La notion de portée sous Powershell

Par Laurent Dardenne, le 07/09/2015.

Niveau

Débutant Avancé Confirmé

Conçu avec Powershell v4 Windows 7 64 bits.

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

Chapitres

1 Son rôle

Pour aborder la notion de portée, ou scope, voyons d'abord rapidement ce que son absence impliquerait. Pour rappel une variable est avant tout une adresse mémoire contenant une valeur. Supposons que chaque nom de variable soit unique dans le code source, l'écriture de traitements intermédiaires nécessitant la manipulation d'une copie d'une variable s'avérerait difficile.

Il nous faudrait créer une seconde variable et lui affecter la valeur de la première :

```
$Nom='Test'
$NouveauNom=$Nom
```

Nous avons bien 2 variables contenant la même valeur, pour les différencier nous avons utilisé un autre identifiant et chaque variable à une adresse différente.

Powershell permet de décomposer le code au travers de fonctions ou de scripts, ce découpage en 'sous-programme' implique la notion de paramètre :

```
$Nom='Test'
Function Set-Majuscule( $Nom ) {
 $Nom=$Nom.ToUpper()
 Return $Nom
}
Set-Majuscule $Nom
```

Si chaque nom de variable était unique ce code modifierait la variable nommée *$Nom* déclarée en début de script, ce serait un effet secondaire indésirable (un effet de bord). L'exécution de la fonction ne devrait pas modifier le contenu initial de la variable *$Nom*.

On pourrait utiliser un autre nom de variable :

```
$Nom='Test'
Function Set-Majuscule( $NouveauNom ) {
 $NouveauNom=$Nom.ToUpper()
 Return $NouveauNom
}
Set-Majuscule $Nom
```

Cela résout le problème, enfin sous réserve de ne plus utiliser le nom de variable *$NouveauNom*. On ne résout rien, on retarde juste l'échéance. Un autre point serait à considérer : l'occupation mémoire que cela engendrerait, bien évidemment on pourrait supprimer au fur et à mesure les variables devenues inutiles.

Comme ces contrôles seraient à notre charge, l'écriture de code fiable resterait un exercice difficile, tout du moins nécessiterait une constante rigueur.

La notion de portée évite que toutes les variables soient globales c'est-à-dire accessible partout dans un programme, les modifications de variable pourront se faire aux grés des besoins de manière locale ou globale. Cette notion de portée résout également le problème de la durée de vie d'une variable. Ce qui reste à notre charge est le choix du type de portée.

2 Contexte

Lorsqu'on décompose le code en fonctions ou scripts, Powershell crée, pour chaque partie de cette décomposition, un bloc de code (une suite d'instructions), ce qui implique la création d'un nouveau contexte de définition notamment pour les variables. L'usage du terme 'bloc' tombe à pic puisqu'en Powershell une fonction ou un script est conçu autour d'un scriptblock.

Lors de l'ouverture d'une console Powershell on utilise par défaut le contexte global, le parent de tous les autres contextes. Lorsqu'on déclare une variable son nom est ajouté dans la table de symbole du bloc courant :

```
$X=10
```

Cette table mémorise les noms et valeurs des variables déclarées.

La notion de portée ne prend forme qu'à partir du moment où on utilise un sous-programme, par exemple une fonction ou un script. Si nous n'utilisons pas de sous-programme, nous savons où se trouve nos variables : à portée de main.

Lors de l'exécution du code d'un sous-programme Powershell crée automatiquement un nouveau bloc et par là même une nouvelle table de symbole. En interne celle-ci est placée sur une pile et devient la portée courante.

C'est ce qui permet de nommer des paramètres avec le nom d'une variable existante. L'exemple du chapitre précèdent est donc valide en Powershell :

```
$Nom='Test'

Function Set-Majuscule( $Nom ) {
 $Nom=$Nom.ToUpper()
 Return $Nom
}
```

Il n'y a donc pas d'effet de bord. La *variable* $Nom et le *paramètre* $Nom de la fonction ***Set-Majuscule*** sont deux variables distinctes portant le même nom, même si elles ne sont pas déclarées au même 'endroit'. Un même nom, mais deux adresses mémoires différentes.

```
Set-Majuscule $Nom
$Nom
TEST
Test
```

