Création d'objet imbriqué sous PowerShell.

Par Laurent Dardenne, le 13/01/2014.

Ce tutoriel aborde la création d'objet composé, c'est-à-dire que certains de ses membres seront eux-mêmes des PSObjects.

Le principe mis en œuvre est une agrégation de *PSObject*, ce qui implique que la suppression de l'objet conteneur supprimera les objets contenus.

Conçu avec Powershell version 2 sous Windows Seven 64.

Merci à Matthew Betton pour sa relecture.

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

Les fichiers sources :
http://ottomatt.pagesperso-orange.fr/Data/Tutoriaux/Powershell/La-creation-objet-imbrique-sous-PowerShell/Sources.zip

Chapitres

1 Rappel sur la construction d'un membre d'objet

Lors de la création d'un objet Powershell personnalisé ses propriétés sont par défaut du type *PSNoteProperty* :

```
$UnObjet = New-Object PSCustomObject -Property @{Nom='Test'}
$UnObjet|Get-Member -Name Nom
```

```
   TypeName: System.Management.Automation.PSCustomObject
Name   MemberType   Definition
----   ----------   ----------
Nom    NoteProperty System.String Nom=Test
```

Le type de son contenu est celui de la valeur qui lui est affectée, ici *System.String*. Si on lui affecte une nouvelle valeur d'un autre type cela ne déclenchera pas d'erreur :

```
$UnObjet.Nom=10
$UnObjet|Get-Member -Name Nom
```

```
   TypeName: System.Management.Automation.PSCustomObject
Name   MemberType   Definition
----   ----------   ----------
Nom    NoteProperty System.Int32 Nom=10
```

Le type du membre ne change pas, celui-ci est toujours du type *PSNoteProperty*. Seul son contenu est concerné.

Ceci est également valable pour cette construction :

```
$Personne = New-Object PSCustomObject -Property @{
                    Nom='Test';
                    Adresse=(New-Object PSCustomObject -Property @{
                              Adresse1='';
                              CP=0;
                              Ville=''})}
$Personne.Adresse=10
```

Pour rappel, une variable peut être typée :

```
   #Crée une variable typée et lui affecte une valeur
[System.Int32]$MaVariable= 10


   #Récupère un objet variable
$ObjetVariable=Get-Variable MaVariable
```

On manipule ici un objet variable qui décrit la variable nommée *MaVariable* contenant l'entier 10 :

```
$ObjetVariable.Name
```

```
MaVariable
```

```
$ObjetVariable.Value
```

```
10
```

On manipule le conteneur et pas le contenu. L'information du type est portée par la propriété *Attributes* :

```
$ObjetVariable.Attributes
```

```
TypeId
------
System.Management.Automation.ArgumentTypeConverterAttribute
```

Ici il s'agit plutôt d'une contrainte sur le contenu de la variable :

```
$ObjetVariable.Attributes[0].PsObject.TypeNames
```

```
System.Management.Automation.ArgumentTypeConverterAttribute
System.Management.Automation.ArgumentTransformationAttribute
System.Management.Automation.Internal.CmdletMetadataAttribute
System.Attribute
System.Object
```

L'affectation d'une valeur d'un autre type déclenchera une erreur :

```
$MaVariable= 'Test'
```

```
MaVariable : Impossible de convertir la valeur « Test » en type « System.Int32 »…
```

Etant dans un shell, ce mécanisme de contrainte permet, entre autre, des conversions implicites. Si on déclare notre variable sans la typer, aucune contrainte n'existe :

```
Remove-Variable MaVariable,ObjetVariable
$MaVariable= 10
$ObjetVariable=Get-Variable MaVariable
$ObjetVariable.Attributes
```

```
# Aucun attribut, c'est-à-dire aucune contrainte
```

```
$MaVariable= 'Test'
```

```
# Aucune erreur
```

Ce mécanisme de contrainte n'existe pas pour les membres d'un PSObject, mais on peut le coupler avec une classe du runtime Powershell.

1.1 Construction d'un membre typé à l'aide de la classe PSVariableProperty

A la différence de la classe *PSNoteProperty*, la classe *PSVariableProperty* permet de créer des membres (des propriétés) dont le contenu est contraint.

Le nom de classe indique que le membre sera construit à partir d'un objet de type *PSVariable*, obtenu par exemple par un appel à **Get-Variable** et c'est cette variable qui portera la ou les contraintes.

NOTE : Le paramètre *MemberType* du cmdlet **Add-Member** ne permet pas de manipuler ce type de membre, on doit donc construire ce type de membre à l'aide des classes dotnet du framework Powershell.

