Usage du dynamisme sous PowerShell

Par Laurent Dardenne, le 18 octobre 2009.

Niveau

PowerShell propose un langage de script dynamique permettant de créer ou de modifier simplement du code ou la structure d'un objet lors de l'exécution d'un script.

Ce tutoriel vous présente les principes de base sur le dynamisme sous PowerShell au travers de quelques exemples.

Merci à shawn12 pour sa relecture et ses corrections.

Testé avec PowerShell V1 sous Windows XP.

Site de l'auteur : http://laurent-dardenne.developpez.com/

Chapitres

1 Les bases du dynamisme sous PowerShell

Le dynamisme d'un langage repose sur ses possibilités, lors de l'exécution, de créer ou de modifier du code ou la structure d'un objet, là où un langage statique doit passer par une phase de compilation.

PowerShell est déjà un shell à typage dynamique ce qui facilite l'écriture du code, il n'est pas nécessaire de déclarer le type d'une variable pour l'utiliser ni d'appeler un constructeur pour l'initialiser.

```
$UneVariable="une chaîne de caractères."
```

Il en est de même lors de la déclaration des paramètres d'une fonction :

```
Function ParametreSansType($UnObjet)
{
 $UnObjet.GetType()
}
$X=9
$Y=10,2
$Z=@(1, "cinq", "C")
$UneVariable,$X,$Y,$Z|Foreach-Object {ParametreSansType $_}
```

Affiche :

```
IsPublic IsSerial  Name            BaseType
--------  --------  ----            --------
True      True      String          System.Object
True      True      Int32           System.ValueType
True      True      Double          System.ValueType
True      True      Object[]        System.Array
```

La fonction *ParametreSansType* peut donc recevoir n'importe type d'objet, mais puisque le contrôle de validité du code se fait à l'exécution, il est de notre ressort de nous assurer de la validité du code contenu dans cette fonction. Ici il ne pose problème, car la méthode *GetType()* existe pour tous les objets.

Vous noterez qu'on ne précise pas un type pour la valeur de retour, PS sait que c'est toujours un objet. Il reste possible de le préciser :

```
[string]$var
#ou
return [string]$var
```

1.1 Les différents conteneurs de code

J'utilise ici le mot *code* en tant que suite d'instructions placée dans un lieu de stockage persistant ou temporaire et pouvant être exécutée. Le mot persistant précise ici l'existence d'un *code* en dehors d'une session PowerShell, le mot temporaire précise une existence liée à la durée de la session PowerShell.

Un *code* peut être persistant en créant un script sur le disque, c'est le point de départ pour tout type de langage dynamique ou pas. Un exemple, les profils utilisateurs de PowerShell.

Un *code* peut être temporaire *via* le provider de fonction ou le provider de variable, ici en tant que scriptblock. Une variable de type string peut contenir du code, mais il sera nécessaire d'utiliser un cmdlet spécifique pour l'exécuter.

Un provider peut également créer du code, des fonctions ou des variables, lors de son initialisation.

L'intérêt d'un langage dynamique est qu'il lui est possible de créer du code transitoire facilement. On peut donc créer du *code ex nihilo* ou modifier un *code* existant.

Par exemple, on peut envisager une fonction créant une fonction, son *code* ne change pas au gré des exécutions, c'est son résultat qui changera. Comme toutes les autres, mais ici ce qu'elle produira sera une donnée exécutable, c'est-à-dire du code source qui peut être du PowerShell ou un langage compilé comme le C#, leVB.NET ou du C. Pour ces langages une étape supplémentaire de compilation sera nécessaire. On peut également envisager une fonction modifiant son propre *code*.

Un script persistant dont le *code* est figé, permet de créer dynamiquement du *code*, c'est donc un programme qui crée un programme. De plus, ce *code* peut évoluer pendant l'exécution de la session et sa durée de vie est liée à celle de la session. Il reste possible de le rendre persistant sur disque soit en créant un script soit à l'aide du cmdlet **Export-Clixml.**

Le dynamisme est donc la faculté qu'a un *code* d'effectuer un traitement sur lui-même, si le système d'introspection permet à un objet de déterminer qui il est, le dynamisme lui permet de changer sa manière d'être selon le contexte ou les besoins. Étant donné la pointe d'anthropomorphisme présente dans cette description, rappelez-vous que l'existence de l'objet en question résulte de votre action, il n'a pas en soi d'autonomie.

1.2 L'expansion de variable

Une autre facilité est l'expansion de variable :

```
"Ceci est $UneVariable"
```

Affiche :

```
Ceci est une chaîne de caractères.
```

On peut penser que l'expansion n'est ni plus ni moins qu'une concaténation, mais ce n'est pas le cas. Le terme d'expansion de variable indique la capacité de remplacer une variable par son contenu sans préciser d'instruction particulière, on parle aussi de substitution de variable.

L'expansion ne fonctionne que si la chaîne de caractères est délimitée par des guillemets doubles, pour l'exemple PowerShell n'effectue aucun traitement sur la chaîne de caractères :

```
'Ceci est $UneVariable'
```

Affiche :

```
Ceci est $UneVariable.
```

Il s'agit ici d'une chaîne littérale, il n'y a aucune interprétation du contenu.

Toutes les variables peuvent être substituées, mais le résultat de l'expansion peut ne pas être exploitable, ainsi autant une variable contenant un tableau l'est :

```
"$Z"
```

autant un objet spécialisé ne l'est pas

```
$cp = new-object Microsoft.CSharp.CSharpCodeProvider
"$cp"
```

Dans ce cas, l'expansion est identique à l'appel de la méthode de conversion *ToString()*, par défaut elle renvoi le nom de la classe de l'objet :

```
  $cp.ToString()
```

On peut aussi vouloir construire une chaîne contenant le nom d'une variable PowerShell mais retarder son expansion :

```
$S= "$X `$Y"
$S
```

Renvoi :

```
9 $Y
```

Le caractère backtick (Alt-Gr 7), apostrophe inverse, permet "d'échapper" le caractère dollar, c'est-à-dire qu'on informe l'interpréteur de le considérer comme un caractère et non plus comme une instruction du langage définissant un nom de variable à traiter. On désactive provisoirement son interprétation.