Notez que les variables peuvent avoir un nom identique mais un type différent :

```
[Int] $V=10

Function MonTraitement ( [String] $V ) {
 Return $V.ToUpper()
}
```

De quel type est la variable **$V** ? Ça dépend du contexte…

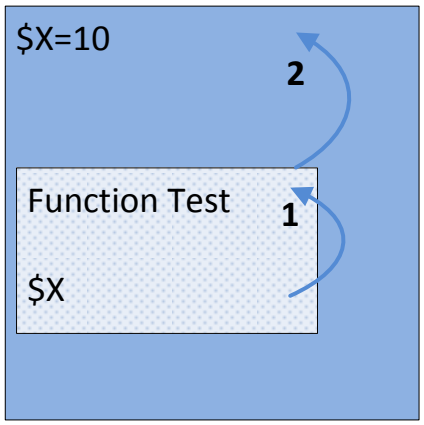
3 Principe

Voyons maintenant le cas où le code d'une fonction référence une variable nommée *$x* qui n'est pas créée dans le code de la fonction et qui n'est pas un nom de paramètre :

```
$x=10
Function Test {
 Write-host "`$X vaut = $X"
}
Test
$x vaut 10
$x
10
```

Ce code affiche bien la valeur de $x, voyons comment Powershell procède.

La fonction *Test* ne contenant pas de code de création de variable nommée **$x**, Powershell recherche en premier dans la table de symbole de la fonction (1), c'est-à-dire dans la portée courante. Ce mécanisme de recherche constate l'absence d'une entrée nommée **$x,** il recherche donc dans la table de symbole du parent, c'est-à-dire dans la portée parente (2). Powershell y trouvant la déclaration de la variable **$x**, la recherche s'arrête.



Imbrication de portée

On peut l'exprimer plus simplement :

> *La variable **$x** n'existe pas dans la portée de la fonction, mais dans la portée parente.*

En cas de multiples niveaux d'imbrication de portée, la recherche parcourt toutes les portées tant que la variable est introuvable, elle s'arrêtera une fois la portée globale atteint.

Vérifions un autre scénario où la variable référencée n'existe pas :

```
Remove-Variable x
Function Test { Write-host "`$X vaut = $X" }
Test
$X vaut =
```

La recherche de la variable **$x** dans les deux portées, celle de la fonction et celle globale, échoue.

Par défaut Powershell est assez souple sur ce comportement afin de ne pas bloquer l'exécution du code, il autorise l'usage de variable inexistante.

En revanche si on bascule, par l'intermédiaire du cmdlet **Set-StrictMode,** dans un comportement strict :

```
Set-StrictMode -Version Latest
Function Test { Write-host "`$X vaut = $X" }
Test
Impossible d'extraire la variable « $X », car elle n'a pas été définie.
```

Powershell déclenche une erreur nous informant que la variable référencée n'a pas été définie, c'est-à-dire que la recherche de la variable **$x** dans la portée de la fonction et dans celle du parent a échouée.

On annule le mode strict ainsi :

```
Set-StrictMode –Off
```

La configuration de ce comportement concerne la portée courante et ses portées enfant.

4 Création d'une portée

Modifions l'exemple précédent en créant une variable dans la portée de la fonction *Test* :

```
$x=10
Function Test {
 $x=99
 Write-host "`$X vaut = $X"
}
Test
$x vaut 99
Write-host "`$X vaut = $X"
$x vaut 10
```

On constate que Powershell référence la valeur de la variable **$x** déclarée dans la fonction et non plus celle déclarée dans la portée globale. Ici aussi il existe deux variables de même nom, mais pas au même endroit. Le mécanisme de recherche trouve maintenant une entrée nommée **$x** dans la table de symbole de la portée courante, du coup la recherche s'arrête.

On peut l'exprimer plus simplement :

*La variable **$x** existe dans la portée de la fonction.*

4.1 Durée de vie

Une fois terminée l'exécution du code de la fonction *Test*, sa variable **$x**, celle valant 99, n'existe plus. Sa durée de vie est égale au temps d'exécution de la fonction.

