記事
Toshihiko Minamoto · 2021年3月31日 13m read

Caché のメソッドジェネレータを使ったコード生成の検証

デベロッパーの方なら、反復的なコードを書いた経験があると思います。
プログラムを使ってコードを生成できたら楽なのに、と考えたことがあるかもしれません。
まさに自分のことだと思った方、ぜひこの記事をお読みください！

まずは例をお見せします。 注意: 次の例で使用する %DynamicObject インターフェースは Caché 2016.2 以上のバージョンが必要です。 このクラスに馴染みのない方は、Using JSON in Caché と題したドキュメンテーションをお読みください。 とても重宝すると思います！

例

データを保管するために使う %Persistent というクラスがあります。 %DynamicObject インターフェースを使い、データを JSON 形式で取り込むとしましょう。 どうすれば %DynamicObject 構造をクラスにマッピングできると思いますか？ ソリューションの 1 つに、値を直接コピーするコードを書くという方法があります。

```
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
}
}
```

しかし、プロパティの数が多かったり、このパターンを複数のクラスに使ったりすると、少し面倒なことになります (もちろん管理も大変です)。 それを解決するのがメソッドジェネレータです！ 簡単に言うと、メソッドジェネレータを使うときは、特定のメソッドのコードを書く代わりに、クラスのコンパイラが実行するコードを書き、それによりメソッドのコードを生成します。 少しややこしいでしょうか？ いたって単純なんですよ。
では、例を一つお見せしましょう。

```
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
}
```

```
}
```

CodeMode = objectgenerator というパラメーターを使い、現在のメソッドはメソッドジェネレータであり、普通のメソッドではないことを示しています。 このメソッドの働きですが、
メソッドジェネレータをデバッグするには、クラスの生成されたコードを見ると便利です。
今回の例で言うと、Test.Generator.1.INT と名付けた INT ルーチンがそれに当たります。 これを開くには、Studio で「Ctrl+Shift+V」と入力してもいいですし、Studio の「Open」ダイアログまたは Atelier から開くこともできます。

INT コードを見ると、このメソッドが実装されているのが分かります。

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

見てお分かりの通り、この実装は %code オブジェクトに書き込まれるテキストを含む単純なものです。 %code は、特殊なタイプのストリームオブジェクトです(%Stream.MethodGenerator)。
このストリームに書き込まれるコードには、マクロやプリプロセッサディレクティブ、埋め込まれた SQL など、MAC ルーチンで有効なコードであれば、何でも含めることができます。
メソッドジェネレータを使用するにあたり、いくつか頭に入れておきたいことがあります。

- メソッドシグネチャは、生成するターゲットメソッドに適用される。
  ジェネレータのコードは、常に成功またはエラー状況を示すステータスコードを返すものである。
- %code に書き込まれるコードは有効な ObjectScript でなければいけない
  (他の言語モードを持つメソッドジェネレータは本記事の範囲外です)。
  つまり、特に重要なこととして、コマンドを含む行はスペースから始めなければいけません。 例にある WriteLine() の呼び出しは、2 つともスペースで始まっています。

%code の変数 (生成されたメソッド)
以外にも、コンパイラは現在のクラスのメタデータを以下の変数が使用できます。

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

最初の 4 つは、それぞれ
%Dictionary.ClassDefinition、%Dictionary.MethodDefinition、%Dictionary.CompiledClass
%Dictionary.CompiledMethod のインスタンスです。 %parameter
は、クラスで定義されたパラメータ名とキーで構成される添え字付き配列です。

(今回の目的において) %class と %compiledclass の主な違いは、%class
には現在のクラスで定義されているクラスメンバー (プロパティやメソッドなど)
のメタデータだけが含まれている点です。 一方の %compiledclass
には、これらのメンバー以外にも、継承されたすべてのメンバーのメタデータが含まれます。 また、%class から参照される型の情報は、クラスコードで指定されている通りに表示される一方で、%compiledclass (および %compiledmethod) の型は完全なクラス名に展開されます。 例えば、%String は %Library.String
に展開され、パッケージが指定されていないクラス名は Package.Class
のように完全なクラス名に展開されます。 詳細は、これらのクラスのクラスリファレンスをご覧ください。