Ici la variable *$Y* ne peut plus être développée puisque l'expansion se fait à la volée par l'interpréteur de PowerShell :

```
"$S"
```

Renvoi :

```
9 $Y
```

Les concepteurs ont prévus ce cas et propose la méthode *ExpandString()* :

```
$ExecutionContext.InvokeCommand.ExpandString($S)
```

Cet appel évalue bien le contenu de la variable passé en paramètre :

```
9 10.2
```

On peut aussi utiliser l'opérateur + :

```
$S="Ceci est "+$UneVariable ; $S
```

Mais dans ce cas il s'agit d'une concaténation, et le contenu de la variable *$UneVariable* sera toujours ajouté :

```
$S='Ceci est '+$UneVariable ; $S
```

1.2.1 Manipulation de membres d'objets

Cette possibilité d'expansion de chaîne offre l'avantage de manipuler aisément des propriétés et des méthodes d'objets :

http://blogs.msdn.com/powershell/archive/2009/01/05/invoking-methods-using-variables.aspx

1.3 Ajout de membre d'objet

L'ajout dynamique de membres à l'aide du cmdlet **Add-Member** permet de modifier la structure d'un objet PowerShell, reportez-vous au tutoriel suivant qui traite dans le détail cette fonctionnalité :

http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/CreationDeMembresSynthetiquesSousPowerShell/

Consultez notamment le chapitre 4.6 *Déclaration d'un membre d'un objet personnalisé en accès privé.*

Avec ce cmdlet, il n'y a pas de phase de compilation, comme cela est le cas avec C# 3.0 et l'extension de méthode, cela se fait instantanément en mémoire. On ne modifie pas l'objet cible, mais un objet adapté par PowerShell.

1.4 Regroupements

Il existe 3 types de regroupement d'instructions, chacun répondant à des besoins différents.

1.4.1 Le regroupement simple

Il se déclare avec un couple de parenthèses :

```
$s=(dir c:\Autoexec.bat).Name
$s
```

Les parenthèses forcent l'exécution du code qu'elles englobent avant d'exécuter le reste de l'expression. La recherche de la propriété *Name* se fera sur le résultat renvoyé par la sous expression **`(dir c:\Autoexec.bat),`** c'est-à-dire sur un objet fichier.

Les parenthèses sont également utilisées pour préciser l'ordre lors de calculs :

```
1+2*4
9 #1+8
(1+2)*4
12 #3*4
```

Le code ne peut contenir qu'une seule instruction ou un pipeline, pouvant être constitué de plusieurs segments :

```
$s=(dir c:\Autoexec.bat;cls).Name
```

```
« ) » de fermeture manquante dans l'expression.
Au niveau de ligne : 1 Caractère : 24
$s=(gps *|where {$_.Threads.count -gt 15}|select -first 1).Name
```

1.4.2 Le regroupement en tant que sous-expression

Il se déclare avec un couple de parenthèses précédé du caractère dollar. Pour ce type de groupe, le code peut contenir plusieurs instructions ou un pipeline.

```
$(cls;dir c:\Autoexec.bat).Name
```

L'inverse fonctionne également, car la construction précédente renvoie un seul objet :

```
$(dir c:\Autoexec.bat;cls).Name
```

En revanche, les suivantes, renvoyant plusieurs objets, ne fonctionnent pas :

```
(dir c:\*).Name
$(dir c:\*;cls).Name
```

Elle renvoie un objet, mais c'est un tableau d'objets :

```
$(dir c:\*;cls).GetType()
 # On manipule un tableau contenant des fichiers
$(dir c:\*;cls).Count
(dir c:\*).Count
$S=(dir c:\*|Where {$_.Name -match "^autoe"}).Name
```

Si la propriété *Name* existait sur la classe *Array*, l'instruction renverrait un résultat.

Ce type de groupe permet également de manipuler au sein d'une chaîne de caractères une propriété d'un objet *via* un nom de variable :

```
$F=dir c:\Autoexec.bat
"$F.name"
"($F).name"
"($F.name)"
```

```
C:\Autoexec.bat.name
(C:\Autoexec.bat).name
(C:\Autoexec.bat.name)
```

Pour ces derniers exemples l'interpréteur développe la variable *$F* puis ajoute le reste de la chaîne, à savoir **.Name**. Pour éviter cela, on utilisera la construction suivante :

```
"$($F.name)"
```

On peut aussi combiner les types de groupe :

```
"$(cls;(dir c:\Autoexec.bat).name)"
 #ou encore
$F="c:\Autoexec.bat"
"$((dir $F).name)"

$sb={dir c:\Autoexec.bat}
"$((&$sb).name)"
```

Ici le résultat d'un tel regroupement peut être un scalaire ou un tableau.

Sous PowerShell version 2, l'affectation du résultat d'un code peut s'affranchir de l'utilisation du regroupement :

```
 # V1
$result = $(ForEach($i in (1..5)) { $i * 2 })
 #V2
$result = ForEach($i in (1..5)) { $i * 2 }
```

1.4.3 Le regroupement de sous-expressions renvoyant toujours un tableau

Il se déclare avec un couple de parenthèses précédé du caractère arrobas :

```
$o=@(dir c:\Autoexec.bat)
```

```
$o.GetType()
```

```
IsPublic IsSerial Name                 BaseType
-------- -------- ----                 --------
True     True     Object[]             System.Array
```

```
$o=$(dir c:\Autoexec.bat ;Write-Host "Renvoi un scalaire")
$o.GetType()
```

```
IsPublic IsSerial Name                 BaseType
-------- -------- ----                 --------
True     True     FileInfo             System.IO.FileSystemInfo
```

La différence d'avec le regroupement précédent est que celui-ci renverra toujours un tableau, même si le résultat de l'expression renvoie un seul objet.

Il permet de regrouper le résultat de plusieurs instructions :

```
$T=@(dir c:\ ; dir d:\)
```

Pour rappel le séparateur d'éléments lors de la construction d'un tableau est la virgule, le point-virgule est lui le séparateur utilisé lors de la construction d'une hashtable, il n'y a donc pas d'ambiguïté possible.