Si lors de l'exécution d'un sous-programme Powershell crée une nouvelle table de symbole et l'empile, lors du retour du sous-programme il doit la dépiler afin de restituer le contexte de son parent :

```
$x=10
Function Test {
 Write-host "Création d'une nouvelle portée"
  Write-host "  Avant la création `$X vaut = $X"
  $X=99
  Write-host "  Après la création `$X vaut = $X"
 Write-host " On quitte la portée courante"
}
Test
Write-host " Dans la portée parent `$X vaut = $X"
Création d'une nouvelle portée
 Avant la création $X vaut = 10
 Après la création $X vaut = 99
on quitte la portée courante
Dans la portée parent $X vaut = 10
```

Ainsi on retrouve la variable $x de la portée courante, celle-ci n'est pas écrasée mais inaccessible pendant l'exécution d'un sous-programme, ici une fonction.

4.2 Portée dynamique

Powershell n'utilise pas la portée lexicale c'est-à-dire que le mécanisme de recherche débute à l'endroit où on a défini le code (par exemple avec les langages statiques compilés), il utilise la portée dynamique, le mécanisme de recherche débute à l'endroit où le code est appelé :

```
$x=10
Function Afficher { Write-host "Afficher : `$X vaut = $X" }
Function Ajouter {
 $X= 20
 Afficher
}
Afficher
Ajouter
Write-host "Dans la portée globale `$X vaut = $X"
Afficher : $X vaut = 10
Afficher : $X vaut = 20
Dans la portée globale $X vaut = 10
```

Bien que le code soit figé ainsi que l'ordre de déclaration, on constate lors des deux appels de la fonction *Afficher* que le mécanisme de recherche référence le contexte en cours au moment de son appel.

5 Accéder à une portée

Maintenant on peut se poser la question : *Comment référencer une variable d'une portée parent* ?

Reprenons le premier exemple du chapitre précédent :

```
$x=10
Function Test {
 $x=99
 Write-host "`$x vaut = $x"
}
Test
$x vaut 99
```

La solution est d'ajouter une indication dans le code afin d'influencer le mécanisme de recherche, rappelez-vous le choix restant à notre charge. Cette indication est un modificateur de portée à ajouter devant le nom de variable :

$*Nom_de_ modificateur****:****Nom_de_variable***

Les deux informations sont séparées par le caractère deux points ':'.

Pour référencer la variable de la portée globale, qui est ici également la portée parent, il suffit d'ajouter le nom de portée ***Global***:

```
$x=10
Function Test {
 $x=99
 Write-host "`$x vaut = $global:x"
}
Test
$x vaut 10
```

Ce nom de portée n'est pas un opérateur, mais une partie optionnelle du nom de chemin d'une variable. Notez qu'un modificateur de portée concerne la lecture ou l'écriture d'une variable :

```
$x=10
Function Test {
 $global:x=99
 Write-host "`$x vaut = $x"
}
Test
$x vaut 99
```

Le code de la fonction 'Test' ne crée pas de nouvelle variable dans sa portée, il modifie la variable **$x** globale.

5.1 VariablePath

La classe VariablePath, d'un usage avancé, nous confirme que l'on manipule bien un chemin de variable :

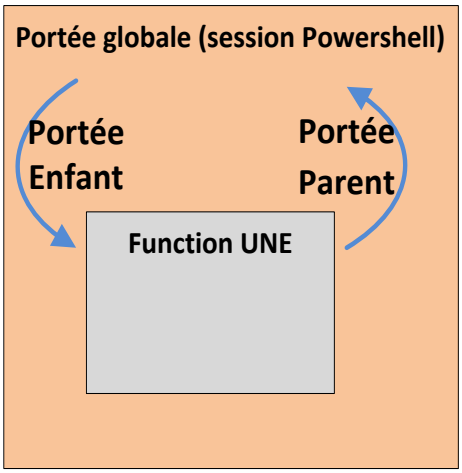
```
$Var=new-object System.Management.Automation.VariablePath 'Global:x'
$Var

UserPath           : Global:x
IsGlobal           : True          !!!
IsLocal            : False
IsPrivate          : False
IsScript           : False
IsUnqualified      : False
IsUnscopedVariable : False
IsVariable         : True          !!!
IsDriveQualified   : False
DriveName          :
$Var.UserPath -split ':'
Global
x
```

