completes the method, and any further attempts to change the method — for example, adding more CIL — are ignored. The following code creates the delegate and invokes it, using a generic delegate.

   C#

   ```
   OneParameter<long, int> invokeSquareIt =
       (OneParameter<long, int>)
       squareIt.CreateDelegate(typeof(OneParameter<long, int>));

   Console.WriteLine($"123456789 squared = {invokeSquareIt(123456789)}");
   ```

6. Declare a delegate type to execute the method. Consider using a generic delegate to minimize the number of delegate types you need to declare. The following code declares a generic delegate type that can be used to execute any method with one parameter and a return value, or a method with two parameters and a return value if the delegate is bound to an object.

C#

```
private delegate TReturn OneParameter<TReturn, TParameter0>
    (TParameter0 p0);
```

7. Create an array that specifies the parameter types for the dynamic method. If the delegate representing the method is to be bound to an object, the first parameter must match the type the delegate is bound to. In this example, there are two parameters, of type `Example` and type `int` (`Integer` in Visual Basic).

C#

```
Type[] methodArgs2 = { typeof(Example), typeof(int) };
```

8. Create a DynamicMethod. In this example the method has no name. The type of the return value is specified as `int` (`Integer` in Visual Basic). The method has access to the private and protected members of the `Example` class.

C#

```
DynamicMethod multiplyHidden = new DynamicMethod(
    "",
    typeof(int),
    methodArgs2,
    typeof(Example));
```

9. Emit the method body. In this example, an ILGenerator object is used to emit the common intermediate language (CIL). Alternatively, a DynamicILInfo object can be used in conjunction with unmanaged code generators to emit the method body for a DynamicMethod.

   The CIL in this example loads the first argument, which is an instance of the `Example` class, and uses it to load the value of a private instance field of type `int`. The second argument is loaded, and the two numbers are multiplied. If the result is larger than `int`, the value is truncated and the most significant bits are discarded. The method returns, with the return value on the stack.

C#

```
ILGenerator ilMH = multiplyHidden.GetILGenerator();
ilMH.Emit(OpCodes.Ldarg_0);

FieldInfo testInfo = typeof(Example).GetField("test",
    BindingFlags.NonPublic | BindingFlags.Instance);
```

```
ilMH.Emit(OpCodes.Ldfld, testInfo);
ilMH.Emit(OpCodes.Ldarg_1);
ilMH.Emit(OpCodes.Mul);
ilMH.Emit(OpCodes.Ret);
```

10. Create an instance of the delegate (declared in step 1) that represents the dynamic method by calling the CreateDelegate(Type, Object) method overload. Creating the delegate completes the method, and any further attempts to change the method—for example, adding more CIL—are ignored.

> **Note**
>
> You can call the **CreateDelegate** method multiple times to create delegates bound to other instances of the target type.

The following code binds the method to a new instance of the `Example` class whose private test field is set to 42. That is, each time the delegate is invoked the instance of `Example` is passed to the first parameter of the method.

The delegate `OneParameter` is used because the first parameter of the method always receives the instance of `Example`. When the delegate is invoked, only the second parameter is required.

C#

```
OneParameter<int, int> invoke = (OneParameter<int, int>)
    multiplyHidden.CreateDelegate(
        typeof(OneParameter<int, int>),
        new Example(42)
    );

Console.WriteLine($"3 * test = {invoke(3)}");
```

Example

The following code example demonstrates a simple dynamic method and a dynamic method bound to an instance of a class.

The simple dynamic method takes one argument, a 32-bit integer, and returns the 64-bit square of that integer. A generic delegate is used to invoke the method.

The second dynamic method has two parameters, of type `Example` and type `int` (`Integer` in Visual Basic). When the dynamic method has been created, it is bound to an instance of `Example`, using a generic delegate that has one argument of type `int`. The delegate does not have an argument of type `Example` because the first parameter of the method always receives the bound instance of `Example`. When the delegate is invoked, only the `int` argument is supplied. This dynamic method accesses a private field of the `Example` class and returns the product of the private field and the `int` argument.

The code example defines delegates that can be used to execute the methods.

C#

```
using System;
using System.Reflection;
using System.Reflection.Emit;

public class Example
{
    // The following constructor and private field are used to
    // demonstrate a method bound to an object.
    private int test;
    public Example(int test) { this.test = test; }

    // Declare delegates that can be used to execute the completed
    // SquareIt dynamic method. The OneParameter delegate can be
    // used to execute any method with one parameter and a return
    // value, or a method with two parameters and a return value
    // if the delegate is bound to an object.
    //
    private delegate long SquareItInvoker(int input);

    private delegate TReturn OneParameter<TReturn, TParameter0>
        (TParameter0 p0);

    public static void Main()
    {
        // Example 1: A simple dynamic method.
        //
        // Create an array that specifies the parameter types for the
        // dynamic method. In this example the only parameter is an
        // int, so the array has only one element.
        //
        Type[] methodArgs = {typeof(int)};

        // Create a DynamicMethod. In this example the method is
        // named SquareIt. It is not necessary to give dynamic
        // methods names. They cannot be invoked by name, and two
        // dynamic methods can have the same name. However, the
        // name appears in calls stacks and can be useful for
        // debugging.
        //
```

```
        // In this example the return type of the dynamic method
        // is long. The method is associated with the module that
        // contains the Example class. Any loaded module could be
        // specified. The dynamic method is like a module-level
        // static method.
        //
        DynamicMethod squareIt = new DynamicMethod(
            "SquareIt",
            typeof(long),
            methodArgs,
            typeof(Example).Module);

        // Emit the method body. In this example ILGenerator is used
        // to emit the MSIL. DynamicMethod has an associated type
        // DynamicILInfo that can be used in conjunction with
        // unmanaged code generators.
        //
        // The MSIL loads the argument, which is an int, onto the
        // stack, converts the int to a long, duplicates the top
        // item on the stack, and multiplies the top two items on the
        // stack. This leaves the squared number on the stack, and
        // all the method has to do is return.
        //
        ILGenerator il = squareIt.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0);
        il.Emit(OpCodes.Conv_I8);
        il.Emit(OpCodes.Dup);
        il.Emit(OpCodes.Mul);
        il.Emit(OpCodes.Ret);

        // Create a delegate that represents the dynamic method.
        // Creating the delegate completes the method, and any further
        // attempts to change the method (for example, by adding more
        // MSIL) are ignored. The following code uses a generic
        // delegate that can produce delegate types matching any
        // single-parameter method that has a return type.
        //
        OneParameter<long, int> invokeSquareIt =
            (OneParameter<long, int>)
            squareIt.CreateDelegate(typeof(OneParameter<long, int>));

        Console.WriteLine($"123456789 squared = {invokeSquareIt(123456789)}");

        // Example 2: A dynamic method bound to an instance.
        //
        // Create an array that specifies the parameter types for a
        // dynamic method. If the delegate representing the method
        // is to be bound to an object, the first parameter must
        // match the type the delegate is bound to. In the following
        // code the bound instance is of the Example class.
        //
        Type[] methodArgs2 = { typeof(Example), typeof(int) };

        // Create a DynamicMethod. In this example the method has no
        // name. The return type of the method is int. The method
```

```
// has access to the protected and private data of the
// Example class.
//
DynamicMethod multiplyHidden = new DynamicMethod(
    "",
    typeof(int),
    methodArgs2,
    typeof(Example));

// Emit the method body. In this example ILGenerator is used
// to emit the MSIL. DynamicMethod has an associated type
// DynamicILInfo that can be used in conjunction with
// unmanaged code generators.
//
// The MSIL loads the first argument, which is an instance of
// the Example class, and uses it to load the value of a
// private instance field of type int. The second argument is
// loaded, and the two numbers are multiplied. If the result
// is larger than int, the value is truncated and the most
// significant bits are discarded. The method returns, with
// the return value on the stack.
//
ILGenerator ilMH = multiplyHidden.GetILGenerator();
ilMH.Emit(OpCodes.Ldarg_0);

FieldInfo testInfo = typeof(Example).GetField("test",
    BindingFlags.NonPublic | BindingFlags.Instance);

ilMH.Emit(OpCodes.Ldfld, testInfo);
ilMH.Emit(OpCodes.Ldarg_1);
ilMH.Emit(OpCodes.Mul);
ilMH.Emit(OpCodes.Ret);

// Create a delegate that represents the dynamic method.
// Creating the delegate completes the method, and any further
// attempts to change the method — for example, by adding more
// MSIL — are ignored.
//
// The following code binds the method to a new instance
// of the Example class whose private test field is set to 42.
// That is, each time the delegate is invoked the instance of
// Example is passed to the first parameter of the method.
//
// The delegate OneParameter is used, because the first
// parameter of the method receives the instance of Example.
// When the delegate is invoked, only the second parameter is
// required.
//
OneParameter<int, int> invoke = (OneParameter<int, int>)
    multiplyHidden.CreateDelegate(
        typeof(OneParameter<int, int>),
        new Example(42)
    );

Console.WriteLine($"3 * test = {invoke(3)}");
```

```
    }
}
/* This code example produces the following output:

123456789 squared = 15241578750190521
3 * test = 126
 */
```

