Article
Lorenzo Scalese · Juin 8, 2022 11m de lecture

Analyse de la génération de code avec la Method Generator

En tant que développeur, vous avez probablement passé au moins un certain temps à écrire un code répétitif. Vous vous êtes peut-être même retrouvé à souhaiter pouvoir générer ce code de manière programmatique. Si vous êtes dans cette situation, cet article est pour vous !

Nous allons commencer par un exemple. Note : les exemples suivants utilisent l'interface %DynamicObject, qui nécessite Caché 2016.2 ou une version supérieure. Si vous n'êtes pas familier avec cette classe, consultez la documentation ici : Utiliser JSON dans Caché. C'est vraiment génial !

Exemple

Vous avez une classe %Persistent que vous utilisez pour stocker des données. Maintenant, supposons que vous allez saisir des données au format JSON en utilisant l'interface %DynamicObject. Comment faire correspondre la structure %DynamicObject à votre classe ? Une solution consiste à écrire du code pour copier directement les valeurs :

```
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
}
}
```

Cependant, si les propriétés sont nombreuses ou si vous utilisez ce modèle pour plusieurs classes, cela devient fastidieux (et difficile à maintenir). C'est là que les Method Generators peuvent vous aider ! En termes simples, lorsqu'on utilise une Method Generator, au lieu d'écrire le code d'une méthode donnée, on écrit du code que le compilateur de la classe exécutera pour générer le code de la méthode. Cela vous semble-t-il gênant ? Non, pas du tout. Prenons un exemple :

```
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
}
```

```
}
```

Nous utilisons le paramètre CodeMode = objectgenerator pour indiquer que la méthode courante est une Method Generator, et non une méthode classique. Comment fonctionne cette méthode ? Afin de déboguer les Method Generators, il est utile de regarder le code généré pour la classe. Dans notre cas, il s'agit d'une routine INT nommée Test.Generator.1.INT. Vous pouvez l'ouvrir dans Studio en tapant Ctrl+Shift+V, ou vous pouvez simplement ouvrir la routine depuis la boîte de dialogue "Open" de Studio, ou depuis l'Atelier.

Dans le code INT, vous pouvez trouver l'implémentation de cette méthode :

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

Comme vous pouvez le voir, l'implémentation de la méthode contient simplement le texte qui est écrit dans l'objet %code. %code est un objet de type spécial de flux (%Stream.MethodGenerator). Le code écrit dans ce flux peut contenir n'importe quel code valide dans une routine MAC, y compris des macros, des directives de préprocesseur, et du SQL intégré. Il y a deux choses à garder à l'esprit quand on travaille avec des Method Generators :

- La signature de la méthode s'applique à la méthode cible que vous allez générer. Le code du générateur doit toujours renvoyer un code d'état indiquant soit un succès, soit une erreur.
- Le code écrit dans %code doit être un ObjectScript valide (les générateurs de méthodes avec d'autres modes de langage ne sont pas concernés par cet article). Cela signifie, entre autres, que les lignes contenant des commandes doivent commencer par un espace. Notez que les deux appels WriteLine() dans l'exemple commencent par un espace.

En plus de la variable %code (représentant la méthode générée), le compilateur rend les métadonnées de la classe courante disponibles dans les variables suivantes :

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

Les quatre premières variables sont des instances de %Dictionary.ClassDefinition, %Dictionary.MethodDefinition, %Dictionary.CompiledClass %Dictionary.CompiledMethod, respectivement. %parameter est un tableau souscrit de noms et de valeurs de paramètres définis dans la classe.

La principale différence (pour nos besoins) entre %class et %compiledclass est que %class ne contient que les métadonnées des membres de la classe (propriétés, méthodes, etc.) définis dans la classe courante. %compiledclass contiendra ces membres, mais aussi les métadonnées de tous les membres hérités. De plus, les informations de type référencées à partir de %class apparaîtront exactement comme spécifié dans le code de la classe, alors que les types dans %compiledclass (et %compiledmethod) seront étendus au nom complet de la classe. Par exemple, %String sera développé en %Library.String, et les noms de classes sans package spécifié seront développés en nom complet Package.Class. Vous pouvez consulter la référence de ces classes pour plus d'informations.