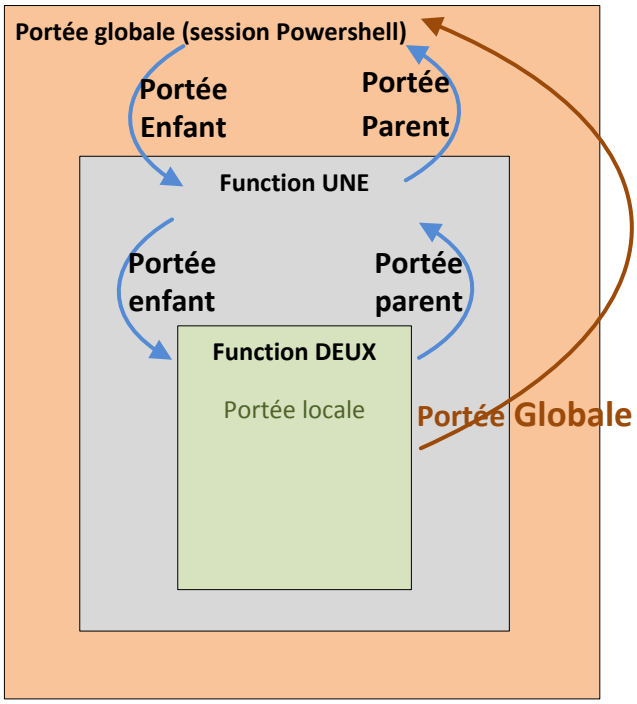
5.2 Les différentes portées

Notez que la notion de portée concerne les éléments (*item*) de type variable, function, alias et PSdrive.

La portée globale est créée au démarrage d'une session Powershell. À la création d'une nouvelle portée et depuis celle-ci, la portée globale devient sa portée parente. Depuis la portée globale, la nouvelle portée devient sa portée enfant :



Ce type de relation se répète tant que l'on imbrique des créations de portée :



La première portée, celle créée au démarrage de la session Powershell, est accessible de n'importe quelle portée via le modificateur *Global:*. Chaque fois que l'on se déplace dans une portée, celle-ci devient la portée locale.

5.2.1 Private

Les variables de portée privées (*private:*) sont inaccessibles en dehors de la portée actuelle. Une portée enfant ne peut accéder aux variables privés des portées parent.

Dans l'exemple suivant :

```
$private:x=10
Function Test {
 Write-host "`$X vaut = $X"
 Write-host "`$X vaut = $global:X"
}
Test
$X vaut =
$X vaut =
```

Bien que la variable **$x** soit créée dans la portée globale, elle l'est en tant que variable privée.

Le code de la fonction 'Test' ne peut plus y accéder. Comme le montre l'exemple suivant, le mécanisme de recherche de variable est impacté.

Avant d'exécuter le code suivant il est nécessaire de supprimer la précédente déclaration de la variable **$x** :

```
Remove-Variable -Name x
$x=-1
function Une{
 $private:x=10
 Function Deux{
   Write-host "`$X vaut = $X"
 }
 Enfant
}
Parent
$X vaut = -1
```

Du fait que la variable **$x** déclarée dans la fonction 'Une' le soit en tant que variable privée, le mécanisme de recherche ne la trouvera pas dans cette portée, il continue donc sa recherche dans les portées parent.

5.2.2 Local

La portée *Local* est la portée par défaut, une variable sans modificateur est créée ou référencée dans la portée *Local*. Sont considérées comme des variables locales celles qui ne précisent pas de modificateur et celles qui précisent le modificateur *Local* ou *Private.*

5.2.3 Global

C'est la portée crée au démarrage de Windows PowerShell. Une variable globale est accessible partout dans un programme. La création de variable globale est à éviter autant que faire se peut.

5.2.4 Portées numérotées

Abordons maintenant le cas où l'on souhaite accéder à une portée particulière :

```
$x=-1
function GrandParent{
   $x=0
   function Parent{
     $x=10
     Function Enfant{
       $X=99
       Write-host "`$X vaut = $global:X"
     }
     Enfant
   }
   Parent
}
GrandParent
$X vaut = -1
```

Ici nous avons quatre niveaux de portée suivant l'ordre de déclaration : la globale, celle de la fonction '*GrandParent'*, celle de la fonction '*Parent'* et celle de la fonction '*Enfant'*.