See also

- DynamicMethod

How to: Define a generic method with reflection emit

03/30/2024

The first procedure shows how to create a simple generic method with two type parameters, and how to apply class constraints, interface constraints, and special constraints to the type parameters.

The second procedure shows how to emit the method body, and how to use the type parameters of the generic method to create instances of generic types and to call their methods.

The third procedure shows how to invoke the generic method.

> **Important**
>
> A method is not generic just because it belongs to a generic type and uses the type parameters of that type. A method is generic only if it has its own type parameter list. A generic method can appear on a nongeneric type, as in this example. For an example of a nongeneric method on a generic type, see How to: Define a Generic Type with Reflection Emit.

Define a generic method

1. Before beginning, it is useful to look at how the generic method appears when written using a high-level language. The following code is included in the example code for this article, along with code to call the generic method. The method has two type parameters, `TInput` and `TOutput`, the second of which must be a reference type (`class`), must have a parameterless constructor (`new`), and must implement `ICollection<TInput>`. This interface constraint ensures that the ICollection<T>.Add method can be used to add elements to the `TOutput` collection that the method creates. The method has one formal parameter, `input`, which is an array of `TInput`. The method creates a collection of type `TOutput` and copies the elements of `input` to the collection.

C#

```
public static TOutput Factory<TInput, TOutput>(TInput[] tarray)
    where TOutput : class, ICollection<TInput>, new()
{
    TOutput ret = new TOutput();
```

```
    ICollection<TInput> ic = ret;

    foreach (TInput t in tarray)
    {
        ic.Add(t);
    }
    return ret;
}
```

2. Define a dynamic assembly and a dynamic module to contain the type the generic method belongs to. In this case, the assembly has only one module, named `DemoMethodBuilder1`, and the module name is the same as the assembly name plus an extension. In this example, the assembly is saved to disk and also executed, so AssemblyBuilderAccess.RunAndSave is specified. You can use the Ildasm.exe (IL Disassembler) to examine DemoMethodBuilder1.dll and to compare it to the common intermediate language (CIL) for the method shown in step 1.

C#

```
AssemblyName asmName = new AssemblyName("DemoMethodBuilder1");
AppDomain domain = AppDomain.CurrentDomain;
AssemblyBuilder demoAssembly =
    domain.DefineDynamicAssembly(asmName,
        AssemblyBuilderAccess.RunAndSave);

// Define the module that contains the code. For an
// assembly with one module, the module name is the
// assembly name plus a file extension.
ModuleBuilder demoModule =
    demoAssembly.DefineDynamicModule(asmName.Name,
        asmName.Name+".dll");
```

3. Define the type the generic method belongs to. The type does not have to be generic. A generic method can belong to either a generic or nongeneric type. In this example, the type is a class, is not generic, and is named `DemoType`.

C#

```
TypeBuilder demoType =
    demoModule.DefineType("DemoType", TypeAttributes.Public);
```

4. Define the generic method. If the types of a generic method's formal parameters are specified by generic type parameters of the generic method, use the DefineMethod(String, MethodAttributes) method overload to define the method. The generic type parameters of the method are not yet defined, so you cannot specify the

types of the method's formal parameters in the call to DefineMethod. In this example, the method is named `Factory`. The method is public and `static` (`Shared` in Visual Basic).

C#

```
MethodBuilder factory =
    demoType.DefineMethod("Factory",
        MethodAttributes.Public | MethodAttributes.Static);
```

5. Define the generic type parameters of `DemoMethod` by passing an array of strings containing the names of the parameters to the MethodBuilder.DefineGenericParameters method. This makes the method a generic method. The following code makes `Factory` a generic method with type parameters `TInput` and `TOutput`. To make the code easier to read, variables with these names are created to hold the GenericTypeParameterBuilder objects representing the two type parameters.

C#

```
string[] typeParameterNames = {"TInput", "TOutput"};
GenericTypeParameterBuilder[] typeParameters =
    factory.DefineGenericParameters(typeParameterNames);

GenericTypeParameterBuilder TInput = typeParameters[0];
GenericTypeParameterBuilder TOutput = typeParameters[1];
```

6. Optionally add special constraints to the type parameters. Special constraints are added using the SetGenericParameterAttributes method. In this example, `TOutput` is constrained to be a reference type and to have a parameterless constructor.

C#

```
TOutput.SetGenericParameterAttributes(
    GenericParameterAttributes.ReferenceTypeConstraint |
    GenericParameterAttributes.DefaultConstructorConstraint);
```

7. Optionally add class and interface constraints to the type parameters. In this example, type parameter `TOutput` is constrained to types that implement the `ICollection(Of TInput)` (`ICollection<TInput>` in C#) interface. This ensures that the Add method can be used to add elements.