この情報を使えば、%DynamicObject 用にメソッドジェネレータを構築することができます。

```
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
}
```

これにより、以下のコードが生成されます。

```
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 }
```

ご覧のとおり、このクラスで定義されている各プロパティを set するコードが生成されます。
この実装では、継承されたプロパティを除外していますが、%class.Properties の代わりに
%compiledclass.Properties を使えば簡単に含めることができます。 また、プロパティを set
しようと試みる前に、%DynamicObject にプロパティが存在するかどうかをチェックするコードも追加しました。
存在しないプロパティを %DynamicObject から参照してもエラーは出ないので絶対に必要な訳ではありませんが
、クラス内のプロパティのいずれかがデフォルト値を定義している場合は便利です。
このチェックを行わなければ、デフォルト値はいつもこのメソッドによって上書きされます。

メソッドジェネレータは継承と組み合わせて使うと大きな威力を発揮します。 FromDynamicObject()
メソッドジェネレータは、抽象クラスに置くことができます。 なお、%DynamicObject
から逆シリアル化できる新しいクラスを作成するのであれば、このクラスを拡張してこの機能を有効化するだけで
OK です。 クラスのコンパイラは、各サブクラスをコンパイルするときに、メソッドジェネレータのコードを実行
し、そのクラスの実装をカスタマイズします。

メソッドジェネレータのデバッグ

基本的なデバッグ作業

メソッドジェネレータを使用すると、プログラミングの間接参照のレベルが増えてしまいます。
これにより、ジェネレータのコードをデバッグする際に問題が起こる場合があります。 それでは、1
つ例を見てみましょう。 次のメソッドをご覧ください。

```
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
}
```

これは、オブジェクトの中身を出力するだけの単純なメソッドです。 オブジェクトは、プロパティの数によって異なる形式で出力されます。具体的には、複数のプロパティを持つオブジェクトは複数の行に渡って出力され、プロパティを持たない、または 1 つしか持たないオブジェクトは 1 つの行に出力されます。 また、オブジェクトは DISPLAYEMTPY というパラメーターを導入しています。これは、プロパティを持たないオブジェクトの出力を抑制するかしないかを制御するものです。 しかし、このコードには問題点があります。
プロパティを持たないクラスでは、オブジェクトが正しく出力されていません。

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

TEST>do obj.PrintObject()