Dans le code de la fonction *Enfant*, le modificateur *Global* référence la portée globale qui contient toutes les autres portées. Comme il n'existe pas de modificateur nommé 'Parent' ou 'GrandParent', on doit utiliser le cmdlet **Get-Variable** pour référencer la variable déclarée dans la portée de la fonction '*Parent*'. C'est-à-dire accéder à une portée relative par rapport à la portée courante.

La portée 0 représente la portée actuelle, ou locale. La portée 1 indique la portée parente immédiate. La portée 2 indique le parent de la portée parente, et ainsi de suite.

Modifions le code de l'exemple précédent :

```
$x=-1
function GrandParent{
  $x=0
  function Parent{
   $x=10
   Function Enfant {
     $X=99
     Write-host "Locale : `$X vaut = $((Get-Variable -name X -Scope 0).Value)"
     Write-host "Parent : `$X vaut = $((Get-Variable -name X -Scope 1).Value)"
     Write-host "GrandParent : `$X vaut = $((Get-Variable -name X -Scope 2).Value)"
     Write-host "Global : `$X vaut = $global:x" #Ici Global = Scope n° 3
   }
   Enfant
  }
  Parent
}
GrandParent
```

Son exécution affiche le contenu de la variable **$x** déclarée dans chaque portée :

```
Locale : $X vaut = 99
Parent : $X vaut = 10
GrandParent : $X vaut = 0
Global : $X vaut = -1
```

Cette notion de portée relative peut être combinée avec le cmdlet **New-Variable,** qui dispose également du paramètre *–Scope*, ce qui lui permet de définir des variables dans une de ses portées parent.

A partir de la version 3 il est possible de retrouver la liste des commandes utilisant ce paramètre :

```
Get-command -ParameterName Scope
```

Notez que dans l'exemple précédent les numéros de portée sont figés, rappelez-vous que Powershell utilise des portées dynamiques :

```
$x=-1
function GrandParent{
  $x=0
  Parent
}

function Parent{
 $x=10
 Enfant
}
```

```
Function Enfant {
 $X=99
 Write-host "Locale : `$X vaut = $((Get-Variable -name X -Scope 0).Value)"
 Write-host "Parent : `$X vaut = $((Get-Variable -name X -Scope 1).Value)"
 Write-host "GrandParent : `$X vaut = $((Get-Variable -name X -Scope 2).Value)"
 Write-host "Global : `$X vaut = $global:x" #Ici Global = Scope n° 3
}
```

Remarquez que l'ordre de déclaration des fonctions ne déclenche pas d'erreur de parsing puisque la recherche des noms de fonction se fait lors de l'exécution :

```
GrandParent
Locale : $X vaut = 99
Parent : $X vaut = 10
GrandParent : $X vaut = 0
Global : $X vaut = -1
Parent
Locale : $X vaut = 99
Parent : $X vaut = 10
GrandParent : $X vaut = -1
Global : $X vaut = -1
Enfant
Locale : $X vaut = 99
Parent : $X vaut = -1
Get-Variable : Le numéro de portée « 2 » dépasse le nombre de portées actives.
GrandParent : $X vaut =
Global : $X vaut = -1
```

Ici les numéros de portées sont relatifs au contexte d'exécution, pour la fonction 'GrandParent' les résultats sont identique, pour la fonction 'Parent' les valeurs affichés sont décalées, ce qui et normal, et pour la fonction 'Enfant' la portée numéro 2 n'existe pas.

On manipule rarement les portées de cette manière et si on doit le faire l'étude des différents scénarios d'appel est nécessaire.