C#

```
Type icoll = typeof(ICollection<>);
Type icollOfTInput = icoll.MakeGenericType(TInput);
```

```
Type[] constraints = {icollOfTInput};
TOutput.SetInterfaceConstraints(constraints);
```

8. Define the formal parameters of the method, using the SetParameters method. In this example, the `Factory` method has one parameter, an array of `TInput`. This type is created by calling the MakeArrayType method on the GenericTypeParameterBuilder that represents `TInput`. The argument of SetParameters is an array of Type objects.

```
C#

Type[] parms = {TInput.MakeArrayType()};
factory.SetParameters(parms);
```

9. Define the return type for the method, using the SetReturnType method. In this example, an instance of `TOutput` is returned.

```
C#

factory.SetReturnType(TOutput);
```

10. Emit the method body, using ILGenerator. For details, see the accompanying procedure for emitting the method body.

> **ⓘ Important**
>
> When you emit calls to methods of generic types, and the type arguments of those types are type parameters of the generic method, you must use the `static` GetConstructor(Type, ConstructorInfo), GetMethod(Type, MethodInfo), and GetField(Type, FieldInfo) method overloads of the TypeBuilder class to obtain constructed forms of the methods. The accompanying procedure for emitting the method body demonstrates this.

11. Complete the type that contains the method and save the assembly. The accompanying procedure for invoking the generic method shows two ways to invoke the completed method.

```
C#

// Complete the type.
Type dt = demoType.CreateType();
// Save the assembly, so it can be examined with Ildasm.exe.
demoAssembly.Save(asmName.Name+".dll");
```

Emit the method body

1. Get a code generator and declare local variables and labels. The DeclareLocal method is used to declare local variables. The `Factory` method has four local variables: `retVal` to hold the new `TOutput` that is returned by the method, `ic` to hold the `TOutput` when it is cast to `ICollection<TInput>`, `input` to hold the input array of `TInput` objects, and `index` to iterate through the array. The method also has two labels, one to enter the loop (`enterLoop`) and one for the top of the loop (`loopAgain`), defined using the DefineLabel method.

   The first thing the method does is to load its argument using Ldarg_0 opcode and to store it in the local variable `input` using Stloc_S opcode.

   C#

   ```
   ILGenerator ilgen = factory.GetILGenerator();

   LocalBuilder retVal = ilgen.DeclareLocal(TOutput);
   LocalBuilder ic = ilgen.DeclareLocal(icollOfTInput);
   LocalBuilder input = ilgen.DeclareLocal(TInput.MakeArrayType());
   LocalBuilder index = ilgen.DeclareLocal(typeof(int));

   Label enterLoop = ilgen.DefineLabel();
   Label loopAgain = ilgen.DefineLabel();

   ilgen.Emit(OpCodes.Ldarg_0);
   ilgen.Emit(OpCodes.Stloc_S, input);
   ```

2. Emit code to create an instance of `TOutput`, using the generic method overload of the Activator.CreateInstance method. Using this overload requires the specified type to have a parameterless constructor, which is the reason for adding that constraint to `TOutput`. Create the constructed generic method by passing `TOutput` to MakeGenericMethod. After emitting code to call the method, emit code to store it in the local variable `retVal` using Stloc_S

   C#

   ```
   MethodInfo createInst =
       typeof(Activator).GetMethod("CreateInstance", Type.EmptyTypes);
   MethodInfo createInstOfTOutput =
       createInst.MakeGenericMethod(TOutput);

   ilgen.Emit(OpCodes.Call, createInstOfTOutput);
   ilgen.Emit(OpCodes.Stloc_S, retVal);
   ```

3. Emit code to cast the new `TOutput` object to `ICollection(Of TInput)` and store it in the local variable `ic`.

```
C#

ilgen.Emit(OpCodes.Ldloc_S, retVal);
ilgen.Emit(OpCodes.Box, TOutput);
ilgen.Emit(OpCodes.Castclass, icollOfTInput);
ilgen.Emit(OpCodes.Stloc_S, ic);
```

4. Get a MethodInfo representing the ICollection<T>.Add method. The method is acting on an `ICollection<TInput>`, so it's necessary to get the `Add` method specific to that constructed type. You cannot use the GetMethod method to get this MethodInfo directly from `icollOfTInput`, because GetMethod is not supported on a type that has been constructed with a GenericTypeParameterBuilder. Instead, call GetMethod on `icoll`, which contains the generic type definition for the ICollection<T> generic interface. Then use the GetMethod(Type, MethodInfo) `static` method to produce the MethodInfo for the constructed type. The following code demonstrates this.

```
C#

MethodInfo mAddPrep = icoll.GetMethod("Add");
MethodInfo mAdd = TypeBuilder.GetMethod(icollOfTInput, mAddPrep);
```

5. Emit code to initialize the `index` variable, by loading a 32-bit integer 0 and storing it in the variable. Emit code to branch to the label `enterLoop`. This label has not yet been marked, because it is inside the loop. Code for the loop is emitted in the next step.

```
C#

// Initialize the count and enter the loop.
ilgen.Emit(OpCodes.Ldc_I4_0);
ilgen.Emit(OpCodes.Stloc_S, index);
ilgen.Emit(OpCodes.Br_S, enterLoop);
```

6. Emit code for the loop. The first step is to mark the top of the loop, by calling MarkLabel with the `loopAgain` label. Branch statements that use the label will now branch to this point in the code. The next step is to push the `TOutput` object, cast to `ICollection(Of TInput)`, onto the stack. It is not needed immediately, but needs to be in position for calling the `Add` method. Next the input array is pushed onto the stack, then the `index` variable containing the current index into the array. The Ldelem opcode pops the index and the array off the stack and pushes the indexed array element onto the stack. The

stack is now ready for the call to the ICollection<T>.Add method, which pops the collection and the new element off the stack and adds the element to the collection.

The rest of the code in the loop increments the index and tests to see whether the loop is finished: The index and a 32-bit integer 1 are pushed onto the stack and added, leaving the sum on the stack; the sum is stored in `index`. MarkLabel is called to set this point as the entry point for the loop. The index is loaded again. The input array is pushed on the stack, and Ldlen is emitted to get its length. The index and the length are now on the stack, and Clt is emitted to compare them. If the index is less than the length, Brtrue_S branches back to the beginning of the loop.

C#

```
ilgen.MarkLabel(loopAgain);

ilgen.Emit(OpCodes.Ldloc_S, ic);
ilgen.Emit(OpCodes.Ldloc_S, input);
ilgen.Emit(OpCodes.Ldloc_S, index);
ilgen.Emit(OpCodes.Ldelem, TInput);
ilgen.Emit(OpCodes.Callvirt, mAdd);

ilgen.Emit(OpCodes.Ldloc_S, index);
ilgen.Emit(OpCodes.Ldc_I4_1);
ilgen.Emit(OpCodes.Add);
ilgen.Emit(OpCodes.Stloc_S, index);

ilgen.MarkLabel(enterLoop);
ilgen.Emit(OpCodes.Ldloc_S, index);
ilgen.Emit(OpCodes.Ldloc_S, input);
ilgen.Emit(OpCodes.Ldlen);
ilgen.Emit(OpCodes.Conv_I4);
ilgen.Emit(OpCodes.Clt);
ilgen.Emit(OpCodes.Brtrue_S, loopAgain);
```

7. Emit code to push the `TOutput` object onto the stack and return from the method. The local variables `retVal` and `ic` both contain references to the new `TOutput`; `ic` is used only to access the ICollection<T>.Add method.

