La portée d'un scriptblock dans un module

Par Laurent Dardenne, le 28/10/2016.

Niveau

Débutant	Avancé	Confirmé

Conçu avec Powershell v5 Windows 7 64 bits.

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

Chapitres

Dans un précédent tutoriel j'avais abordé la notion de portée, dans celui-ci je vais aborder la portée liée à un objet de type scriptblock et les problèmes que cela peut poser lorsque l'on déclare des paramètres de ce type dans une fonction d'un module.

1 Rappel

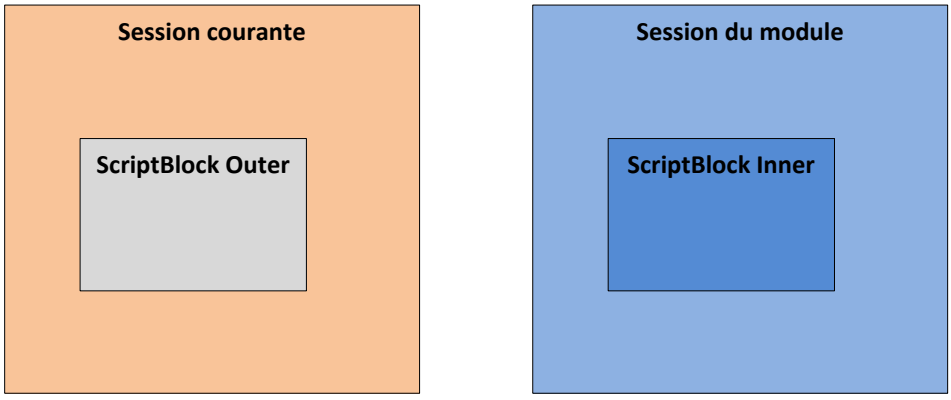
Techniquement, un runspace est un environnement d'exécution dans lequel fonctionne Windows PowerShell. Chaque runspace inclut une instance du moteur de *System.Management.Automation* qui définit une session de Windows PowerShell.

Chaque runspace possède un contexte d'exécution qui lui est propre, par exemple les variables existantes dans un runspace **A** ne sont pas visibles d'un runspace **B** et inversement.

On ne peut exécuter d'instructions PowerShell qu'au travers d'un pipeline, et lui-même ne peut fonctionner que dans un runspace. Un runspace peut contenir plusieurs objets pipeline, mais on ne peut exécuter qu'un seul pipeline à la fois. La fonctionnalité des Job permet d'exécuter plusieurs pipelines mais chacun dans un runspace qui lui est propre.

Lors de l'ouverture d'une console Powershell on dispose automatiquement d'une session et lorsque l'on importe un module, Powershell crée une autre session dédié au module. C'est ce qui permet de déclarer des variables privées au module sans avoir à préciser une portée 'private:' (qui n'existe pas).

Nous avons donc schématiquement ceci :



Le zones indiquées '*Scriptbloc Outer'* et '*Scriptblock Inner'* référence des variables contenant un scriptblock.

Nous avons dit que chaque runspace possède un contexte d'exécution qui lui est propre, par exemple les variables existantes dans la session courante ne sont pas visibles dans la session du module et inversement. Pour qu'une variable d'un module soit accessible depuis l'extérieur on doit la préciser dans un appel à **Export-ModuleMember** –Variable *Name* et pour qu'une variable de la session courante soit accessible on la passe en paramètre à une fonction du module, on passe son contenu.

2 Exemple

Lors du portage d'une fonction en un module, j'ai rencontré un problème de portée embarrassant.

J'ai un script contenant la déclaration d'une fonction :

```
#Edit-String.sp1
Function Edit-String {}
```

Celui-ci est exécuté en dot source dans la portée courante (la console Powershell):

```
. .\Edit-String.ps1
```

Cette fonction utilise la méthode *Replace* de la classe ***[Regex].*** Cette méthode ci plus particulièrement :

```
 string Replace(string input, MatchEvaluator evaluator)
```

Le paramètre nommé 'evaluator' contient un bloc de code ou scriptblock qui est identique à ce que l'on utilise avec le cmdlet **Where-Object** *{ filtre }* ou **Foreach-Object** *{enumération}* :