1.5 Here-String

Pour la manipulation de nombreuses lignes, on peut utiliser une chaine littérale appelée Here-String sous PowerShell (chaîne verbatim en C#). Ce type de chaine peut contenir du texte, des variables et du code, renvoyant ou non un résultat, on combine ainsi plusieurs approches :

```
$i=$true
$Texte=@"
première ligne
   Seconde ligne
`t`t`tTroisième ligne $f $(
  If ($i) {"`r`nQuatrième ligne"}
)
"@
$Texte
```

Le caractère échappé ***`t*** représente une tabulation.

Ici aussi l'usage de guillemets doubles déclenche l'expansion de chaîne.

```
$Texte = @'
première ligne
   Seconde ligne
`t`t`tTroisième ligne $f $(
  If ($i) {"`r`nQuatrième ligne"}
)
'@
$Texte
```

Notez que dans le premier exemple la position du groupe :

```
`t`t`tTroisième ligne $f $( …
```

Evite la création d'une ligne vide si $i est égal à *$false :*

```
`t`t`tTroisième ligne $f
$( … #génère une ligne vide : un espace + un line feed
```

Mais impose, si toutefois on le souhaite, l'ajout d'un retour chariot dans la chaîne :
"`r`nQuatrième ligne"

Les caractères échappés **`r** et **`n** représentent respectivement un retour chariot, ligne suivante.

Dans le premier exemple de chaîne *here-string* contenant du code, son résultat renvoie une chaîne de caractère et pas un ensemble de données et de code. On la construit une seule fois.

Le second exemple est bien un ensemble de données et de code, la chaîne peut être construite en utilisant la méthode *ExpandString($Texte).* Cet ensemble de données et de code ne constitue pas pour autant un objet, cela reste une chaîne de caractères. Ici c'est la méthode *ExpandString* qui prend en considération le code présent dans le texte de la variable *$Texte*.

Attention les combinaisons **@"** ou **@'** doivent être suivi d'un retour chariot sinon une erreur d'analyse se déclenchera :

```
Jeton non reconnu dans le texte source.
```

Note : PowerShell V2 propose l'imbrication de *here-string* :

http://www.nivot.org/2008/12/23/TheTwelveDaysOfPowerShell20CTP3.aspx

1.5.1 REPL

Le code contenu dans une here-string est donc analysé puis exécuté, comme il l'est sous la console. PowerShell étant basé sur boucle d'évaluation ou d'interrogation de type REPL (Read-Eval-Print-Loop) :

(loop (print (eval (read))))

- **Read** lit une expression.
- **Eval** évalue (calcule le résultat de) cette expression.
- **Print** imprime sur la sortie standard le résultat de l'évaluation.
- **Loop** recommence en Read.

D'après fr.wikibooks.org

Sous PowerShell tout est exécuté dans un pipeline, et PowerShell affiche par défaut le résultat du pipeline sur l'écran de la console, la sortie standard dépend du contexte. Par exemple, une affectation ne produit pas d'affichage, bien que l'étape **Print** soit tout de même exécutée.

ExpandString assume les étapes **Eval** et **Print**.

Voir aussi **C# interactive shell** : http://www.mono-project.com/CsharpRepl

2 Fonctions

Dans ce chapitre nous utiliserons les cmdlets de manipulation d'éléments de provider (*Get-Item, Set-Item,* etc.) sur le provider **Function**.

2.1 Création

Le plus souvent les fonctions sont créées à partir d'un script, bien que sous PowerShell on puisse envisager une fonction créant une fonction. L'exemple suivant se borne à créer une fonction d'une seule ligne de code :

```
function New-FunctionTest($Number)
{
  $FunctionName="TestFcnt-$Number"
  $code="Write-host $FunctionName"
  New-Item function:$FunctionName -value $code # –options "Allscope"
}
 #crée 5 fonctions
1..5|% {. New-FunctionTest $_}
```

Par défaut la création d'une fonction se fait dans la portée locale, on doit préciser le point dans l'appel de *New-FunctionTest* pour que les fonctions soient créées dans la portée de l'appelant.

Sinon on les crée dans la portée globale, en précisant le paramètre dynamique *Options* du provider de fonctions, comme indiqué dans l'aide en ligne :

```
help function -Category Provider
 #ou encore par
New-Item function:global:$FunctionName
```

2.1.1 Exemple avancé

Le post suivant propose une fabrique de fonctions qui créé des fonctions créant un objet personnalisé : http://dougfinke.com/blog/index.php/2009/09/12/powershell-function-factory/

Sous PowerShell version 1 remplacez la ligne :

```
param(`$$($parts -join ',$'))
```

par

```
param(`$$([string]::Join(',$',$parts) ))
```

2.1.2 Paramètre dynamique

Les paramètres dynamiques sont des paramètres de cmdlet ajoutés par un provider PowerShell et sont disponibles lorsqu'un cmdlet référence un élément en précisant un provider *via* un nom de PSdrive, exemple ***Function****:Prompt.*

Le paramètre *Options* n'est pas ajouté si on référence un PSdrive sur la base de registre, sa présence provoquera une erreur :

```
New-Item HKLM:$FunctionName -value $code -options "Allscope"
```

```
New-Item : Impossible de trouver un paramètre correspondant au nom « options ».
```

```
Au niveau de ligne : 1 Caractère : 51
```

Ce provider propose par contre le paramètre *Type*:

http://msdn.microsoft.com/fr-fr/library/microsoft.win32.registryvaluekind(VS.80).aspx

Note : La version 2 de PowerShell permettra, sous certaines conditions, d'utiliser des paramètres dynamiques dans le code de fonctions et de scripts.