5.2.1 Script

Il reste une portée nommée *script:* qui comme son nom l'indique concerne en premier lieu les scripts. L'exécution d'un script crée également une nouvelle portée, dans l'exemple suivant :

```
@'
  $x=10
  Function Test {
    $x=99
    Write-host "`$global:X vaut = $global:x"
     Write-host "Parent : `$X vaut = $((Get-Variable -name X -Scope 1).Value)"
    Write-host "`$X vaut = $x"
  }
  Test
'@ > C:\temp\Test.ps1

$x=-1
C:\temp\Test.ps1
$global:X vaut = -1
Parent : $X vaut = 10
$X vaut = 99
```

L'usage du modificateur de portée *global* : référence toujours la portée créée au démarrage d'une session Powershell. Nous avons vu que l'accès à une variable par un numéro de portée dépend du contexte d'appel, ici si on souhaite adresser la variable **$x** créée en début de script indépendamment de l'endroit où l'on se trouve dans le code du script, on utilisera la portée *script:*.

Un exemple :

```
@'
  $x=10
  Function Test {
    $x=99
    Write-host "`$global:X vaut = $global:x"
    Write-host "`$script:X vaut = $script:x"
     Write-host "Parent : `$X vaut = $((Get-Variable -name X -Scope 1).Value)"
    Write-host "`$X vaut = $x"
  }
 Test
'@ > C:\temp\Test.ps1
C:\temp\Test.ps1
$global:X vaut = -1
$script:X vaut = 10
Parent : $X vaut = 10
$X vaut = 99
```

La portée script s'étend sur l'ensemble du script, dans l'exemple suivant on utilise un scriptblock afin d'exécuter la fonction 'Test' dans une nouvelle portée :

```
@'
  $x=10
  Function Test {
    $x=99
    Write-host "`$global:X vaut = $global:x"
    Write-host "`$script:X vaut = $script:x"
     Write-host "Parent : `$X vaut = $((Get-Variable -name X -Scope 1).Value)"
    Write-host "`$X vaut = $x"
  }
  # Scriptblock
 &{
    $x=123; Write-host 'Imbrication'; Test
  }
'@ > C:\temp\Test.ps1
C:\temp\Test.ps1
Imbrication
$global:X vaut = -1
$script:X vaut = 10
Parent : $X vaut = 123
$X vaut = 99
```

On voit que la notion de parent dépend du contexte et la portée *script:*, comme la portée *global:* référence toujours la même portée.

Si on supprime la création de **$x** du début de script :

```
@'
  Function Test {
    $x=99
    Write-host "`$X vaut = $global:x"
    Write-host "`$script:X vaut = $script:x"
    Write-host "`$X vaut = $x"
  }
  Test
'@ > C:\temp\Test.ps1

$x=-1
C:\temp\Test.ps1
$X vaut = -1
$script:X vaut =
$X vaut = 99
```

La variable *$script:x* n'existe plus, Powershell ne référence ni la portée *global* ni la portée *local*.

Un dernier exemple où un script appel un autre script :

```
@'
  Write-Host 'Script Test2.ps1'
  $x=21
  Function Test {
   Write-host "`t`$script:X vaut = $script:x"
  }
  Test
  Write-host "`tParent : `$X vaut = $((Get-Variable -name X -Scope 1).Value)"
'@ > C:\temp\Test2.ps1
@'
  Write-Host 'Script Test.ps1'
  $x=1
  Function Test {
   Write-host "`t`$script:X vaut = $script:x"
  }
  Test
  &C:\temp\Test2.ps1
'@ > C:\temp\Test.ps1

$x=-1
C:\temp\Test.ps1
```

```
Script Test.ps1
        $script:X vaut = 1
Script Test2.ps1
        $script:X vaut = 21
        Parent : $X vaut = 1
```

Chaque usage de la portée *script:* référence le script courant et le parent du script *Test2.ps1* et bien celui qui l'appelle, c'est-à-dire le script *Test.ps1*.

6 Opérateurs

Nous venons de voir que l'on peut manipuler des portées, sachez qu'il est également possible d'influencer l'exécution d'un code à l'aide de deux opérateurs.

Reprenons un des premiers exemples :

```
$x=10
Function Test {
 $X=99
 Write-host "`$X vaut = $X"
}
```

6.1 L'opérateur d'appel &

Cet opérateur (*call operator*) exécute du code dans une nouvelle portée, c'est le comportement par défaut. Vous pouvez vérifier le comportement de l'opérateur '**&**' :

```
&Test
$x vaut 99
Write-host "`$X vaut = $X"
$x vaut 10
$Cmd = 'Get-Variable'
&$Cmd E*
```

6.2 L'opérateur . ou dotsourcing

L'opérateur point '.' exécute du code dans la portée courante. A la différence d'une manipulation de portée qui se fait dans le code appelé, avec cet opérateur c'est l'appelant qui manipule la portée, ce comportement est appelé **le dotsourcing**.