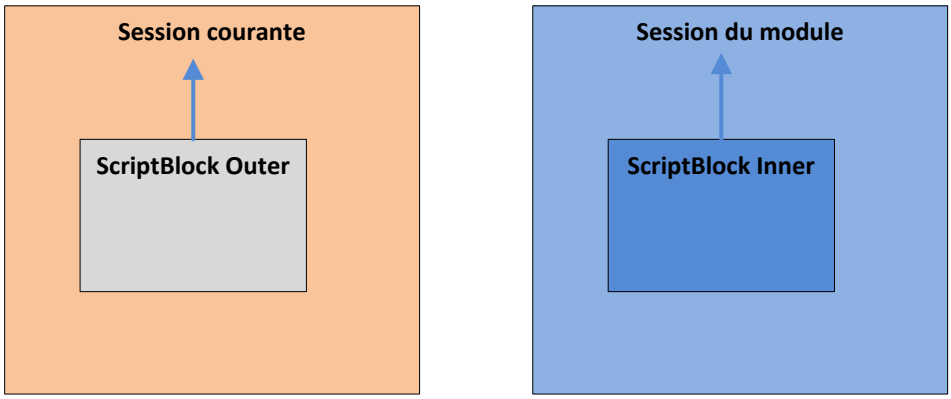
```
Function Edit-String {
  param(
   [string]$pattern,
   [string] $Text,
   [scriptblock] $evaluator
  )
  $Regex=New-Object System.Text.RegularExpressions.RegEx $pattern
  $Regex.Replace($Text, $evaluator)
}
```

Ainsi on paramètre une fonction avec du code.

Note :

La méthode *Replace* ayant été conçue avant Powershell, un mécanisme interne lui permet d'appeler du code Powershell, l'association de l'état de session courante et du scriptblock en fait partie.

Par défaut le code d'un scriptblock est exécuté dans la portée courante, celle où on le déclare :



Ce qui fait que lors de son exécution il référence les variables et fonctions déclarées au même 'endroit' :

```
$maVar='Portée courante.'
$evaluator= {'.';Write-Warning "maVar=$maVar"}
 #& est un opérateur d'invocation
&$evaluator
```

```
.
WARNING: maVar=Portée courante
```

Avec une fonction exécutée en dot source, le scriptblock et les variables déclarées dans le code sont créées le même état de session :

```
  #Remplace les espaces par des points
$Params=@{
 Pattern='\s'
 Text="Dans l'espace personne ne vous entend crier."
 Evaluator=$evaluator
}
Edit-String @Params
```

```
WARNING: maVar=Portée courante.    #Est affiché 6 fois
Dans.l'espace.personne.ne.vous.entend.crier.
```

L'objectif est atteint.Le scriptblock *$evaluator* référence notre variable *$maVar*.

Mais une fois cette fonction replacée dans un module, le fonctionnement change, car un scriptblock contient en interne une référence à l'endroit (état de session) où il est déclaré, si on utilise ce module qui redéclare les variables *$maVar* et *$evaluator* (de même contenu) :

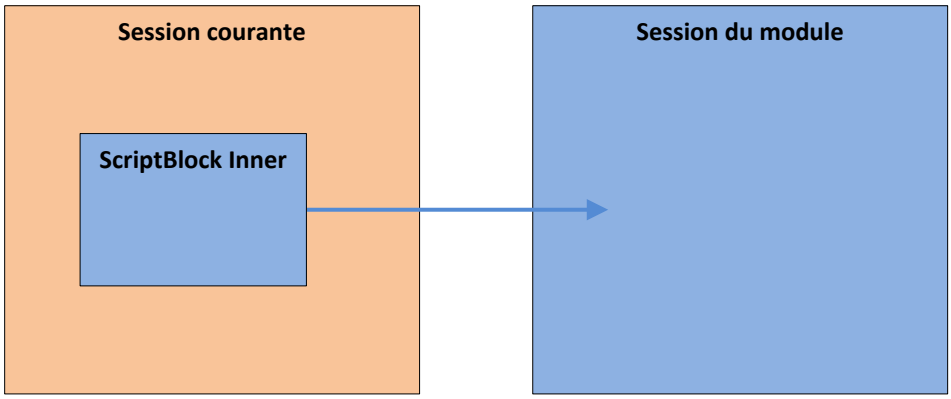
```
$maVar='Portée du module.'
$script:evaluator={'.';Write-Warning "maVar=$maVar"}

Function Edit-String {
  param(
   [string]$pattern,
   [string] $Text,
   [scriptblock] $evaluator
  )
  $Regex=New-Object System.Text.RegularExpressions.RegEx $pattern
  $Regex.Replace($Text, $script:evaluator)
}
Export-ModuleMember -variable evaluator -function Edit-String
```