2.1.3 Fonction constante

Le paramètre dynamique *Options* permet également de déclarer une fonction constante, dès lors son contenu ne pourra plus être modifié :

```
New-Item function:Test -value {Write-host "Test"} -options "constant"

CommandType     Name            Definition
-----------     ----            ----------
Function        Test            Write-host "Test"
New-Item function:Test -value {Write-host "Nouveau Test"} -force
New-Item : Impossible d'écrire dans la fonction Test, car elle est constante ou en lecture seule.
```

2.2 Modification

On peut également envisager une fonction modifiant son propre *code*, ici on ajoute des instructions :

```
Function Auto([string]$Statement,[switch] $Build){
  function modify {
    Write-Host "Ajout des instructions : $Statement"
    $AncienCode=$Function:Auto
     Write-Debug "`t ancien code `r`n$AncienCode"
    $null=new-item function:Auto -value "$AncienCode`r`n$Statement" -force
    $NouveauCode=$Function:Auto
     Write-Debug "`t nouveau code`r`n$NouveauCode"
  }
  Write-Host "Fonction Auto"
  if ($Build) { Modify; return }
}

function Test{
 Write-Host "Premier appel: fonction existante" -fore green
 Auto
 Write-Host "Second appel: modification" -fore green
 Auto  -Statement  'Write-Host "Ajout de code"; Write-host "Fait quelque
chose"' -Build
 Write-Host "Dernier appel: fonction modifiée" -fore green
```

```
 Auto
}
cls
Test
```

Cette fonction se modifie en se recréant dans le provider de fonction. On s'aperçoit qu'une fois en cours d'exécution, ce code est indépendant de celui contenu dans le provider de fonction.

2.2.1 Une possible application

Editez le script *Demo-Add-Remove-Fonction.ps1*

La fonction *Add-FunctionStatements* insère une chaîne dans le code d'une fonction, l'insertion se fait à la ligne 1. La fonction *Remove-FunctionStatements* supprime n lignes dans le code d'une fonction, la suppression se fait à partir de la ligne 1.

Le problème à résoudre est d'insérer les nouvelles lignes après la clause ***Param*** si elle existe. On doit donc effectuer une recherche à l'aide d'une expression régulière reconnaissant les possibles imbrications de parenthèses. Par exemple :

```
Function Add-FunctionStatements(
 [String]$F=$(Throw "error."),
 [String]$S=$( $Objet.Get("Truc",(1+2))),
)
```

Voici l'expression régulière décomposée, la première partie recherche la clause ***Param***, la seconde recherche en mode *SingleLine*, c'est-à-dire que l'on prend en compte les caractères cr+lf lors de la recherche, puisque la clause ***Param*** peut s'étendre sur plusieurs lignes :

```
 $PatternGroupeParameters="(?<Parameters>(?>[^()]+|\((?<DEPTH>)|\)(?<-
DEPTH>))*(?(DEPTH)(?!))\))"
 $PatternParamClause="(?s)^param\($PatternGroupeparameters(.*)$"


   if( $Code -Match $PatternParamClause)
…
```

Les deux fonctions utilisent la même approche, PowerShell version 1 ne disposant pas de module j'ai fait le choix de dupliquer le code. Un exemple d'utilisation ajoutant 2 lignes à la fonction nommé *Auto* puis les supprimant :

```
$Function:Auto
Add-FunctionStatements "Auto" `
    "Write-Debug `"Code debug ajouté pour $FunctionName`"`r`n"
$Function:Auto
Remove-FunctionStatements "Auto" 2
$Function:Auto
```

Ainsi il est possible d'ajouter dynamiquement du code dédié au débogage, sans avoir à modifier le script d'origine.

2.3 Suppression

Si le code exécuté est une copie de celui contenu dans le provider, une fonction peut donc se supprimer elle-même du provider :

```
function TestFcnt-6{
  dir function:TestFcnt-[0-6]| remove-item –Verbose
}
TestFcnt-6
```

Cela reste possible, car sous PowerShell il n'y pas de système de dépendance statique, entre ces objets. Bien évidemment, toute référence par la suite à une fonction inexistante provoquera une erreur.

```
TestFcnt-6
```

```
Le terme « TestFcnt-6 » n'est pas reconnu en tant qu'applet de commande, fonction, programme exécutable ou fichier de script. Vérifiez le terme et réessayez.
```

3 Création de code

3.1 Création dynamique de variable

Les cmdlets ***-Variable** permettent de manipuler des variables de manière dynamique, que ce soit dans sa portée ou dans celles de la chaîne de l'appelant.

Un exemple avec **New-Variable** :

```
Function F1{
 trap { Set-Variable -name I -value 7 -scope 1 ; continue }

 Function F2{
   #créé dans la portée de F1
  New-Variable –name J –value 10 –scope 1 -ea SilentlyContinue
  Function F3{
     #créé dans la portée de F1
   New-Variable K 15 –scope 2 -ea SilentlyContinue
     #créé dans la portée principale,
     #ici elle est à trois niveaux par rapport à celui-ci
   New-Variable Test 20 –scope 3 -ea SilentlyContinue
  }#F3
  F3
 }#F2
 $I=5
 F2
 $I; $J ; $K
 throw "Test"
 Write-Host
 $I; $J ; $K
```

```
}#F1

F1
$Test
Write-Host ("`$I={0} `$J={1} `$K={2}" -F  $I,$J,$K)
```

C'est le paramètre -*Scope* qui définit la portée dans laquelle créer la variable, ainsi la fonction F3 peut créer une variable dans la portée 'primaire', celle de PowerShell.

Pour supprimer des variables on utilisera **Remove-Variable :**

```
 #Création
$Var=@("I","J","K")
$var|
 Foreach { New-Variable -name $_  -value ("$_") -ea SilentlyContinue}
Dir variable:[A-Z]

 #Suppression
$Var|
   #on gère silencieusement les erreurs si la variable n'existe pas
  Get-variable -ea SilentlyContinue|
  Remove-variable -ea SilentlyContinue
```

L'existence d'une variable peut être testée à l'aide du cmdlet **Test-Path** :

```
Test-Path Variable:J
```

Get-Variable permet lui de récupérer des variables d'une portée parente :

```
function Get-ScriptDirectory
{  #Renvoi le nom du répertoire d'un script parent,
   #celui appelé sur la ligne de commande.
    #Auteur Jeffrey Snover
  $Invocation = (Get-Variable MyInvocation -Scope 1).Value
  Split-Path $Invocation.MyCommand.Path
}
```

Vérifions dans les cmdlets liés aux variables celles qui utilisent le paramètre -*Scope* :

```
Get-Command -Noun Variable|
 Where {
 (get-help ($_.Name) -parameter scope -ea SilentlyContinue) -ne $null
 }
```

3.2 Invoke-Expression

Ce cmdlet exécute du code contenu dans une chaîne de caractères :

```
$Commande="5*3"
$Res=Invoke-Expression $Commande
```

```
$T=1..5
"$F.Count"
$F='$T'
"$F.Count"
Invoke-Expression "$F.Count"
```

Pour ces exemples l'usage de fonctions suffit, le principe sera toujours le même, on construit le code puis on l'exécute.