C#

```
ilgen.Emit(OpCodes.Ldloc_S, retVal);
ilgen.Emit(OpCodes.Ret);
```

Invoke the generic method

1. `Factory` is a generic method definition. In order to invoke it, you must assign types to its generic type parameters. Use the MakeGenericMethod method to do this. The following code creates a constructed generic method, specifying String for `TInput` and `List(Of String)` (`List<string>` in C#) for `TOutput`, and displays a string representation of the method.

```
C#

MethodInfo m = dt.GetMethod("Factory");
MethodInfo bound =
    m.MakeGenericMethod(typeof(string), typeof(List<string>));

// Display a string representing the bound method.
Console.WriteLine(bound);
```

2. To invoke the method late-bound, use the Invoke method. The following code creates an array of Object, containing as its only element an array of strings, and passes it as the argument list for the generic method. The first parameter of Invoke is a null reference because the method is `static`. The return value is cast to `List(Of String)`, and its first element is displayed.

```
C#

object o = bound.Invoke(null, new object[]{arr});
List<string> list2 = (List<string>) o;

Console.WriteLine($"The first element is: {list2[0]}");
```

3. To invoke the method using a delegate, you must have a delegate that matches the signature of the constructed generic method. An easy way to do this is to create a generic delegate. The following code creates an instance of the generic delegate `D` defined in the example code, using the Delegate.CreateDelegate(Type, MethodInfo) method overload, and invokes the delegate. Delegates perform better than late-bound calls.

```
C#

Type dType = typeof(D<string, List <string>>);
D<string, List <string>> test;
test = (D<string, List <string>>)
    Delegate.CreateDelegate(dType, bound);

List<string> list3 = test(arr);
Console.WriteLine($"The first element is: {list3[0]}");
```

4. The emitted method can also be called from a program that refers to the saved assembly.

Example

The following code example creates a nongeneric type, `DemoType`, with a generic method, `Factory`. This method has two generic type parameters, `TInput` to specify an input type and `TOutput` to specify an output type. The `TOutput` type parameter is constrained to implement `ICollection<TInput>` (`ICollection(Of TInput)` in Visual Basic), to be a reference type, and to have a parameterless constructor.

The method has one formal parameter, which is an array of `TInput`. The method returns an instance of `TOutput` that contains all the elements of the input array. `TOutput` can be any generic collection type that implements the ICollection<T> generic interface.

When the code is executed, the dynamic assembly is saved as DemoGenericMethod1.dll, and can be examined using the Ildasm.exe (IL Disassembler).

> **Note**
>
> A good way to learn how to emit code is to write a program that performs the task you're trying to emit, and use the disassembler to examine the CIL produced by the compiler.

The code example includes source code that's equivalent to the emitted method. The emitted method is invoked late-bound and also by using a generic delegate declared in the code example.

C#

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Reflection.Emit;

// Declare a generic delegate that can be used to execute the
// finished method.
//
public delegate TOut D<TIn, TOut>(TIn[] input);

class GenericMethodBuilder
{
    // This method shows how to declare, in Visual Basic, the generic
    // method this program emits. The method has two type parameters,
    // TInput and TOutput, the second of which must be a reference type
    // (class), must have a parameterless constructor (new()), and must
    // implement ICollection<TInput>. This interface constraint
```

```
    // ensures that ICollection<TInput>.Add can be used to add
    // elements to the TOutput object the method creates. The method
    // has one formal parameter, input, which is an array of TInput.
    // The elements of this array are copied to the new TOutput.
    //
    public static TOutput Factory<TInput, TOutput>(TInput[] tarray)
        where TOutput : class, ICollection<TInput>, new()
    {
        TOutput ret = new TOutput();
        ICollection<TInput> ic = ret;

        foreach (TInput t in tarray)
        {
            ic.Add(t);
        }
        return ret;
    }

    public static void Main()
    {
        // The following shows the usage syntax of the C#
        // version of the generic method emitted by this program.
        // Note that the generic parameters must be specified
        // explicitly, because the compiler does not have enough
        // context to infer the type of TOutput. In this case, TOutput
        // is a generic List containing strings.
        //
        string[] arr = {"a", "b", "c", "d", "e"};
        List<string> list1 =
            GenericMethodBuilder.Factory<string, List <string>>(arr);
        Console.WriteLine($"The first element is: {list1[0]}");

        // Creating a dynamic assembly requires an AssemblyName
        // object, and the current application domain.
        //
        AssemblyName asmName = new AssemblyName("DemoMethodBuilder1");
        AppDomain domain = AppDomain.CurrentDomain;
        AssemblyBuilder demoAssembly =
            domain.DefineDynamicAssembly(asmName,
                AssemblyBuilderAccess.RunAndSave);

        // Define the module that contains the code. For an
        // assembly with one module, the module name is the
        // assembly name plus a file extension.
        ModuleBuilder demoModule =
            demoAssembly.DefineDynamicModule(asmName.Name,
                asmName.Name+".dll");

        // Define a type to contain the method.
        TypeBuilder demoType =
            demoModule.DefineType("DemoType", TypeAttributes.Public);

        // Define a public static method with standard calling
        // conventions. Do not specify the parameter types or the
        // return type, because type parameters will be used for
```

```
        // those types, and the type parameters have not been
        // defined yet.
        //
        MethodBuilder factory =
            demoType.DefineMethod("Factory",
                MethodAttributes.Public | MethodAttributes.Static);

        // Defining generic type parameters for the method makes it a
        // generic method. To make the code easier to read, each
        // type parameter is copied to a variable of the same name.
        //
        string[] typeParameterNames = {"TInput", "TOutput"};
        GenericTypeParameterBuilder[] typeParameters =
            factory.DefineGenericParameters(typeParameterNames);

        GenericTypeParameterBuilder TInput = typeParameters[0];
        GenericTypeParameterBuilder TOutput = typeParameters[1];

        // Add special constraints.
        // The type parameter TOutput is constrained to be a reference
        // type, and to have a parameterless constructor. This ensures
        // that the Factory method can create the collection type.
        //
        TOutput.SetGenericParameterAttributes(
            GenericParameterAttributes.ReferenceTypeConstraint |
            GenericParameterAttributes.DefaultConstructorConstraint);