Le résultat est différent :

```
Ipmo .\Test.psm1 -force
Edit-String '\s' "Dans l'espace personne ne vous entend crier." $evaluator
WARNING: maVar=Portée du module.     #Est affiché 6 fois
Dans.l'espace.personne.ne.vous.entend.crier.
```

L'export de la variable *$evaluator* permet son accéess dans la portée courante, mais l'objet qu'elle contient, le scriptblock, est déclaré dans la portée du module, ce qui fait qu'il référence l'état de session du module :



Le scriptblock *$evaluator* référence la variable *$maVar* déclarée dans le module.

Ceci est correct.

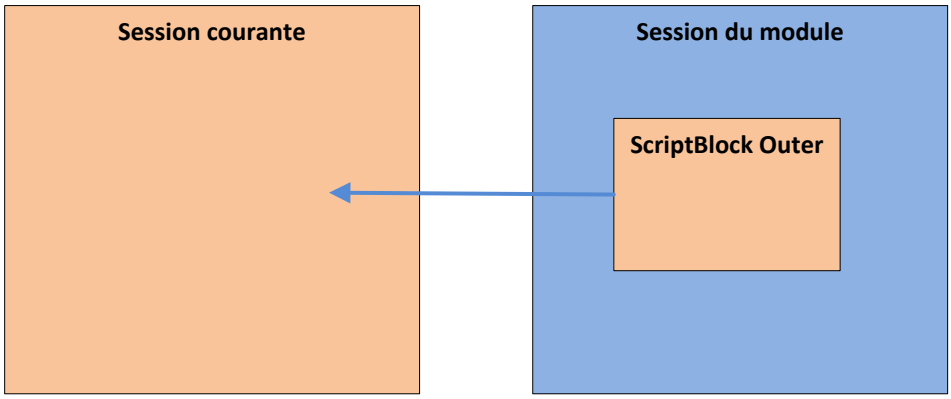
Redéclarons notre variable dans la portée courante :

```
$evaluator= {'.';Write-Warning "maVar=$maVar"}
```

Et exécutons à nouveau la fonction du module :

```
Edit-String '\s' "Dans l'espace personne ne vous entend crier." $evaluator
WARNING: maVar=Portée courante.    #Est affiché 6 fois
Dans.l'espace.personne.ne.vous.entend.crier.
```

On obtient le même résultat que précédemment, pourtant le code du scriptblock est exécuté dans la portée du module, on peut s'attendre à ce qu'il référence la portée où il est exécutée. Eh Non !



De plus le scriptblock déclaré dans le module est désormais inaccesssible, il nous faut soit réimporter le module pour recréer la variable, soit ajouter dans le module une fonction qui la déclare :

```
Function Get-DefaultEvaluator {
  {'.';Write-Warning "maVar=$maVar"}
}
$script:evaluator= Get-DefaultEvaluator
Export-ModuleMember -variable evaluator`
                    -function Edit-String, Get-DefaultEvaluator
```

Dans mon cas la variable *$evaluator* est une configuration par défaut qui est exportée.

J'aimerais la déclarer dans la portée du module mais que le code du scriptblock qu'elle contient référence la portée de l'appelant.

Le problème qui se pose est comment faire ?

La solution n'est pas de changer le problème :-)

3 Une solution basée API

Il existe l'API *InvokeWithContext* qui se rapproche du comportement souhaité.

Le premier paramètre de cette méthode est une hashtable contenant les déclarations des fonctions contenues dans le scriptblock : *@{ NomDeFonction={Code} }*, on peut lui passer une hashtable vide.

Le second paramètre est une liste générique de PSVariable. Voir le module 'Test2.psm1'.