Pour l'exemple suivant, l'objectif est d'ajouter un membre en lecture seule :

```
 #Création de la séquence
$Sequence= new-object System.Management.Automation.PsObject

$Sequence_Name="FileName"
$Comment="Génère un numéro de fichier."
```

Premier essai, on utilise directement **Add-Member** dans le code du script :

```
$Sequence|
 add-member ScriptProperty Name -value "$Sequence_Name" -Pass|
 add-member ScriptProperty Comment -value "$Comment"
```

Le problème est qu'il déclenche l'exception suivante puisque, par mesure de sécurité sous PowerShell version 1, une string ne peut être casté (transformé) en un scriptblock :

```
Add-Member : Impossible de convertir la valeur « FileName » en type
« System.Management.Automation.ScriptBlock ».
Erreur : « Cast non valide de 'System.String' en System.Management.Automation.ScriptBlock. »
```

Second essai, on utilise des scriptblocks :

```
$Sequence= new-object System.Management.Automation.PsObject
$Sequence|
 add-member ScriptProperty Name -value {"$Sequence_Name"} -Pass|
 add-member ScriptProperty Comment -value {"$Comment"}
$Sequence
```

Ici cela fonctionne, avec un léger inconvénient, visible à l'aide de **Get-Member :**

```
$Sequence|gm
```

```
   TypeName: System.Management.Automation.PSCustomObject

Name        Member          Type            Definition
----        ------          ----            ----------
…
Comment     ScriptProperty  System.Object   Comment {get="$Comment";}
Name        ScriptProperty  System.Object   Name    {get="$Sequence_Name";}
```

Le scriptblock contient du texte, du code, il n'est donc pas traité par l'expansion de chaîne. Ce qui fait que chaque modification des variables référencées sera automatiquement répercutée dans les membres les utilisant. Cela pourrait servir à créer des membres d'objets s'adaptant au contexte, mais ce n'est pas notre objectif.

Pour prendre en compte l'expansion de variable puis exécuter le code de création du membre, on utilise une *here-string.* C'est donc une chaîne de caractères contenant du code :

```
$Sequence= new-object System.Management.Automation.PsObject
$MakeReadOnlyMember=@"
`$Sequence|
 add-member ScriptProperty Name -value {"$Sequence_Name"} -Pass|
 add-member ScriptProperty Comment -value {"$Comment"}
"@
```

L'affichage de cette variable renvoie le code suivant :

```
$Sequence|
 add-member ScriptProperty Name -value {"FileName"} -Pass|
 add-member ScriptProperty Comment -value {"Génère un numéro de fichier."}
```

Il correspond bien au résultat attendu, reste à l'exécuter

```
Invoke-Expression $MakeReadOnlyMember
$Sequence
```

Ici nous avons créé du code, $*MakeReadOnlyMember*, qui à son tour appelle du code d'ajout dynamique de membre *via* le cmdlet **Add-Member.** Le cmdlet **Invoke-Expression** quant à lui évalue le code dans la portée courante.

Si on supprime la variable séquence, l'appel génère une exception :

```
Remove-Variable Sequence
Invoke-Expression $MakeReadOnlyMember
```

```
Add-Member : Impossible de lier l'argument au paramètre « InputObject », car il a la valeur Null.
Au niveau de ligne : 2 Caractère : 12
+  add-member  <<<< ScriptProperty Name -value {"FileName"} -Pass|
```

Attention au risque d'injection de code en couplant le cmdlet Invoke-Expression avec le résultat d'une saisie utilisateur.

http://blogs.msdn.com/powershell/archive/2006/11/23/protecting-against-malicious-code-injection.aspx

Note : La méthode suivante propose la même fonctionnalité

```
$executioncontext.InvokeCommand.InvokeScript('5*3')
```

3.3 Scriptblock

Un scriptblock est une classe spécifique au framework PowerShell, il contient un bloc de code comme en contient une fonction ou un script. En regardant ceux-ci de plus près on s'aperçoit qu'il utilise chacun un scriptblock ou bloc de script.

La différence réside dans le nommage, chaque *code* présent dans un conteneur possède obligatoirement un nom :

- un script est créé dans le provider ***FileSystem*** (persistant), on peut supprimer un script, mais ici, s'il n'existe plus il ne nous intéresse pas.

- une fonction est créée dans le provider ***Function*** (temporaire),
- une variable est créée dans le provider ***Variable*** (temporaire), une variable peut contenir une instance d'un scriptblock.

Mais un scriptblock peut aussi être anonyme (une lambda expression d'après Bruce Payette). vous en utilisez très souvent, par exemple avec les cmdlets **Where-Object** et **Foreach-Object** :

```
1..5| Foreach-Object {$_* 2}
```

Ou avec l'instruction **Trap :**

```
trap { Set-Variable -name I -value 7 -scope 1 ; continue }
```

Dans ce cas il n'est pas nécessaire de créer une fonction ou une variable, la seule présence d'accolade ouvrante et fermante suffit pour l'utiliser :

```
{Dir C:\}.gettype()
```

```
IsPublic   IsSerial   Name            BaseType
--------   --------   ----            --------
True       False      ScriptBlock     System.Object
```

```
{Dir C:\}
```

```
Dir C:\
```

La présence de l'opérateur **&** (caractère esperluette, aussi appelé *et commercial*) exécute le code contenu dans le scriptblock :

```
&{Dir C:\}
```

```
   Répertoire : Microsoft.PowerShell.Core\FileSystem::C:\
Mode    LastWriteTime        Length    Name
----    -------------        ------    ----
d----   09/10/2009   16:48    <DIR>   WINDOWS
…
```

L'usage du dotsourcing sur un scriptblock l'exécute dans la portée courante, celle de l'appelant, alors que l'usage du **&** l'exécute dans une nouvelle portée.