        // Add interface and base type constraints.
        // The type parameter TOutput is constrained to types that
        // implement the ICollection<T> interface, to ensure that
        // they have an Add method that can be used to add elements.
        //
        // To create the constraint, first use MakeGenericType to bind
        // the type parameter TInput to the ICollection<T> interface,
        // returning the type ICollection<TInput>, then pass
        // the newly created type to the SetInterfaceConstraints
        // method. The constraints must be passed as an array, even if
        // there is only one interface.
        //
        Type icoll = typeof(ICollection<>);
        Type icollOfTInput = icoll.MakeGenericType(TInput);
        Type[] constraints = {icollOfTInput};
        TOutput.SetInterfaceConstraints(constraints);

        // Set parameter types for the method. The method takes
        // one parameter, an array of type TInput.
        Type[] parms = {TInput.MakeArrayType()};
        factory.SetParameters(parms);

        // Set the return type for the method. The return type is
        // the generic type parameter TOutput.
        factory.SetReturnType(TOutput);

        // Generate a code body for the method.
        // ------------------------------------
```

```
        // Get a code generator and declare local variables and
        // labels. Save the input array to a local variable.
        //
        ILGenerator ilgen = factory.GetILGenerator();

        LocalBuilder retVal = ilgen.DeclareLocal(TOutput);
        LocalBuilder ic = ilgen.DeclareLocal(icollOfTInput);
        LocalBuilder input = ilgen.DeclareLocal(TInput.MakeArrayType());
        LocalBuilder index = ilgen.DeclareLocal(typeof(int));

        Label enterLoop = ilgen.DefineLabel();
        Label loopAgain = ilgen.DefineLabel();

        ilgen.Emit(OpCodes.Ldarg_0);
        ilgen.Emit(OpCodes.Stloc_S, input);

        // Create an instance of TOutput, using the generic method
        // overload of the Activator.CreateInstance method.
        // Using this overload requires the specified type to have
        // a parameterless constructor, which is the reason for adding
        // that constraint to TOutput. Create the constructed generic
        // method by passing TOutput to MakeGenericMethod. After
        // emitting code to call the method, emit code to store the
        // new TOutput in a local variable.
        //
        MethodInfo createInst =
            typeof(Activator).GetMethod("CreateInstance", Type.EmptyTypes);
        MethodInfo createInstOfTOutput =
            createInst.MakeGenericMethod(TOutput);

        ilgen.Emit(OpCodes.Call, createInstOfTOutput);
        ilgen.Emit(OpCodes.Stloc_S, retVal);

        // Load the reference to the TOutput object, cast it to
        // ICollection<TInput>, and save it.
        //
        ilgen.Emit(OpCodes.Ldloc_S, retVal);
        ilgen.Emit(OpCodes.Box, TOutput);
        ilgen.Emit(OpCodes.Castclass, icollOfTInput);
        ilgen.Emit(OpCodes.Stloc_S, ic);

        // Loop through the array, adding each element to the new
        // instance of TOutput. Note that in order to get a MethodInfo
        // for ICollection<TInput>.Add, it is necessary to first
        // get the Add method for the generic type defintion,
        // ICollection<T>.Add. This is because it is not possible
        // to call GetMethod on icollOfTInput. The static overload of
        // TypeBuilder.GetMethod produces the correct MethodInfo for
        // the constructed type.
        //
        MethodInfo mAddPrep = icoll.GetMethod("Add");
        MethodInfo mAdd = TypeBuilder.GetMethod(icollOfTInput, mAddPrep);

        // Initialize the count and enter the loop.
        ilgen.Emit(OpCodes.Ldc_I4_0);
```

```
            ilgen.Emit(OpCodes.Stloc_S, index);
            ilgen.Emit(OpCodes.Br_S, enterLoop);

            // Mark the beginning of the loop. Push the ICollection
            // reference on the stack, so it will be in position for the
            // call to Add. Then push the array and the index on the
            // stack, get the array element, and call Add (represented
            // by the MethodInfo mAdd) to add it to the collection.
            //
            // The other ten instructions just increment the index
            // and test for the end of the loop. Note the MarkLabel
            // method, which sets the point in the code where the
            // loop is entered. (See the earlier Br_S to enterLoop.)
            //
            ilgen.MarkLabel(loopAgain);

            ilgen.Emit(OpCodes.Ldloc_S, ic);
            ilgen.Emit(OpCodes.Ldloc_S, input);
            ilgen.Emit(OpCodes.Ldloc_S, index);
            ilgen.Emit(OpCodes.Ldelem, TInput);
            ilgen.Emit(OpCodes.Callvirt, mAdd);

            ilgen.Emit(OpCodes.Ldloc_S, index);
            ilgen.Emit(OpCodes.Ldc_I4_1);
            ilgen.Emit(OpCodes.Add);
            ilgen.Emit(OpCodes.Stloc_S, index);

            ilgen.MarkLabel(enterLoop);
            ilgen.Emit(OpCodes.Ldloc_S, index);
            ilgen.Emit(OpCodes.Ldloc_S, input);
            ilgen.Emit(OpCodes.Ldlen);
            ilgen.Emit(OpCodes.Conv_I4);
            ilgen.Emit(OpCodes.Clt);
            ilgen.Emit(OpCodes.Brtrue_S, loopAgain);

            ilgen.Emit(OpCodes.Ldloc_S, retVal);
            ilgen.Emit(OpCodes.Ret);

            // Complete the type.
            Type dt = demoType.CreateType();
            // Save the assembly, so it can be examined with Ildasm.exe.
            demoAssembly.Save(asmName.Name+".dll");

            // To create a constructed generic method that can be
            // executed, first call the GetMethod method on the completed
            // type to get the generic method definition. Call MakeGenericType
            // on the generic method definition to obtain the constructed
            // method, passing in the type arguments. In this case, the
            // constructed method has string for TInput and List<string>
            // for TOutput.
            //
            MethodInfo m = dt.GetMethod("Factory");
            MethodInfo bound =
                m.MakeGenericMethod(typeof(string), typeof(List<string>));
```

```
        // Display a string representing the bound method.
        Console.WriteLine(bound);

        // Once the generic method is constructed,
        // you can invoke it and pass in an array of objects
        // representing the arguments. In this case, there is only
        // one element in that array, the argument 'arr'.
        //
        object o = bound.Invoke(null, new object[]{arr});
        List<string> list2 = (List<string>) o;

        Console.WriteLine($"The first element is: {list2[0]}");

        // You can get better performance from multiple calls if
        // you bind the constructed method to a delegate. The
        // following code uses the generic delegate D defined
        // earlier.
        //
        Type dType = typeof(D<string, List <string>>);
        D<string, List <string>> test;
        test = (D<string, List <string>>)
            Delegate.CreateDelegate(dType, bound);

        List<string> list3 = test(arr);
        Console.WriteLine($"The first element is: {list3[0]}");
    }
}

/* This code example produces the following output:

The first element is: a
System.Collections.Generic.List`1[System.String] Factory[String,List`1]
(System.String[])
The first element is: a
The first element is: a
 */
```

See also

- MethodBuilder
- How to: Define a Generic Type with Reflection Emit

MethodBuilder Class

Definition

Namespace: System.Reflection.Emit

Assemblies: netstandard.dll, System.Reflection.Emit.dll

Source: MethodBuilder.cs

Defines and represents a method (or constructor) on a dynamic class.

C#