Par la suite j'utiliserai de préférence le terme 'contraint' au lieu de 'typé', car une variable ne peut être que d'un seul type, mais il reste possible de lui ajouter plusieurs contraintes *via* sa collection *Attributs*. Sur ce sujet vous pouvez consulter le tutoriel suivant :

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

Pour créer une instance la classe *PSVariableProperty*, on créé dans un premier temps une variable contrainte :

```
[System.Int32]$MaVariable= 10
```

Dans un second temps on récupère l'objet variable :

```
$ObjetVariable=Get-Variable MaVariable
$ObjetVariable
```

Puis on appelle le constructeur de la classe ***PSVariableProperty*** en lui passant en paramètre l'objet *PSVariable* :

```
$PSVP = New-Object `
System.Management.Automation.PSVariableProperty($ObjetVariable)
```

Dans un troisième temps, on crée notre objet orienté utilisateur :

```
$UnObjet = New-Object PSCustomObject
```

Enfin, on ajoute dans sa liste de membre notre objet de type ***PSVariableProperty*** :

```
$UnObjet.PSObject.Members.Add($PSVP)
```

Affichons notre objet :

```
$UnObjet|Fl
```

```
MaVariable : 10
```

On constate que le nom du nouveau membre, de type propriété, est le nom de la variable utilisée au tout début du cycle de la construction de ce membre.

Vérifions le type du nouveau membre :

```
$UnObjet.PSObject.Members.Match('MaVariable')
```

```
MemberType    : NoteProperty
IsSettable    : True
IsGettable    : True
Value         : 10
TypeNameOfValue : System.Int32
Name          : MaVariable
IsInstance    : True
```

L'affichage indique toujours un membre de type *PSNoteProperty*, bien que ce ne soit pas tout à fait le cas en interne :

```
$UnObjet.PSObject.Members.Match('MaVariable')|gm
```

```
   TypeName: System.Management.Automation.PSVariableProperty
Name        MemberType   Definition
----        ----------   ----------
Copy        Method       System.Management.Automation.PSMemberInfo Copy()
```

```
$M=$UnObjet.PSObject.Members.Match('MaVariable')
$M[0].psobject.TypeNames
```

```
System.Management.Automation.PSVariableProperty
System.Management.Automation.PSNoteProperty
System.Management.Automation.PSPropertyInfo
System.Management.Automation.PSMemberInfo
System.Object
```

Testons l'affectation d'une nouvelle valeur d'un type différent de celui attendu :

```
$UnObjet.MaVariable='Test'
```

```
. : Impossible de convertir la valeur « Test » en type « System.Int32 ».
Erreur : « Le format de la chaîne d'entrée est incorrect. »
```

Le contenu de notre membre est bien contraint sur le type.

Revenons manipuler la variable à l'origine de la construction du nouveau membre :

```
$MaVariable =987
$UnObjet|fl
```

```
MaVariable : 987
```

C'est un effet de bord à connaitre.

Ici la durée de vie de la variable, associée au membre, impacte la cohérence de l'objet $UnObjet, c'est à dire que si, dans la même portée, vous modifiez ou supprimez la variable *$MaVariable* alors le contenu du membre nommé 'MaVariable' de notre objet utilisateur sera modifié.

Pour éviter ceci, on doit créer une variable dans une portée différente :

```
function New-VariableInScope {
   #On crée une variable dans une nouvelle portée
  $Nom='LocalScope'
   # On renvoi un objet de type PSVariable
  Get-Variable Nom
}
```

```
$PSVP = New-Object System.Management.Automation.PSVariableProperty (New-VariableInScope)
$UnObjet = New-Object PSCustomObject
$UnObjet.PSObject.Members.Add($PSVP)


$UnObjet
```

```
Nom
---
LocalScope
```

Une fois le membre créé, seul l'accès à la valeur du membre *Nom* reste possible, car la variable utilisée par le constructeur ***PSVariableProperty*** n'est plus accessible.

Le fait de sortir de la portée de la fonction supprime bien la variable dans le provider de fonction, mais ne supprime pas l'objet qui lui a été associé. C'est lors de la suppression de l'objet utilisateur (*$UnObjet)* que l'objet associé au membre sera supprimé, car seul $UnObjet possède une référence sur cette variable interne, la chaîne de caractères 'LocalScope'. Cette construction s'appuie donc sur le mécanisme de gestion des références d'objet du framework dotnet, le fameux Garbage Collector.