Cela permet par exemple de gérer une exception localement alors que l'exécution des instructions se fait dans la portée de l'appelant :

```
trap { Write-Error $Error[0].Exception.Message ; continue} #globale
.{
   #Le point(dotsource) charge $PckScripts dans la portée de l'appelant
   # et pas dans celle de ce scripblock.
  trap { Finalyze ;  Throw $_ }  #Locale
   { Throw $_.Exception.Message}
  .$PckScripts
}
```

On peut visualiser le code d'une fonction de la manière suivante :

```
(get-Item function:Prompt).ScriptBlock
(get-Item function:Prompt).Definition
```

Les deux propriétés affichent le même contenu, mais l'une renvoi un objet de la classe scriptblock l'autre un objet de la classe String. Toutes les deux sont en lecture seule, on ne peut donc pas faire ceci :

```
(get-Item function:Prompt).ScriptBlock={Dir C:\}
```

Un scriptblock peut être utilisé comme argument d'un paramètre et éviter des variables locales

```
$Sq=New-Sequence -maxvalue {[int]::MaxValue+1}
```

L'usage d'un regroupement est possible, mais il est plutôt dédié aux chaînes de caractères :

```
$Sq=New-Sequence -maxvalue $([int]::MaxValue+1)
```

Un exemple par Jeffrey Snover :

```
dir *.ps1 |copy-Item -Destination {if ($_.length -ge 100) {$_.Name +
".BIG"} else {$_.Name + ".SMALL"}} -whatif
```

3.3.1 Création de scriptblock

La classe **System.Management.Automation.Scritpblock,** raccourci ***[scriptblock]***, ne contenant pas de constructeur il n'est pas possible d'en créer directement à partir d'une chaine de caractère, comme nous l'avons vu dans le chapitre sur le cmdlet **Invoke-Expression**. Encore une fois les concepteurs ont prévu une méthode renvoyant un objet scriptblock à partir d'une chaîne de caractère, la méthode *NewScriptBlock()*.

Prenons comme objectif de valider un nom de variable reçu en paramètre d'une fonction, ce nom de variable est utilisé par la suite dans du code dynamique.

Certaines constructions de nom ne sont pas autorisées, par exemple la présence d'opérateur, dans ce cas au lieu de tester toutes les combinaisons possibles, le plus simple est d'évaluer la validité du nom en exécutant un scriptblock utilisant ce nom de variable :

```
$VariableName="Tes-T"
```

La variable *$VariableName* contient le nom de la variable reçue en paramètre, il contient un opérateur ceci provoquera une erreur :

```
$Tes-T="erreur"
```

```
Vous devez fournir une expression de valeur à droite de l'opérateur « - ».
```

Avant d'aller plus loin dans le traitement, on doit s'assurer que le nom reçu est correct.

Pour ce faire, on utilise une expression utilisant de nombreuses fonctionnalités de PowerShell.

```
&$ExecutionContext.InvokeCommand.NewScriptBlock("`$$VariableName=0")
```

Ou :

- *"`$$VariableName=0"* est une chaîne de création du nom de variable **$Tes-T=0**,
- *NewScriptBlock* créé un scriptblock **{$Tes-T=0}** et
- *&* exécute le scriptblock dans une nouvelle portée.

Reste à gérer les exceptions :

```
 #Stop le pipeline
trap {break }
 &{  #Gére l'erreur du scriptblock de validation du nom de variable
    trap { Throw "Le nom de variable ( $VariableName ) pose problème." }
    &$ExecutionContext.InvokeCommand.NewScriptBlock("`$$VariableName=0")
  }
```

```
…
```

3.3.2 Usage d'un Scriptblock avec le pipeline

Sur le sujet, consultez le chapitre 5 du tutoriel suivant :

http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/Pipelining/

3.3.3 Vérification de la syntaxe

Un scriptblock peut également servir à vérifier en partie la syntaxe d'un code généré, consultez l'article suivant *Preparsing Scripts to Check for Syntax Errors* :

http://keithhill.spaces.live.com/blog/cns!5A8D2641E0963A97!6036.entry

3.3.4 Remaniement de l'exemple avancé de création de fonction

Dans le chapitre de création de fonction, l'exemple avancé a pour moi un petit défaut qui est qu'on ne peut déclarer de fonctions dans une portée locale. Ceci permettrait d'en déclarer plusieurs de même nom, mais offrant un comportement différent, et leurs suppressions seraient automatisés.

Modifions-la pour lui permettre de créer une fonction soit locale soit globale. Le principal problème est sa déclaration dans une portée locale. L'utilisation du mot clé ***Global***, lors de la création du code de la fonction, ne pose pas de problème, de plus en fin d'exécution les variables locales utilisées par la fonction sont automatiquement libérées, ce ne sera pas le cas pour la création de la fonction dans une portée locale.

Comme nous l'avons vu, une fonction peut être exécutée dans la portée locale en utilisant le dotsourcing, première chose à régler, comment déterminer si une fonction est exécutée avec le dotsourcing ?

La variable automatique *$MyInvocation* peut nous aider à le savoir, si sa propriété *InvocationName* contient un caractère point cela indique que la fonction a été exécutée en dotsourcing :

```
   if (MyInvocation.InvocationName -eq '.')
    { $Scope=[String]::Empty }
   else
    { $Scope="Global:" }
```

Ainsi, on peut adapter le type de portée lors de la création du code de la fonction :

```
@"
Function $Scope$name {
```

Une fois ceci fait, et puisqu'on exécute la fonction *New-Function* dans la portée locale afin d'y créer notre fonction, on génère un effet de bord. En utilisant le dotsourcing toutes les variables utilisées dans *New-Function* sont créées dans la portée de l'appelant, on risque donc d'écraser des variables de même nom, de plus leurs suppressions seraient à notre charge.

Pour régler ce problème on doit créer une nouvelle portée, l'usage d'un scriptblock répond à ce besoin :

```
$sbNewFunction={
```

```
   if ((Get-Variable MyInvocation -Scope 1).Value.InvocationName -eq '.')
    { $Scope=[String]::Empty }
   else
    { $Scope="Global:" }
…
```

Mais son usage force à interroger la variable automatique *$MyInvocation* de l'appelant, sinon on interrogerait la variable automatique *$MyInvocation* de la portée du scriptblock. Celui-ci renvoie le code de la fonction à créer dans le pipeline.