Modifions le type d'appel de la fonction 'Test' en utilisant le dotsourcing :

```
. Test
$x vaut 99
Write-host "`$X vaut = $X"
$x vaut 99
```

L'affectation de la variable **$x** s'effectue dans la portée courante, cet appel équivaut au code suivant :

```
$x=10
$X=99
```

Nous n'utilisons pas de modificateur de portée sur la variable **$x**, on déclare une seule fonction et selon l'usage on utilisera l'un ou l'autre de ces opérateurs.

Utilisez le dotsourcing si vous souhaitez inclure des déclarations de variables ou de fonctions à partir de fichier .ps1, par exemple dans un fichier profile de Powershell.

Dans l'exemple suivant on configure un traitement à partir d'un script dédié :

```
@'
$Server='SrvDC001'
$AccountName='MonCompte'
Function Test {
 Write-Host "Traitement"
}
'@ > C:\Temp\Initialize-Traitement.ps1
#Configure la portée courante à partir d'un script
. C:\Temp\Initialize-Traitement.ps1
"$Server\$AccountName"
SrvDC001\MonCompte
```

Par convention, nommez votre script en utilisant le nom de verbe Initialize adéquat.

6.2.1 Portée Script

Si le code d'un script référence la portée *script:* et qu'on l'exécute en dotsource, la portée *script:* est égale à la portée *global:*

```
@'
  $x=10
  Function Test {
    $x=99
    Write-host "`$global:X vaut = $global:x"
    Write-host "`$script:X vaut = $script:x"
    Write-host "`$X vaut = $x"
    $script:x=123
  }
 Test
'@ > C:\temp\Test.ps1
$x=-1
C:\temp\Test.ps1
$global:X vaut = -1
$script:X vaut = 10
$X vaut = 99
```

Si on exécute ce même script en dotsource :

```
. C:\temp\Test.ps1
$global:X vaut = 10
$script:X vaut = 10
$X vaut = 99
$x
123
```

On constate que :

- ✓ l'affectation de la valeur 10 à la variable **$x** se fait sur la variable **$x** de la portée courante,
- ✓ dans la fonction 'Test' l'affectation de la valeur 123 à la variable **$script:x** se fait également sur la variable **$x** de la portée courante.

Attention, le dotsourcing peut affecter les modificateurs de portée du code appelé.

7 Options de variable

Sous Powershell il est possible de créer des constantes à l'aide du cmdlet ***New-Variable*** :

```
New-Variable X -Option Constant -Value 10
Function Test {
 $X=99
 Write-host "`$X vaut = $X"
}
Test
$X vaut = 99
$x
10
```

Bien qu'on ne puisse pas modifier le contenu de la variable **$x** dans la portée où on l'a déclarée :

```
$x=-1
Impossible de remplacer la variable X, car elle est constante ou en
lecture seule.
```

Il reste possible de la modifier dans une portée enfant, car dans ce cas Powershell crée une nouvelle variable.

Si on souhaite que cette variable **$x** soit considérée comme constante dans chaque nouvelle portée on doit lors de sa création préciser l'option ***AllScope***.

L'exemple suivant nécessite de fermer la session Powershell courante puis d'en créer une nouvelle, car une variable constante globale ne peut être supprimée :

```
#Nouvelle session
New-Variable X -Option Constant,AllScope -Value 10 -Force
Function Test {
 $X=99 #Dans la fonction Test
 Write-host "`$X vaut = $X"
}
Test
Impossible de remplacer la variable X, car elle est constante ou en
lecture seule.
Au caractère Ligne:2 : 2
+  $X=99 #dans la fonction Test
$X vaut = 10
```

L'option ***AllScope*** copie la variable, avec ses options, dans chaque nouvelle portée. Cette option s'applique à la portée courante, mais pas à la portée globale :

```
 # !!! Ici aussi crée une nouvelle session !!!
$x=-1
&{
  New-Variable X -Option Constant,AllScope -Value 10 -Force
  Function Test {
   $X=99
   Write-host "`$X vaut = $X"
  }
  Test
}
Impossible de remplacer la variable X, car elle est constante ou en
lecture seule.
$X vaut = 10
$x
-1
$x=0
$x
0
```