```
public abstract class MethodBuilder : System.Reflection.MethodInfo
```

Inheritance Object → MemberInfo → MethodBase → MethodInfo → MethodBuilder

Examples

The following example uses the MethodBuilder class to create a method within a dynamic type.

C#

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class DemoMethodBuilder
{
    public static void AddMethodDynamically (TypeBuilder myTypeBld,
                                             string mthdName,
                                             Type[] mthdParams,
                                             Type returnType,
                                             string mthdAction)
    {

        MethodBuilder myMthdBld = myTypeBld.DefineMethod(
                                             mthdName,
                                             MethodAttributes.Public |
                                             MethodAttributes.Static,
                                             returnType,
                                             mthdParams);

        ILGenerator ILout = myMthdBld.GetILGenerator();
```

```
        int numParams = mthdParams.Length;

        for (byte x=0; x < numParams; x++)
        {
            ILout.Emit(OpCodes.Ldarg_S, x);
        }

        if (numParams > 1)
        {
            for (int y=0; y<(numParams-1); y++)
            {
                switch (mthdAction)
                {
                    case "A": ILout.Emit(OpCodes.Add);
                              break;
                    case "M": ILout.Emit(OpCodes.Mul);
                              break;
                    default: ILout.Emit(OpCodes.Add);
                              break;
                }
            }
        }
        ILout.Emit(OpCodes.Ret);
    }

    public static void Main()
    {
        AppDomain myDomain = AppDomain.CurrentDomain;
        AssemblyName asmName = new AssemblyName();
        asmName.Name = "MyDynamicAsm";

        AssemblyBuilder myAsmBuilder = myDomain.DefineDynamicAssembly(
                                       asmName,
                                       AssemblyBuilderAccess.RunAndSave);

        ModuleBuilder myModule = myAsmBuilder.DefineDynamicModule("MyDynamicAsm",

"MyDynamicAsm.dll");

        TypeBuilder myTypeBld = myModule.DefineType("MyDynamicType",
                                                    TypeAttributes.Public);

        // Get info from the user to build the method dynamically.
        Console.WriteLine("Let's build a simple method dynamically!");
        Console.WriteLine("Please enter a few numbers, separated by spaces.");
        string inputNums = Console.ReadLine();
        Console.Write("Do you want to [A]dd (default) or [M]ultiply these numbers?
");
        string myMthdAction = Console.ReadLine().ToUpper();
        Console.Write("Lastly, what do you want to name your new dynamic method?
");
        string myMthdName = Console.ReadLine();

        // Process inputNums into an array and create a corresponding Type array
        int index = 0;
```

```
        string[] inputNumsList = inputNums.Split();

        Type[] myMthdParams = new Type[inputNumsList.Length];
        object[] inputValsList = new object[inputNumsList.Length];

        foreach (string inputNum in inputNumsList)
        {
            inputValsList[index] = (object)Convert.ToInt32(inputNum);
                myMthdParams[index] = typeof(int);
                index++;
        }

        // Now, call the method building method with the parameters, passing the
        // TypeBuilder by reference.
        AddMethodDynamically(myTypeBld,
                             myMthdName,
                             myMthdParams,
                             typeof(int),
                             myMthdAction);

        Type myType = myTypeBld.CreateType();

        Console.WriteLine("---");
        Console.WriteLine("The result of {0} the inputted values is: {1}",
                          ((myMthdAction == "M") ? "multiplying" : "adding"),
                          myType.InvokeMember(myMthdName,
                          BindingFlags.InvokeMethod | BindingFlags.Public |
                          BindingFlags.Static,
                          null,
                          null,
                          inputValsList));
        Console.WriteLine("---");

        // Let's take a look at the method we created.
        // If you are interested in seeing the MSIL generated dynamically for the
method
        // your program generated, change to the directory where you ran the com-
piled
        // code sample and type "ildasm MyDynamicAsm.dll" at the prompt. When the
list
        // of manifest contents appears, click on "MyDynamicType" and then on the
name of
        // of the method you provided during execution.

        myAsmBuilder.Save("MyDynamicAsm.dll");

        MethodInfo myMthdInfo = myType.GetMethod(myMthdName);
        Console.WriteLine("Your Dynamic Method: {0};", myMthdInfo.ToString());
    }
}
```

Remarks

For more information about this API, see Supplemental API remarks for MethodBuilder.

Constructors

Expand table

MethodBuilder()	Initializes a new instance of the MethodBuilder class.

Properties

Expand table

Attributes	Retrieves the attributes for this method.
CallingConvention	Returns the calling convention of the method.
ContainsGeneric Parameters	Not supported for this type.
CustomAttributes	Gets a collection that contains this member's custom attributes. (Inherited from MemberInfo)
DeclaringType	Returns the type that declares this method.
InitLocals	Gets or sets a Boolean value that specifies whether the local variables in this method are zero initialized. The default value of this property is `true`.
InitLocalsCore	When overridden in a derived class, gets or sets a value that indicates whether the local variables in this method are zero-initialized.
IsAbstract	Gets a value indicating whether the method is abstract. (Inherited from MethodBase)
IsAssembly	Gets a value indicating whether the potential visibility of this method or constructor is described by Assembly; that is, the method or constructor is visible at most to other types in the same assembly, and is not visible to derived types outside the assembly. (Inherited from MethodBase)
IsCollectible	Gets a value that indicates whether this MemberInfo object is part of an assembly held in a collectible AssemblyLoadContext. (Inherited from MemberInfo)
IsConstructed GenericMethod	(Inherited from MethodBase)
IsConstructor	Gets a value indicating whether the method is a constructor.

	(Inherited from MethodBase)
IsFamily	Gets a value indicating whether the visibility of this method or constructor is described by Family; that is, the method or constructor is visible only within its class and derived classes. (Inherited from MethodBase)
IsFamilyAndAssembly	Gets a value indicating whether the visibility of this method or constructor is described by FamANDAssem; that is, the method or constructor can be called by derived classes, but only if they are in the same assembly. (Inherited from MethodBase)
IsFamilyOrAssembly	Gets a value indicating whether the potential visibility of this method or constructor is described by FamORAssem; that is, the method or constructor can be called by derived classes wherever they are, and by classes in the same assembly. (Inherited from MethodBase)
IsFinal	Gets a value indicating whether this method is `final`. (Inherited from MethodBase)
IsGenericMethod	Gets a value indicating whether the method is a generic method.
IsGenericMethodDefinition	Gets a value indicating whether the current MethodBuilder object represents the definition of a generic method.
IsHideBySig	Gets a value indicating whether only a member of the same kind with exactly the same signature is hidden in the derived class. (Inherited from MethodBase)
IsPrivate	Gets a value indicating whether this member is private. (Inherited from MethodBase)
IsPublic	Gets a value indicating whether this is a public method. (Inherited from MethodBase)
IsSecurityCritical	Throws a NotSupportedException in all cases.
IsSecuritySafeCritical	Throws a NotSupportedException in all cases.
IsSecurityTransparent	Throws a NotSupportedException in all cases.
IsSpecialName	Gets a value indicating whether this method has a special name. (Inherited from MethodBase)
IsStatic	Gets a value indicating whether the method is `static`. (Inherited from MethodBase)
IsVirtual	Gets a value indicating whether the method is `virtual`. (Inherited from MethodBase)