En utilisant ces informations, nous pouvons construire une Method Generator pour notre exemple %DynamicObject :

```
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
}
```

Le code suivant est ainsi créé :

```
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 }
```

Comme vous pouvez le voir, cela génère du code pour configurer chaque propriété définie dans cette classe. Notre implémentation exclut les propriétés héritées, mais nous pourrions facilement les inclure en utilisant %compiledclass.Properties au lieu de %class.Properties. Nous avons également ajouté une vérification pour voir si la propriété existe dans le %DynamicObject avant de tenter de la définir. Ce n'est pas strictement nécessaire, puisque la référence à une propriété qui n'existe pas dans un %DynamicObject n'entraînera pas d'erreur, mais c'est utile si l'une des propriétés de la classe définit une valeur par défaut. Si nous n'effectuons pas cette vérification, la valeur par défaut sera toujours surchargée par cette méthode.

Les Method Generators peuvent être très puissants lorsqu'ils sont combinés à l'héritage. Nous pouvons prendre le générateur de méthodes FromDynamicObject() et le placer dans une classe abstraite. Maintenant, si nous voulons écrire une nouvelle classe qui doit être capable d'être désérialisée à partir d'un %DynamicObject, tout ce que nous devons faire est d'étendre cette classe pour activer cette fonctionnalité. Le compilateur de classes exécutera le code de la Method Generator lors de la compilation de chaque sous-classe, créant ainsi une implémentation personnalisée pour cette classe.

Débogage des générateurs de méthodes

Débogage de base

L'utilisation de Method Generator permet d'ajouter un niveau d'indirection à votre programmation. Cela peut poser quelques problèmes lorsqu'on essaie de déboguer le code du générateur. Prenons un exemple. Considérons la méthode suivante :

```
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
}
```

Il s'agit d'une méthode simple conçue pour imprimer le contenu d'un objet. Elle affiche les objets dans un format différent selon le nombre de propriétés : un objet avec plusieurs propriétés sera imprimé sur plusieurs lignes, tandis qu'un objet avec zéro ou une propriété sera imprimé sur une ligne. De plus, l'objet contient un paramètre DISPLAYEMTPY, qui permet de supprimer ou non l'affichage des objets ayant zéro propriété. Cependant, il y a un problème avec le code. Pour une classe avec zéro propriété, l'objet n'est pas affiché correctement :

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

TEST>do obj.PrintObject()