Attention, l'usage de l'option ***AllScope*** référence la même variable, c'est-à-dire son adresse :

```
$x='Début'
$TestScriptBlock={
  Remove-Variable X -Force -EA SilentlyContinue
  New-Variable X -Option AllScope -Value -1
  function GrandParent{
    $x=0
     Write-host "Parent : `$X vaut = $((Get-Variable -name X -Scope 1).Value)"
    Write-host "Global : `$X vaut = $global:x"

    function Parent{
     $x=1
      Write-host "Parent : `$X vaut = $((Get-Variable -name X -Scope 1).Value)"
     Write-host "Global : `$X vaut = $global:x"

     Function Enfant {
       $X=2
      Write-host "Parent : `$X vaut = $((Get-Variable -name X -Scope 1).Value)"
      Write-host "GrandParent : `$X vaut = $((Get-Variable -name X -Scope
2).Value)"
      Write-host "Arrière grandParent  : `$X vaut = $((Get-Variable -name X -Scope
3).Value)"
       Write-host "Global : `$X vaut = $global:x"
     }
     Enfant
    }
    Parent
  }
  GrandParent
}

#Exécuté dans sa propre portée
&$TestScriptBlock
```

```
Parent : $X vaut = 0
Global : $X vaut = Début
Parent : $X vaut = 1
Global : $X vaut = Début
Parent : $X vaut = 2
GrandParent : $X vaut = 2
Arrière grandParent  : $X vaut = 2
Global : $X vaut = Début
```

La variable **$x** globale n'est pas impactée par les modifications, mais celle du code de test est modifiée dans toutes les portées.

Si on exécute ce code dans la portée courante :

```
 #Exécuté dans la portée courante
. $TestScriptBlock
```

```
Parent : $X vaut = 0
Global : $X vaut = 0
Parent : $X vaut = 1
Global : $X vaut = 1
Parent : $X vaut = 2
GrandParent : $X vaut = 2
Arrière grandParent  : $X vaut = 2
Global : $X vaut = 2
```

La variable **$x** globale est bien modifiée, bien que les variables **$x** des différentes portées ne précisent pas le modificateur *Global:*.

7.1 Autre Items

Pour un PSDrive le cmdlet **New-PSDrive** propose le paramètre *–Scope* :

```
New-PsDrive -Scope Script -Name Prj -PSProvider FileSystem -Root $SvnTrunk
```

Un objet PSDrive ne posséde pas de propriété nommée 'option'.

Pour un alias le cmdlet **New-Alias** propose le paramètre *–Scope* :

```
New-Alias list get-childitem -Scope Local
```

Le cmdlet **Set-Alias** propose les paramètres *–Scope* et *–Option.*

Pour une fonction il est possible de préciser un modificateur de portée lors de sa création :

```
function global:Test ()
{
   $x
}
```

C'est la fonction qui est accessible globalement, le code suit toujours les règles de portée.

Pour modifier les options d'une fonction on doit utiliser le cmdlet **New-Item** :

```
Set-Item –Path function:Test -Option readonly
Get-Item –Path function:Test|fl *
```

Il faut toutefois préciser le nom du provider ***Function*** dans le nom du chemin.

Note : Il n'existe pas de possibilité de modifier une fonction d'une portée parent, l'usage d'une variable contenant un scriptblock, qui est une fonction anonyme, peut être une solution…

8 Conclusion

La connaissance de cette notion de portée est essentielle sous Powershell, elle est simple à comprendre, sous réserve de prendre le temps de l'étudier. Le dernier exemple combinant plusieurs comportements n'est pas d'un usage courant, il présente les possibilités d'écriture à l'aide des portées ainsi que les pièges que l'on peut rencontrer.

Sachez qu'il existe d'autres comportements liés aux scopes, par exemple pour les modules. Une nouvelle version peut en proposer de nouveau, par exemple la version 3 ajoute le modificateur **Workflow:** qui référence la portée d'un workflow.