MemberType	Gets a MemberTypes value indicating that this member is a method. (Inherited from MethodInfo)
MetadataToken	Gets a token that identifies the current dynamic module in metadata.
MethodHandle	Retrieves the internal handle for the method. Use this handle to access the underlying metadata handle.
Method Implementation Flags	Gets the MethodImplAttributes flags that specify the attributes of a method implementation. (Inherited from MethodBase)
Module	Gets the module in which the current method is being defined.
Name	Retrieves the name of this method.
ReflectedType	Retrieves the class that was used in reflection to obtain this object.
ReturnParameter	Gets a ParameterInfo object that contains information about the return type of the method, such as whether the return type has custom modifiers.
ReturnType	Gets the return type of the method represented by this MethodBuilder.
ReturnTypeCustom Attributes	Returns the custom attributes of the method's return type.

Methods

Expand table

CreateDelegate(Type, Object)	Creates a delegate of the specified type with the specified target from this method. (Inherited from MethodInfo)
CreateDelegate(Type)	Creates a delegate of the specified type from this method. (Inherited from MethodInfo)
CreateDelegate<T>()	Creates a delegate of type `T` from this method. (Inherited from MethodInfo)
CreateDelegate<T>(Object)	Creates a delegate of type `T` with the specified target from this method. (Inherited from MethodInfo)
DefineGenericParameters(String[])	Sets the number of generic type parameters for the current method, specifies their names, and returns an array of GenericTypeParameterBuilder objects that can be used to define their constraints.

DefineGenericParameters Core(String[])	When overridden in a derived class, sets the number of generic type parameters for the current method, specifies their names, and returns an array of GenericTypeParameterBuilder objects that can be used to define their constraints.
DefineParameter(Int32, Parameter Attributes, String)	Sets the parameter attributes and the name of a parameter of this method, or of the return value of this method. Returns a ParameterBuilder that can be used to apply custom attributes.
DefineParameterCore(Int32, ParameterAttributes, String)	When overridden in a derived class, defines a parameter or return parameter for this method.
Equals(Object)	Determines whether the given object is equal to this instance.
GetBaseDefinition()	Return the base implementation for a method.
GetCustomAttributes(Boolean)	Returns all the custom attributes defined for this method.
GetCustomAttributes(Type, Boolean)	Returns the custom attributes identified by the given type.
GetCustomAttributesData()	Returns a list of CustomAttributeData objects representing data about the attributes that have been applied to the target member. (Inherited from MemberInfo)
GetGenericArguments()	Returns an array of GenericTypeParameterBuilder objects that represent the type parameters of the method, if it is generic.
GetGenericMethodDefinition()	Returns this method.
GetHashCode()	Gets the hash code for this method.
GetILGenerator()	Returns an `ILGenerator` for this method with a default Microsoft intermediate language (MSIL) stream size of 64 bytes.
GetILGenerator(Int32)	Returns an `ILGenerator` for this method with the specified Microsoft intermediate language (MSIL) stream size.
GetILGeneratorCore(Int32)	When overridden in a derived class, gets an ILGenerator that can be used to emit a method body for this method.
GetMethodBody()	When overridden in a derived class, gets a MethodBody object that provides access to the MSIL stream, local variables, and exceptions for the current method. (Inherited from MethodBase)
GetMethodImplementationFlags()	Returns the implementation flags for the method.
GetParameters()	Returns the parameters of this method.
HasSameMetadataDefinition As(MemberInfo)	(Inherited from MemberInfo)

Invoke(Object, BindingFlags, Binder, Object[], CultureInfo)	Dynamically invokes the method reflected by this instance on the given object, passing along the specified parameters, and under the constraints of the given binder.
IsDefined(Type, Boolean)	Checks if the specified custom attribute type is defined.
MakeGenericMethod(Type[])	Returns a generic method constructed from the current generic method definition using the specified generic type arguments.
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
SetCustomAttribute(ConstructorInfo, Byte[])	Sets a custom attribute using a specified custom attribute blob.
SetCustomAttribute(CustomAttributeBuilder)	Sets a custom attribute using a custom attribute builder.
SetCustomAttributeCore(ConstructorInfo, ReadOnlySpan<Byte>)	When overridden in a derived class, sets a custom attribute on this assembly.
SetImplementationFlags(MethodImplAttributes)	Sets the implementation flags for this method.
SetImplementationFlagsCore(MethodImplAttributes)	When overridden in a derived class, sets the implementation flags for this method.
SetParameters(Type[])	Sets the number and types of parameters for a method.
SetReturnType(Type)	Sets the return type of the method.
SetSignature(Type, Type[], Type[], Type[], Type[][], Type[][])	Sets the method signature, including the return type, the parameter types, and the required and optional custom modifiers of the return type and parameter types.
SetSignatureCore(Type, Type[], Type[], Type[], Type[][], Type[][])	When overridden in a derived class, sets the method signature, including the return type, the parameter types, and the required and optional custom modifiers of the return type and parameter types.
ToString()	Returns this `MethodBuilder` instance as a string.

Expand table

Extension Methods

Expand table

GetCustomAttribute(MemberInfo, Type, Boolean)	Retrieves a custom attribute of a specified type that is applied to a specified member, and optionally inspects the ancestors of that member.
GetCustomAttribute(MemberInfo, Type)	Retrieves a custom attribute of a specified type that is applied to a specified member.
GetCustomAttribute<T>(MemberInfo, Boolean)	Retrieves a custom attribute of a specified type that is applied to a specified member, and optionally inspects the ancestors of that member.
GetCustomAttribute<T>(MemberInfo)	Retrieves a custom attribute of a specified type that is applied to a specified member.
GetCustomAttributes(MemberInfo, Boolean)	Retrieves a collection of custom attributes that are applied to a specified member, and optionally inspects the ancestors of that member.
GetCustomAttributes(MemberInfo, Type, Boolean)	Retrieves a collection of custom attributes of a specified type that are applied to a specified member, and optionally inspects the ancestors of that member.
GetCustomAttributes(MemberInfo, Type)	Retrieves a collection of custom attributes of a specified type that are applied to a specified member.
GetCustomAttributes(MemberInfo)	Retrieves a collection of custom attributes that are applied to a specified member.
GetCustomAttributes<T>(MemberInfo, Boolean)	Retrieves a collection of custom attributes of a specified type that are applied to a specified member, and optionally inspects the ancestors of that member.
GetCustomAttributes<T>(MemberInfo)	Retrieves a collection of custom attributes of a specified type that are applied to a specified member.
IsDefined(MemberInfo, Type, Boolean)	Indicates whether custom attributes of a specified type are applied to a specified member, and, optionally, applied to its ancestors.
IsDefined(MemberInfo, Type)	Indicates whether custom attributes of a specified type are applied to a specified member.
GetMetadataToken(MemberInfo)	Gets a metadata token for the given member, if available.
HasMetadataToken(MemberInfo)	Returns a value that indicates whether a metadata token is available for the specified member.
GetBaseDefinition(MethodInfo)	
GetRuntimeBaseDefinition(MethodInfo)	Retrieves an object that represents the specified method on the direct or indirect base class where the method was first declared.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8, 9, 10
.NET Framework	1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	2.0 (package-provided), 2.1