Le code de création se résume à appeler **Invoke-Expression** sur ce résultat :

```
Invoke-Expression (&$sbNewFunction)
```

De cette manière, il est possible d'adapter le comportement du script selon le type d'appel :

```
 #Création de New-Personne dans la portée globale
New-Function 'New-Personne' 'Nom Prénom Adresse Ville CodePostal'

Function Test {
   #Création de New-Personne dans la portée locale de TEST
 . New-Function 'New-Personne' 'Nom Prénom Adresse Ville CodePostal'
 …
}
```

Pour l'appel en dotsourcing, il reste un dernier problème, les paramètres nommés de la fonction sont déclarés dans la portée locale, on risque donc d'écraser une variable existante portant le même nom. Pour éviter cela on utilisera directement le tableau d'arguments *$Args* pour récupérer les paramètres désormais anonymes :

```
$sbNewFunction={
  $Name=((gv Args -scope 1).value)[0]
     #Validation du paramètre
    Test-Variable (gv Name) String -strict -TestEmptyString
  $Properties=((gv Args -scope 1).value)[1]
    Test-Variable (gv Properties) String -strict –TestEmptyString
…
```

Vous trouverez la fonction *Test-Variable* dans le projet Add-Lib (PackageDebugTools.ps1).

3.4 Variable liée (tied variable)

Sur son blog Lee Holmes aborde le principe des variables liées qui est similaire à celui d'un scriptblock, la différence étant qu'un scriptblock doit être explicitement appelé pour retourner un résultat, celui-ci devant être affecté à une variable avant de pouvoir être manipulé :

```
$MaVariable=&$MonScriptBlock
```

Une variable liée associe un nom de variable au résultat retourné par un scriptblock, l'accès à une telle variable déclenche automatiquement l'appel au scriptblock associé.

```
 #Crée une variable liée de portée global
```

```
New-ScriptVariable GLOBAL:today { Get-Date -uformat "%A" }
```

```
$Today
Samedi  #Renvoie le nom du jour courant
```

Si on affiche le contenu de l'objet variable *today*, on découvre que son contenu n'est pas un scriptblock mais bien le résultat de son exécution :

```
$v=Get-variable Today
$v
```

```
Name                 Value
----                 -----
GLOBAL:today         {Samedi}
```

Selon le code contenu dans le scriptblock, la valeur changera à chaque appel.

3.4.1 Hébergement d'un objet

L'article de Lee Holmes proposant cette fonction référence un post du blog de l'équipe PowerShell : http://blogs.msdn.com/powershell/archive/2009/03/26/tied-variables-in-powershell.aspx

Celui-ci propose une approche au cas par cas, de cette manière il est possible d'héberger un objet privé. Avec la fonction *New-ScriptVariable* cela n'est pas possible, car pour y accéder on utiliserait un nom de propriété sur le même nom de variable, cet accès déclencherait un appel récursif provoquant une exception :

```
New-ScriptVariable GLOBAL:Random -Get { $Random.Obj.Next(1,10) }
```

```
Échec du script en raison d'un dépassement de la profondeur des appels. La profondeur des appels a atteint 103 alors que le maximum est 100.
```

Par contre, on peut utiliser un autre objet global ou local, voici un exemple :

```
[PSObject] $_Random=New-Object System.Random
$_Random|
 Add-Member Scriptproperty Min -Value {[Int]5} -Secondvalue `
          {Throw "La propriété Min est en lecture seule."} -PassThru |
 Add-Member Scriptproperty Max -Value {[Int]15} -Secondvalue `
         {Throw "La propriété Max est en lecture seule."}

New-ScriptVariable GLOBAL:Random -Get {
                               $_Random.Next($_Random.Min,$_Random.Max) }
```

Si on supprime la variable référencée dans le scriptblock, l'accès à la variable liée provoquera une exception :

```
rv _random
$random
```

```
Vous ne pouvez pas appeler de méthode sur une expression ayant la valeur Null.
Au niveau de ligne : 1 Caractère : 54
+ New-ScriptVariable GLOBAL:Random -Get { $_Random.Next( <<<< $_Random.Min,$_Random.Max) }
```

Si on recrée un variable de même nom, mais d'un contenu différent, par exemple les valeurs de *Min* et *Max*, l'accès à la variable liée fonctionne de nouveau.

3.5 Création de code C#

La création dynamique de code dotnet, C# ou autre, suit le même principe, on créé un code source puis on l'exécute. Elle requiert toutefois une étape supplémentaire: la compilation du code source, suivi implicitement de son chargement dans le domaine d'application de PowerShell.

Là on est à la frontière entre PowerShell et le framework dotnet, le mécanisme appelé ***CodeDOM*** jouera le rôle du passeur. Il crée du code dans un assembly que soit dans une dll sur disque ou en mémoire, mais dans tous les cas une compilation intermédiaire se fera sur disque. Il va sans dire que des connaissances de base du langage cible sont nécessaires.

Nous prendrons comme exemple le code de génération d'une structure C# :

http://projets.developpez.com/wiki/add-lib/Function_New-Struct

Pour retrouver les blocs aisément, visualisons-le dans un éditeur de code :

```
        $code = @"
using System;
 $(
  $Structs.Keys|
   Foreach {
   $Name=$_;
   $Properties=$Structs.$_;
   "`n  public struct $Name {`r`n"
   $(
      $Properties.Keys |
      Foreach {
        "  public {0} {1};`n" -f $Properties[$_],($_.ToUpper()[0] + $_.SubString(1))
      }
    )
   "`n   public $Name (" + $(
    [String]::join(',',($Properties.Keys | % { "{0} {1}" -f $Properties[$_],($_.ToLower()) }))
    ) + ") {`r`n"
   $($Properties.Keys | Foreach {
      "    {0} = {1};`n" -f ($_.ToUpper()[0] + $_.SubString(1)),($_.ToLower())
     }
    )
     "`n  }`n }"
   }
)
"@
```

Ce qui nous donne pour cet appel :

```
New-Struct @{Album=@{Numero=[Int32];Titre=[String]}}
```

Le code C# suivant

```
using System;

 public struct Album {
  public System.Int32 Numero;
  public System.String Titre;

   public Album (System.Int32 numero,System.String titre) {
     Numero = numero;
     Titre = titre;
  }
 }
```

Le respect du formatage aura son importance lors des phases de mise au point du code source généré.