TEST>
```

Nous nous attendons à ce que cela produise un objet vide "{}", et non un rien. Pour déboguer cela, nous pouvons regarder dans le code INT pour voir ce qui se passe. Cependant, en ouvrant le code INT, vous découvrez qu'il n'y a pas de définition pour zPrintObject() ! Ne me croyez pas sur parole, compilez le code et regardez par vous-même. Allez-y... Je vais attendre.

OK. Retour ? Qu'est-ce qui se passe ici ? Les lecteurs astucieux ont peut-être trouvé le problème initial : il y a une faute de frappe dans la première clause de l'instruction IF. La valeur par défaut du paramètre DISPLAYEMPTY devrait être 1 et non 0. Il devrait être le suivant : $get(%parameter("DISPLAYEMPTY"),1) not $get(%parameter("DISPLAYEMPTY"),0). Ceci explique le comportement. Mais pourquoi la méthode n'était-elle pas dans le code INT ? Il était encore exécutable. Nous n'avons pas eu d'erreur <METHOD DOES NOT EXIST> ; la méthode n'a simplement rien fait. Maintenant que nous voyons l'erreur, regardons ce que le code *aurait* été s'il avait été dans le code INT. Puisque nous n'avons satisfait à aucune des conditions de la construction if ... elseif ..., le code aurait été simplement comme suit :

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

Remarquez que ce code ne fonctionne pas réellement ; il renvoie simplement une valeur littérale. Il s'avère que le compilateur de classe Caché est assez intelligent. Dans certaines situations, il peut détecter que le code d'une méthode n'a pas besoin d'être exécuté, et peut optimiser le code INT de la méthode. Il s'agit d'une excellente optimisation, car la répartition du noyau vers le code INT peut impliquer une quantité considérable de surcharge, en particulier pour les méthodes simples.

Notez que ce comportement n'est pas spécifique aux Method Generators. Essayez de compiler la méthode

suivante, et cherchez-la dans le code INT :

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

Il peut être très utile de vérifier le code INT pour déboguer le code de votre Method Generator. Cela vous permettra de savoir ce que le générateur a réellement produit. Cependant, vous devez être attentif au fait qu'il y a des cas où le code généré n'apparaîtra pas dans le code INT. Si cela se produit de manière inattendue, il y a probablement un bug dans le code du générateur qui l'empêche de générer un code significatif.

Utilisation du débogueur

Comme nous l'avons vu, s'il y a un problème avec le code généré, nous pouvons le voir en regardant le code INT. Nous pouvons également déboguer la méthode normalement en utilisant ZBREAK ou le débogueur de Studio. Vous vous demandez peut-être s'il existe un moyen de déboguer le code de la Method Generator elle-même. Bien sûr, vous pouvez toujours ajouter des instructions "write" à la Method Generator ou définir des globaux de débogage comme un homme des cavernes. Mais il doit bien y avoir un meilleur moyen, n'est-ce pas ?

La réponse est "Oui", mais pour comprendre la manière dont cela se passe, nous devons connaître le fonctionnement du compilateur de classes. En gros, lorsque le compilateur de classes compile une classe, il va d'abord analyser la définition de la classe et générer les métadonnées de la classe. Il s'agit essentiellement de générer les données pour les variables %class et %compiledclass dont nous avons parlé précédemment. Ensuite, il génère le code INT pour toutes les méthodes. Au cours de cette étape, il va créer une routine séparée pour contenir le code de génération de tous les Method Generators. Cette routine est nommée <classname>.G1.INT. Il exécute ensuite le code dans la routine *.G1 pour générer le code des méthodes, et les stocke dans la routine <classname>.1.INT avec le reste des méthodes de la classe. Il peut ensuite compiler cette routine et voilà ! Nous avons notre classe compilée ! Il s'agit bien sûr d'une simplification énorme d'un logiciel très complexe, mais cela suffira pour nos besoins.

Cette routine *.G1 semble intéressante. Jetons-y un coup d'œil !

```
    ;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 }
```

Vous êtes peut-être habitué à modifier le code INT d'une classe et à ajouter du code de débogage. Normalement, c'est bien, même si c'est un peu primitif. Cependant, cela ne va pas fonctionner ici. Afin d'exécuter ce code, nous devons recompiler la classe. (C'est le compilateur de la classe qui l'appelle, après tout.) Mais recompiler la classe régénérera cette routine, effaçant toutes les modifications que nous avons apportées. Heureusement, nous

pouvons utiliser ZBreak ou le débogueur de Studio pour parcourir ce code. Puisque nous connaissons maintenant le nom de la routine, l'utilisation de ZBreak est assez simple :

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

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

La compilation a commencé le 14/11/2016 17:13:59 avec les qualificatifs '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>
```

L'utilisation du débogueur de Studio est également simple. Vous pouvez définir un point de contrôle dans la routine *.G1.MAC, et configurer la cible de débogage pour qu'elle invoque $System.OBJ.Compile() sur la classe :

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

Et maintenant vous vous lancez dans le débogage.

Conclusion

Cet article a été un bref aperçu des générateurs de méthodes. Pour de plus amples informations, veuillez consulter la documentation ci-dessous :

- Defining Method and Trigger Generators
- Pour plus d'informations sur les objets %class et %compiledclass, consultez :
  - Using the %Dictionary Classes
  - %Dictionary.ClassDefinition class reference
  - %Dictionary.CompiledClass class reference

#Bonnes pratiques #Compilateur #Object Data Model #Caché #InterSystems IRIS

URL de la source:https://fr.community.intersystems.com/post/analyse-de-la-g%C3%A9n%C3%A9ration-de-code-avec-la-method-generator