C#/.NET Most performant way to call a method dynamically

Asked 6 years, 6 months ago Modified 6 years, 6 months ago Viewed 2k times

3

We are developing a system, which reads commands from a tcp/ip stream and then executes those commands. Commands consist of a method call on an object also identified by an id int the command. You could think of a command as the information of an element id (addressing element we want to call a the command on) and an command id (addressing the method which should be called on the element). Additionally we also have the issue that we need to check some kind of permissions on every command and also how this command should be executed. (Should it be started in a new `Thread`, etc.)

An example of how such a command call could look like would be this:

```
class Callee
{
    public void RegularCall(int command, parameters)
    {
        switch (command)
        {
            case 1: // Comand #1
                // Check if the permissions allow this command to be called.
                // Check if it should be outsourced to the ThreadPool and
                // call it accordingly. +Other Checks.
                // Finally execute command #1.
                break;
            case 2: // Comand #2
                // Check if the permissions allow that command to be called.
                // Check if it should be outsourced to the ThreadPool and
                // call it accordingly. +Other Checks.
                // Finally execute command #2.
                break;
            // Many more cases with various combinations of permissions and
            // Other flags.
        }
    }
}
```

And somewhere:

```
static Dictionary<int, Callee> callees = new Dictionary<int, Callee>();

static void CallMethod(int elementId, int commandId, parameters)
{
    callees[elementId].RegularCall(commandId, parameters);
}
```

However, this approach is some kind of unelegant:

- This may be error prone due to copying the same code over and over again.
- In some circumstances it's hard to see, which commands exist and what their flags are.

- The command method is full of checks which could have made outside the method.

My first approach was by using reflection, which would have looked that way:

```
class Callee
{
    [Command(1)]
    [Permissions(0b00111000)]
    [UseThreadPool]
    public void SpeakingNameForCommand1(parameters)
    {
        // Code for command #1.
    }

    [Command(2)]
    [Permissions(0b00101011)]
    public void SpeakingNameForCommand2(parameters)
    {
        // Code for command #2.
    }

    // Again, many more commands.
}
```

This code must have been initialized with some reflection heavy code:

1. Find all classes which may represent an element.
2. Find all methods which have a command attribute, etc.
3. Store all those information in a dictionary, including the corresponding `MethodInfo`.

A call of a received command would look like this, where `CommandInfo` is a class containing all the information required for the call (`MethodInfo`, run in `ThreadPool`, permissions...):

```
static Dictionary<int, CommandInfo> commands = new Dictionary<int, CommandInfo>
();

static void CallMethod(int elementId, int commandId)
{
    CommandInfo ci = commands[commandId];

    if (ci.Permissions != EVERYTHING_OK)
        throw ...;

    if (ci.UseThreadPool)
        ThreadPool.Queue...(delegate { ci.MethodInfo.Invoke(callees[elementId],
params); });
    else
        ci.MethodInfo.Invoke(callees[elementId], params);
}
```

When I micro-benchmark this, the call to `MethodInfo.Invoke` is about 100x slower than the direct call. **The question is: Is there a faster way of calling those "command" methods, without losing the elegance of the attributes defining the way how those commands should be called?**

I also tried deriving a delegate from the `MethodInfo`. However, this didn't work well, because I need to be able to call the method on any instance of the `Callee` class and don't want to reserve the memory for the delegate for every possible element * commands. (There will be many elements.)

Just to make this clear: `MethodInfo.Invoke` is 100x slower than the function call including the `switch` / `case` statement. This excludes the time to walk over all classes, methods and attributes, because those informations have already been prepared.

Please refrain from informing me about other bottlenecks like the network. They are not the issue. And they are no reason to use slow calls on another location in the code. Thank you.

c# .net performance reflection

Share Edit Follow Flag

edited Feb 17, 2019 at 21:56

asked Feb 17, 2019 at 21:00

Matthias
1,056 1 8 28

3 If you're reading from a network stream, the bottleneck will most likely be your network, not `MethodInfo.Invoke`. Optimizing the latter will probably make no observable difference in performance. – Michael Liu Feb 17, 2019 at 21:05

1 In addition to Michael's answer - All answers will likely boil down to "Not invoking methods dynamically are going to be the fastest" – Patrick Hollweck Feb 17, 2019 at 21:11

1 What Michael Liu said, but you might also consider so-called open delegates for delegates to instance methods (blog.slaks.net/2011/06/open-delegates-vs-closed-delegates.html), if the number of possible "normal"/closed delegates (delegates containing a reference to the target object instance) could be an issue... – user2819245 Feb 17, 2019 at 21:12

2 I really can't help my self, but why are you even arguing about pseudo-code. And how will you know that something is wrong or not? You even don't have all the necessary information to give that kind of a statement. – Matthias Feb 18, 2019 at 2:29

2 Hey, it seems Scharle saved me from writing an answer... :-) – user2819245 Feb 18, 2019 at 18:19

2 Answers

Sorted by: Reset to default Trending (recent votes count more)

4

You could use open delegates, which are about ten times faster than `MethodInfo.Invoke`. You would create such a `delegate` from a `MethodInfo` like this:

```
delegate void OpenCommandCall(Callee element, parameters);

OpenCommandCall occDelegate =
(OpenCommandCall)Delegate.CreateDelegate(typeof(OpenCommandCall), methodInfo));
```

You then would call this delegate like:

```
occDelegate.Invoke(callee, params);
```

Where `callee` is the element you want to call the method on, `methodInfo` is the `MethodInfo` of the method and `parameters` are a placeholder for various other parameters.

Share Edit Follow Flag

answered Feb 17, 2019 at 21:52
Scharle
160 2 7

Comments

Add a comment

3

Maybe you want to give a try to the ObjectMethodExecutor

According to Hanselman:

> If you ever need to invoke a method on a type via reflection and that method could be async, we have a helper that we use everywhere in the ASP.NET Core code base that is highly optimized and flexible called the ObjectMethodExecutor.
>
> The team uses this code in MVC to invoke your controller methods. They use this code in SignalR to invoke your hub methods. It handles async and sync methods. It also handles custom awaitables and F# async workflows

Share Edit Follow Flag

edited Jun 20, 2020 at 9:12
Community Bot
1 1

answered Feb 17, 2019 at 22:07
Vlad
852 1 11 23

Comments

Add a comment

c# .net performance reflection

See similar questions with these tags.