On voit que ce code utilise toutes les possibilités du langage PowerShell, on retrouve dans une here-string un regroupement **$(…)** contenant des pipelines, d'autres regroupements, des concaténations et des formatages de chaînes. Je vous laisse le soin de l'analyser, vous avez, je pense, tous les éléments pour le faire.

Une fois le code généré on passe à la phase compilation et de chargement en mémoire :

```
  $provider = New-Object Microsoft.CSharp.CSharpCodeProvider
  $dllName = [PsObject].Assembly.Location

# Configure the compiler parameters
  $compilerParameters = New-Object `
System.CodeDom.Compiler.CompilerParameters

  $assemblies = @("System.dll", $dllName)
  $compilerParameters.ReferencedAssemblies.AddRange($assemblies)
  $compilerParameters.IncludeDebugInformation = $true
  $compilerParameters.GenerateInMemory = $true

  $compilerResults = $provider.CompileAssemblyFromSource(
$compilerParameters, $code)
  if($compilerResults.Errors.Count -gt 0) {
    $compilerResults.Errors | % { Write-Error ("{0} :`t {1}" -F
$_.Line,$_.ErrorText) }
  }
```

Voir aussi :

Utilisation de CodeDOM en C# : http://vincentlaine.developpez.com/tuto/dotnet/codedom/

3.5.1 Création de code MSIL

Il existe une autre possibilité, qui est l'émission de code IL à l'aide des classes de l'espace de nom System.Reflection.Emit :

« *L'espace de noms System.Reflection.Emit contient des classes qui permettent à un compilateur ou à un outil d'émettre des métadonnées et le langage MSIL (Microsoft Intermediate Language) et de générer éventuellement un fichier exécutable portable (PE) sur le disque. Les clients principaux de ces classes sont les compilateurs et les moteurs de script.* »

Certains scripts, voir le projet ***add-lib***, tel que celui de création d'un délégué ou d'une énumération en font usage, mais leur étude dépasse de loin le cadre de ce tutoriel.

La version 2 de PowerShell offrira d'autres possibilités notamment celle d'accéder au parseur de PowerShell et autres méta données.

Il sera par exemple possible de compiler puis d'exécuter du code sur une machine distante :

http://blogs.msdn.com/powershell/archive/2009/07/22/dynamic-binary-modules.aspx

Dans le post suivant Jeffrey Snover propose du code avancé au sujet de la méta programmation :

http://blogs.msdn.com/powershell/archive/2009/01/04/extending-and-or-modifing-commands-with-proxies.aspx

People have no idea of the power and flexibility that they are going to get with PowerShell V2. It is going to be awesome!, Jeffrey Snover

4 Liens

Vous pouvez consulter le projet ***add-lib*** qui contient quelques scripts utilisant le dynamisme. La recherche d'occurrences telles que *NewScriptBlock, ExpandString* ou *Invoke-Expression* vous aidera à les retrouver : http://projets.developpez.com/wiki/add-lib

Construction dynamique d'un GUI :

http://blogs.vmware.com/vipowershell/2007/10/powershell-guis.html

Substitution de chaine personnalisée, *Powershell Template Engine* :

http://blogs.msdn.com/mwilbur/archive/2007/03/14/powershell-template-engine.aspx

Calling a Webservice from PowerShell :

http://www.leeholmes.com/blog/CallingAWebserviceFromPowerShell.aspx

PowerShell Class Definition Library :

http://cid-bb10621fcfe1bcf8.spaces.live.com/blog/cns!BB10621FCFE1BCF8!123.entry

PowerShell Deep Dive: Discovering dynamic parameters :

http://poshoholic.com/2007/11/28/powershell-deep-dive-discovering-dynamic-parameters/

Meta-Programming with PowerShell and Regular Expressions :

http://www.tellingmachine.com/post/Meta-Programming-with-PowerShell-and-Regular-Expressions.aspx

Powershell version 2

Génération de code PowerShell à partir d'un assembly .net,

Generating New-Object Wrapper Functions for an Assembly

http://keithhill.spaces.live.com/Blog/cns!5A8D2641E0963A97!6958.entry

A Module to Create Modules and Advanced Functions

http://blogs.msdn.com/powershell/archive/2009/01/02/a-module-to-create-modules-and-advanced-functions.aspx

Analyse des jetons (token) d'un code PowerShell :

Parsing PowerShell scripts

http://blogs.microsoft.co.il/blogs/scriptfanatic/archive/2009/09/07/parsing-powershell-scripts.aspx

Le cmdlet Add-Type facilite l'usage de code à la volée,

Inline F# in PowerShell

http://get-powershell.com/2008/12/30/inline-f-in-powershell/

Custom Accelerators for PowerShell 2, http://poshcode.org/1398

À propos du mot-clé Dynamic du C# 4.0

Le langage C# 4.0 introduit le mot-clé dynamic, permettant un typage dynamique :

http://dougfinke.com/blog/index.php/2009/03/07/powershell-and-the-dynamic-keyword-in-c-40/

Ce mot clé retarde les contrôles normalement effectués à la compilation jusqu'au moment où le code est exécuté. Dans l'exemple cité, le type de la valeur de retour de la fonction *GetVariable* sera toujours celui de la variable *variableName*.

5 Conclusion

Ces différentes possibilités sont simples et puissantes à la fois, elles nécessitent toutefois d'adopter une autre manière d'appréhender le scripting pour en tirer avantage.

Les langages compilés permettent ce type de codage, bien que sous PowerShell cela soit beaucoup plus facilite d'accès. Un exemple, le projet **Sepi** écrit en Delphi Win32 : http://sjrd.developpez.com/sepi/faq/?page=infosgenerales#sepicestquoi