Article
Sean Klingensmith · Nov 18, 2016 10m read

Exploring Code Generation with Caché Method Generators

As a developer, you have probably spent at least some time writing repetetive code. You may have even found yourself wishing you could generate the code programmatically. If this sounds familiar, this article is for you!

We'll start with an example. Note: the following examples use the %DynamicObject interface, which requires Caché 2016.2 or later. If you are unfamiliar with this class, check out the documentation here: Using JSON in Caché. It's really cool!

Example

You have a %Persistent class that you use to store data. Now, suppose that you are ingesting some data in JSON format, using the %DynamicObject interface. How do you map the %DynamicObject structure to your class? One solution is to simply write code to copy the values over directly:

```
Class Test.Generator Extends %Persistent
{
Property SomeProperty As %String;

Property OtherProperty As %String;

ClassMethod FromDynamicObject(dynobj As %DynamicObject) As Test.Generator
{
    set obj = ..%New()
    set obj.SomeProperty = dynobj.SomeProperty
    set obj.OtherProperty = dynobj.OtherProperty
    quit obj
}
}
```

However, this will get tedious (not to mention difficult to maintain) if there are many properties, or if you use this pattern for multiple classes. This is where method generators can help! Simply put, when using a method generator, instead of writing the code for a given method, you write some code that the class compiler will run to generate the code for the method. Does this sound confusing? It really isn't. Let us look at an example:

```
Class Test.Generator Extends %Persistent
{
ClassMethod Test() As %String [ CodeMode = objectgenerator ]
{
    do %code.WriteLine(" write ""This is a method Generator!"",!")
    do %code.WriteLine(" quit ""Done!""")

    quit $$$OK
}
}
```

We use the parameter CodeMode = objectgenerator to indicate that the current method is a method generator, and not a normal method. What does this method do? In order to debug method generators, it is useful to look at the generated code for the class. In our case, this will be in an INT routine named Test.Generator.1.INT. You can open this in Studio by typing Ctrl+Shift+V, or you can just open the routine from the Studio "Open" dialog, or from Atelier.

In the INT code, you can find the implementation for this method:

```
zTest() public {
 write "This is a method Generator!",!
 quit "Done!" }
```

As you can see, the method implementation simply contains the text that is written to the %code object. %code is a special type of stream object (%Stream.MethodGenerator). The code written to this stream can contain any code valid in a MAC routine, including macros, preprocessor directives, and embedded SQL. There are a couple of things to keep in mind when working with method generators:

- The method signature applies to the target method you are generating. The generator code should always return a Status code indicating either success or an error condition.
- The code written to %code must be valid ObjectScript (method generators with other language modes are outside the scope of this article). This means, among other things, that lines containing commands must start with whitespace. Note that the two WriteLine() calls in the example begin with a space.

In addition to the %code variable (representing the generated method), the compiler makes the metadata for the current class available in the following variables:

- %class
- %method
- %compiledclass
- %compiledmethod
- %parameter

The first four of these variables are instances of %Dictionary.ClassDefinition, %Dictionary.MethodDefinition, %Dictionary.CompiledClass %Dictionary.CompiledMethod, respectively. %parameter is a subscripted array of parameter names and values defined in the class.

The main difference (for our purposes) between %class and %compiledclass is that %class only contains metadata for class members (properties, methods, etc.) defined in the current class. %compiledclass will contain these members, but will also contain metadata for all inherited members. In addition, type information referenced from %class will appear exactly as specified in the class code, whereas types in %compiledclass (and %compiledmethod) will be expanded to the full classname. For instance, %String will be expanded to %Library.String, and class names without a package specified will be expanded to the full Package.Class name. You can see the class reference for these classes for further information.