Lors de l'exécution de la fonction, Powershell recherche dans l'état de session de l'appelant les variables précisées, sous réserve que celles recherchées n'existent pas dans le module :

```
$Regex.Replace(
  $Text,
  $script:evaluator.InvokeWithContext( @{}, $ListVar)
)
```

C'est une solution mais elle est inappropriée pour notre cas.

L'appel de la méthode *InvokeWithContext* se fait **avant** l'appel de la méthode *Replace* et renvoie une collection contenant un seul objet, un point '.'. Ce qui fait que l'algorithme de recherche de signature de méthode de Powershell désigne comme candidate la méthode *Replace(string input, string replacement).*

Et surtout c'est la méthode *Regex.Replace* qui devrait appeler en interne cette API et ce plusieurs fois. Un coup dans l'eau !

Note : Un exemple approprié pour cette méthode, un autre en C#.

4 Une solution basée comportements

Cette solution combine des comportements et fonctionnalités de Powershell.

Première étape, utiliser la clé *ScriptsToProcess* d'un manifeste de module :

```
@{
 RootModule = 'Test.psm1'
 ModuleVersion = '0.1.0'
 GUID = '8c5b4061-f1c0-48b5-95ae-4af0dd93a4ce'
 ScriptsToProcess = @('Init.ps1')
 }
```

Le ou les scripts indiqués dans cette clé sont exécutés en dot source dans la portée de l'appelant du module et avant le chargement du module :

```
function New-SBOuter {
  {'.';Write-Warning "maVar=$maVar"}
}
```

Le script *Init.ps1* déclare une fonction dans la portée où l'on importe le module. Son rôle est de créer et renvoyer un objet scriptblock.

Puisque que la fonction *New-SBOuter* crée un scriptblock dans la portée souhaitée, il nous reste à l'affecter à notre variable interne :

```
$script:evaluator=New-SBOuter
```

Puis à supprimer la fonction :

```
Remove-item function:New-SBOuter
```

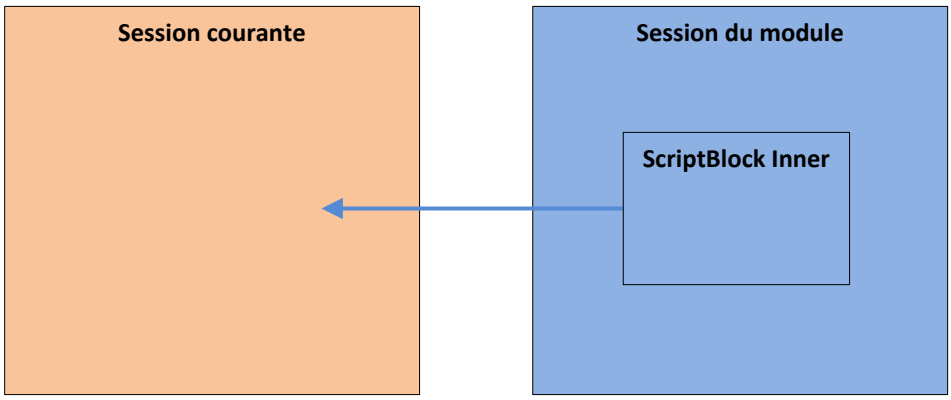
Sachez que l'objet scriptblock créé n'est pas lié au code de la fonction mais à l'état de session où il est déclaré. On manipule une référence de scriptblock

```
$maVar='Portée courante.'
ipmo .\Test3.psd1 -force
WARNING: Utilise le scripblock du module
WARNING: maVar=Portée du module.
WARNING: Affecte un scriptblock créée dans la portée de l'appelant.
WARNING: Utilise le Scripblock du module
WARNING: maVar=Portée courante.
```

Reste à vérifier l'appel dans la session courante :

```
Edit-String '\s' "Dans le module personne ne vous entend crier."
$evaluator
WARNING: maVar=Portée courante.
Dans.le.module.personne.ne.vous.entend.crier.
```

Nous avons bien une variable déclarée dans la portée du module et le code du scriptblock qu'elle contient référence la portée de l'appelant :



Un problème de réglé ☺

5 Un peu de réflexion

Maintenant que l'on sait résoudre ce cas, on peut également vouloir que le module crée une variable dans la portée de l'appelant.