TEST>
```

ここでは、何も出力されないのではなく、空のオブジェクト "{}" が出力されるはずなのです。
これをデバッグするに、INT コードの中身を確認します。 ところが、INT コードを開いてみると、なんと zPrintObject() の定義が見当たらないのです！
私の言うことを鵜呑みにせず、コードをコンパイルしてご自身の目でお確かめください。 どうぞ...
終わるまでお待ちします。

はい、 終わりましたでしょうか？ 何か分かりましたか？ 鋭い方なら、1 つ目の問題の原因が分かったのではないでしょうか。そうです、IF 文の最初の節に入力ミスがあります。DISPLAYEMPTY パラメーターのデフォルト値は 0 ではなく、1 でなければいけません。
正しくは、$get(%parameter("DISPLAYEMPTY"),1)。$get(%parameter("DISPLAYEMPTY"),0) は間違いです。
これで原因がはっきりしましたね。 でも、どうして INT コードにメソッドがなかったのでしょう？
でも、実行はできましたよね。 <METHOD DOES NOT EXIST>
エラーは出なかったし。メソッドは全く何もしなかったのです。 ミスが解明したところで、このメソッドが INT コードにあればどうようなコードになっていたかを見てみましょう。 if ... else if ...
コンストラクトの条件を１つも満たしていないので、コードは単純に以下のようなります。

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

このコードは、リテラル値を返す以外には、何もしないことに注目してください。 Caché のクラスのコンパイラは非常に賢いことが分かりました。

特定の状況では、メソッドのコードを実行する必要がないことに気付き、INT
コードをメソッドに合わせて最適化できるのです。
これは紛れもなく素晴らしい最適化機能です。なぜなら、主にシンプルなメソッドの場合は、カーネルから INT
コードにディスパッチすると膨大なオーバーヘッドが生じるからです。

この動作は、メソッドジェネレータ固有のものではないことに注意してください。
次のメソッドをコンパイルしてから、INT コードの中で探してみてください。

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

メソッドジェネレータのコードをデバッグするときは、INT コードを確認すると非常に便利です。
ジェネレータによって実際に作成されたものを確認できます。 但し、生成されたコードが INT
コードに表示されない場合があるので、注意が必要です。 そういった予想外の事象が発生する場合は、ジェネレータのコードにバグがあり、ジェネレータが有意義なコードを生成できない原因となっていることが考えられます。

デバッガーの使用について

先ほど説明しましたが、生成されたコードに問題がある場合は、INT コードを見れば確認できます。
また、ZBREAK や Studio のデバッガーを使って、メソッドをデバッグすることもできます。 メソッドジェネレータのコードそのものをデバッグする方法はないだろうか、と気になっている方もいるのではないでしょうか。
もちろん、いつでもメソッドジェネレータに「write」式を追加したり、caveman
のようなデバッググローバルを設定したりできます。 でも、もっといい方法があるはずだと思いませんか？

そうです、あるんです。しかし、その方法を理解するには、まずクラスのコンパイラーが機能する仕組みを理解する必要があります。 大まかに説明すると、クラスのコンパイラーは、クラスをコンパイルするとき、まず最初にクラスの定義を解析して、そのクラス用にメタデータを生成します。 基本的には、先ほど説明した %class 変数と %compiledclass 変数用にデータを生成していることになります。 次に、すべてのメソッドに対し INT
コードを生成します。
この段階で、すべてのメソッドジェネレータの生成コードを格納する個別のルーチンを作成します。
このルーチンは、<classname>.G1.INT と呼ばれています。 そして、*.G1
ルーチンのコードを実行してメソッドのコードを生成し、そのコードをクラスの残りのメソッドと一緒に
<classname>.1.INT ルーチンに保管します。 そして、このルーチンをコンパイルすると、
コンパイルされたクラスが作成されます！ もちろん、これは非常に複雑なソフトウェアを極端に単純化したものですが、この記事の目的を果たすには十分です。

この *.G1 ルーチンは面白そうですね。 ではその中身を見てみましょう！

```
    ;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 }
```

クラスの INT
コードを編集して、デバッグコードを追加するということに慣れている方もいるのではないでしょうか。
少しやり方が粗いですが、通常ならそれでも構いません。 しかし、この場合はそれだとうまく行きません。
このコードを実行するには、クラスをコンパイルし直す必要があります。
(結局はクラスのコンパイラに呼び出されます。) しかし、クラスをまたコンパイルすると、このルーチンが再生成されるので、加えた変更がすべて消去されてしまいます。 幸い、ZBreak か Studio のデバッガーを使えば、このコードを細かく確認できます。 ルーチン名が分かっているので、ZBreak の使い方はいたって単純です。

```
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>
```

Studio のデバッガーの使い方も簡単です。 *.G1.MAC
ルーチンにブレークポイントを設定し、$System.OBJ.Compile()
をクラスに対して呼び出すようにデバッグターゲットを設定できます。

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

これでデバッグ作業が開始されます。

結論

この記事では、メソッドジェネレータについて簡単にまとめました。
詳細にご興味のある方は、以下のドキュメンテーションをお読みください。

- メソッドジェネレータとトリガージェネレータの定義
- %class オブジェクトと %compiledclass オブジェクトの詳細は、以下をご覧ください。
  - %Dictionary クラスの使用について
  - %Dictionary.ClassDefinition のクラスリファレンス
  - %Dictionary.CompiledClass のクラスリファレンス

#オブジェクトデータモデル #コンパイラ #Caché #InterSystems IRIS

ソースURL:
https://jp.community.intersystems.com/post/cach%C3%A9-%E3%81%AE%E3%83%A1%E3%82%BD%E3%83%8

3%E3%83%89%E3%82%B8%E3%82%A7%E3%83%8D%E3%83%AC%E3%83%BC%E3%82%BF%E3%82%92%E4%BD%BF%E3%81%A3%E3%81%9F%E3%82%B3%E3%83%BC%E3%83%89%E7%94%9F%E6%88%90%E3%81%AE%E6%A4%9C%E8%A8%BC