Using this information, we can build a method generator for our %DynamicObject example:

```
ClassMethod FromDynamicObject(dynobj As %DynamicObject) As Test.Generator [ CodeMode
= objectgenerator ]
{
    do %code.WriteLine(" set obj = ..%New()")
    for i=1:1:%class.Properties.Count() {
```

```
        set prop = %class.Properties.GetAt(i)
        do %code.WriteLine(" if dynobj.%IsDefined("""_prop.Name_""") {")
        do %code.WriteLine("   set obj."_prop.Name_" = dynobj."_prop.Name)
        do %code.WriteLine(" }")
    }

    do %code.WriteLine(" quit obj")
    quit $$$OK
}
```

This creates the following code:

```
zFromDynamicObject(dynobj) public {
 set obj = ..%New()
 if dynobj.%IsDefined("OtherProperty") {
   set obj.OtherProperty = dynobj.OtherProperty
 }
 if dynobj.%IsDefined("SomeProperty") {
   set obj.SomeProperty = dynobj.SomeProperty
 }
 quit obj }
```

As you can see, this generates code to set each property defined in this class. Our Implementation excludes inherited properties, but we could easily include them by using %compiledclass.Properties instead of %class.Properties. We also added a check to see if the property exists in the %DynamicObject before attempting to set it. This isn't strictly necessary, since referencing a property that does not exists from a %DynamicObject will not result in an error, but it is helpful if any of the properties in the class define a default value. If we didn't perform this check, the default value would always be overwritten by this method.

Method generators can be very powerful when combined with inheritance. We can take the FromDynamicObject() method generator and put it in an abstract class. Now if we want to write a new class that needs to be able to be deserialized from a %DynamicObject, all we need to do is to extend this class to enable this functionality. The class compiler will run the method generator code when compiling each subclass, creating a custom implementation for that class.

Debugging Method Generators

Basic debugging

Using method generators adds a level indirection to your programming. This can cause some problems when trying to debug our generator code. Let's look at an example. Consider the following method:

```
Method PrintObject() As %Status [ CodeMode = objectgenerator ]
{
    if (%class.Properties.Count()=0)&&($get(%parameter("DISPLAYEMPTY"),0)) {
        do %code.WriteLine(" write ""{}"",!")
    } elseif %class.Properties.Count()=1 {
        set pname = %class.Properties.GetAt(1).Name
        do %code.WriteLine(" write ""{ "_pname_": ""_.."_pname_"_""}"",!")
    } elseif %class.Properties.Count()>1 {
        do %code.WriteLine(" write ""{"",!")
        for i=1:1:%class.Properties.Count() {
            set pname = %class.Properties.GetAt(i).Name
            do %code.WriteLine(" write """_pname_": ""_.."_pname_",!")
```

```
        }
        do %code.WriteLine(" write ""}""")
    }

    do %code.WriteLine(" quit $$$OK")
    quit $$$OK
}
```

This is a simple method designed to print the contents of an object. It will output the objects using a different format depending on the number of properties: an object with multiple properties will be printed on multiple lines, while an object with zero or one properties will be printed on one line. Additionally the object introduces a Parameter DISPLAYEMTPY, which will control whether to suppress output for objects with zero properties. However, there is a problem with the code. For a class with zero properties, the object isn't being output correctly:

```
TEST>set obj=##class(Test.Generator).%New()

TEST>do obj.PrintObject()

TEST>
```

We expect this to output an empty object "{}", not nothing. To debug this we can look in the INT code to see what is happening. However, upon opening the INT code, you find that there is no definition for zPrintObject()! Don't take my word for it, compile the code and look for yourself. Go on... I'll wait.

OK. Back? What's going on here? Astute readers may have figured out the initial problem: There is a typo in the first clause of the IF statement The default for the DISPLAYEMPTY parameter should be 1 not 0. It should be: $get(%parameter("DISPLAYEMPTY"),1) not $get(%parameter("DISPLAYEMPTY"),0). This explains the behavior. But why wasn't the method in the INT code? It was still executable. We didn't get a <METHOD DOES NOT EXIST> error; the method just didn't do anything. Now that we see the mistake, let's look at what the code *would* have been if it were in the INT code. Since we failed to satisfy any of the conditions in the if ... elseif ... construct the code would simply be:

```
zPrintObject() public {
    quit 1 }
```

Notice that this code doesn't actually do anything; it just returns a literal value. It turns out that the Caché class compiler is pretty clever. In certain situations it can detect that the code for a method doesn't need to be executed, and can optimize away the INT code for the method. This is a great optimization, since dispatching from the kernel to the INT code can involve a fair amount of overhead, especially for simple methods.