Nous avons vu au début de ce document que chaque runspace possède un contexte d'exécution qui lui est propre, Le module doit donc connaitre l'état de session de l'appelant, on peut lui passer en paramètre la variable automatique ***$ExecutionContext.SessionState***, la bien nommée.

On souhaite faciliter l'usage et masquer l'implémentation, on réutilise l'approche précédente en lui ajoutant un mécanisme basé sur le système de reflection de dotnet.

Il s'agit de manipuler une propriété interne d'un objet scriptblock, cette approche comporte un risque de breaking change, ce qui est normal. On retrouve cette technique dans le module Pester ou dans la version originale du module LINQ for Powershell, celui-ci étant utilisé par l'équipe de développement de Powershell, je me suis dit qu'elle n'allait pas 'casser' ce qu'elle utilise. On justifie une prise de risque comme on peu ☺

Le module 'Test4.psm1' contient deux fonctions *Get-ScriptBlockScope* et *Set-ScriptBlockScope*.

La première récupère l'état de session d'un scriptblock :

```
$p=[scriptblock].GetProperty('SessionStateInternal',$flags)
$p.GetValue($ScriptBlock,$null)
```

La seconde modifie la propriété du scriptblock contenant l'état de session :

```
$p=[scriptblock].GetProperty('SessionStateInternal',$flags)
$p.SetValue($ScriptBlock, $SessionStateInternal, $null)
```

Pour créer la variable *$FromModule* on utilise le cmdlet **New-Variable** au sein d'un scritpblock déclaré dans le module :

```
$SbPrivate={
 Write-Warning "Crée la variable FromModule dans la portée de l'appelant"
 New-Variable -Name 'FromModule' -Value 'Créée par un module' -scope 1
}
```

Avant tout, on récupère l'état de session du scriptblock externe :

```
$SbOuter=New-SbOuter
$SsCaller=Get-ScriptBlockScope $SbOuter
```

Puis on l'affecte au scriptblock déclaré dans le module :

```
Set-ScriptBlockScope -ScriptBlock $SbPrivate –SessionStateInternal $SsCaller
```

Il reste à exécuter le code :

```
&$SbPrivate
```

Lors de l'exécution de ce scriptblock, Powershell crée une nouvelle portée, on doit donc adresser le scope du parent, ici la session courante ayant appelé le module.

Bien évidement si on utilise l'opérateur dotsource pour exécuter le scriptblock, le paramétrage de la portée de **New-Variable** n'est plus nécessaire.

Si la variable existe déjà, le cmldet **New-Variable** émet une erreur :

```
New-Variable : A variable with name 'FromModule' already exists.
```

A vous de décider du comportement souhaité.

6 Demi-tour

Jusqu'ici nous avons vu comment modifier l'appelant à partir du module, voyons maintenant l'inverse, comment modifier module à partir de l'appelant.

6.1 De la session vers le module

Pour accéder à l'état de session du module on procède ainsi :

```
$M=Get-module Test #Ou  $M=IMPO .\Test.psm1 -passthru
&$M { Write-host "$Mavar"}
```

6.2 Configurer des options internes via une fonction externe

Pour le projet Psionic, j'ai rencontré un autre problème.

Je laisse la possibilité à l'utilisateur de configurer ses options par défaut en déclarant une fonction externe au module Psionic, cette fonction doit être chargée avant l'import.

Cette fonction externe appelle une fonction publique du module qui configure les options. A ce stade, les fonctions du module ne sont pas encore accessibles à la fonction externe, je ne peux donc utiliser la clé *ScriptToProcess* ni exécuter cette méthode externe dans le module car les éléments exportés par **Export-ModuleMember** ne sont accessibles qu'une fois l'appel à Import-Module terminé.

Dans ce cas on doit lier le code de la fonction externe au code du module en appelant la méthode

NewBoundScriptBloc :

```
$FunctionBounded=$MyInvocation.MyCommand.ScriptBlock.Module.
                 NewBoundScriptBlock(
                    ${function:Get-PsIonicDefaultSfxConfiguration}
                 )

 #configure les options
&$FunctionBounded|Set-PsIonicSfxOptions
```

De cette façon on laisse faire Powershell et on ne se préoccupe pas des états de session. On doit tout de même préciser notre intention.