Note that this behavior isn't specific to method generators. Try compiling the following method, and looking for it in the INT code:

```
ClassMethod OptimizationTest() As %Integer
{
    quit 10
}
```

Looking in the INT code can be very helpful when debugging your method generator code. This will tell you what the generator really produced. However, you have to be careful to realize that there are some cases when the generated code will not appear in the INT code. If this is happening unnexpectedly, there is likely a bug in the generator code that is causing it to fail to generate any meaningful code.

Using a debugger

As we saw, if there is a problem with the generated code, we can see it by looking at the INT code. We can also debug the method normally using ZBREAK or the Studio debugger. You might be wondering if there is a way to debug the method generator code itself. Of course, you can always add "write" statements to the method generator or set debug globals like a caveman. But there has to be a better way, right?

The answer is "Yes", but in order to understand how, we need to get some background on how the class compiler works. Broadly speaking, when the class compiler compiles a class it will first parse the class definition and generate the metadata for the class. It is essentially generating the data for the %class and %compiledclass variables we discussed earlier. Next it generates the INT code for all the methods. During this step, it will create a separate routine to contain the generation code for all the method generators. This routine is named <classname>.G1.INT. It then executes the code in the *.G1 routine to generate the code for the methods, and stores them in the <classname>.1.INT routine with the rest of the class's methods. It can then compile this routine and voila! We have our compiled class! This is of course a gross simplification of a very complex piece of software - but it will do for our purposes.

This *.G1 routine sounds interesting. Let's take a look!

```
    ;Test.Generator3.G1
    ;(C)InterSystems, method generator for class Test.Generator3.  Do NOT edit.
    Quit
    ;
FromDynamicObject(%class,%code,%method,%compiledclass,%compiledmethod,%parameter) pub
lic {
    do %code.WriteLine(" set obj = ..%New()")
    for i=1:1:%class.Properties.Count() {
        set prop = %class.Properties.GetAt(i)
        do %code.WriteLine(" if dynobj.%IsDefined("""_prop.Name_""") {")
        do %code.WriteLine("   set obj."_prop.Name_" = dynobj."_prop.Name)
        do %code.WriteLine(" }")
    }
    do %code.WriteLine(" quit obj")
    quit 1
 Quit 1 }
```

You may be used to editing the INT code for a class and adding debug code. Normally that's fine, if a little primitive. However, that is not going to work here. In order to execute this code, we need to recompile the class. (It is called by the class compiler, after all.) But recompiling the class will regenerate this routine, wiping out any changes we made. Fortunately we can use ZBreak or the Studio debugger to walk through this code. Since we now know the name of the routine, using ZBreak is pretty straightforward:

```
TEST>zbreak FromDynamicObject^Test.Generator.G1

TEST>do $system.OBJ.Compile("Test.Generator","ck")

Compilation started on 11/14/2016 17:13:59 with qualifiers 'ck'
Compiling class Test.Generator
FromDynamicObject(%class,%code,%method,%compiledclass,%compiledmethod,%parameter) pub
l
            ^
ic {
<BREAK>FromDynamicObject^Test.Generator.G1
TEST 21e1>write %class.Name
```

```
Test.Generator
TEST 21e1>
```

Using the Studio Debugger is also simple. You can set a breakpoint in the *.G1.MAC routine, and configure the debug target to invoke $System.OBJ.Compile() on the class:

```
$System.OBJ.Compile("Test.Generator","ck")
```

And now you are up and debugging.

Conclusion

This article has been a brief overview of method generators. For further information, please check out the documentation below:

- Defining Method and Trigger Generators
- For further information on the %class and %compiledclass objects see:
  - Using the %Dictionary Classes
  - %Dictionary.ClassDefinition class reference
  - %Dictionary.CompiledClass class reference

#Best Practices #Compiler #Object Data Model #Caché #InterSystems IRIS