1.2 Créer un membre dont le contenu est en lecture seule

Notez que la modification du contenu du membre '*Nom*' reste possible :

```
$UnObjet.Nom=10
$UnObjet
```

```
Nom
---
10
```

Il est possible de créer des membres en lecture seule en agissant sur le contenu des propriétés *Options* et *Attributs* de notre ***PSVariable***. Cette opération doit être réalisée avant la création de l'instance ***PSVariableProperty*** :

```
$V= New-VariableInScope
$V.Options="ReadOnly"
$PSVP = New-Object System.Management.Automation.PSVariableProperty ($V)
$UnObjet = New-Object PSCustomObject
$UnObjet.PSObject.Members.Add($PSVP)
$UnObjet.Nom='impossible'
```

```
. : Impossible de remplacer la variable Nom, car elle est constante ou en lecture seule.
```

Note : la modification de la propriété *Visibility*, sur l'objet PsVariable $V, n'a pas d'impact sur l'accès à son contenu (ici l'entier 10), car cette propriété modifie uniquement l'accès à l'objet variable **$Nom** référencée par $V :

```
$Nom='LocalScope'
$v=Get-Variable Nom
$Nom
```

```
LocalScope
```

```
$v.Visibility='private'
$Nom
```

```
Nom : Impossible d'accéder à la variable « $Nom », car il s'agit d'une variable privée
```

```
&{$Nom}
```

```
LocalScope
```

```
.{$Nom}
```

```
LocalScope
```

1.3 Créer un membre de type tableau dont le contenu est en lecture seule

Si la valeur d'un membre est un tableau, le code précédent ne protégera que la référence du tableau :

```
function New-VariableInScope {
   #On crée une variable dans une nouvelle portée
  [int[]]$Tableau=@(1..5)
   # On renvoi un objet de type PSVariable
  Get-Variable Tableau
}

$V= New-VariableInScope
$V.Options="ReadOnly"
$PSVP = New-Object System.Management.Automation.PSVariableProperty ($V)
$UnObjet = New-Object PSCustomObject
$UnObjet.PSObject.Members.Add($PSVP)
$UnObjet.Tableau +=99
```

```
[. : Impossible de remplacer la variable Tableau, car elle est constante ou en lecture seule.
```

On ne peut pas ajouter un nouveau membre car dans ce cas le framework dotnet crée une nouvelle instance d'un tableau et l'affecte au membre.

En revanche les éléments du tableau sont toujours en lecture/écriture :

```
$UnObjet.Tableau
```

```
1..5
```

```
$UnObjet.Tableau[0]=99
$UnObjet.Tableau
```

```
99..5
```

Pour créer un tableau dont les éléments sont en lecture seule, on utilisera la fonction suivante :

```
Function New-ArrayReadOnly {
 param([ref]$Tableau)
   #La méthode AsReadOnly retourne un wrapper en lecture seule pour le tableau spécifié.
   #On recherche les informations de la méthode publique spécifiée.
  [System.Reflection.MethodInfo] $Methode = [System.Array].GetMethod("AsReadOnly")

   #Crée une méthode générique
   #On précise le même type que celui déclaré pour la variable $Tableau
  $MethodeGenerique = $Methode.MakeGenericMethod($Tableau.Value.GetType().GetElementType())

   #Appel la méthode générique créée qui renvoi
   #une collection en lecture seule, par exemple :
   # [System.Collections.ObjectModel.ReadOnlyCollection``1[System.String]]
  $TableauRO=$MethodeGenerique.Invoke($null,@(,$Tableau.Value.Clone()))
  ,$TableauRO
} #New-ArrayReadOnly
```

Celle-ci crée un wrapper en ReadOnly autour du tableau d'origine.

Ensuite on modifie la fonction de création de la variable

```
function New-VariableInScope {
   #On crée une variable dans une nouvelle portée
  [int[]]$Tab=@(1..5)
      #protége les éléments du tableau
  $Tableau=New-ArrayReadOnly ([ref]$Tab)
   # On renvoi un objet de type PSVariable
  Get-Variable Tableau
}
```

Puis on construit le membre de la même manière :

```
$V= New-VariableInScope
$V.Options="ReadOnly"
$PSVP = New-Object System.Management.Automation.PSVariableProperty ($V)
$UnObjet = New-Object PSCustomObject
$UnObjet.PSObject.Members.Add($PSVP)
```

Cette fois le tableau et les éléments du tableau sont bien en lecture seule :

```
$UnObjet.Tableau +=99
```

```
[. : Impossible de remplacer la variable Tableau, car elle est constante ou en lecture seule.
```

```
$UnObjet.Tableau[0]=99
```

```
[ : Impossible d'indexer dans un objet de type
System.Collections.ObjectModel.ReadOnlyCollection`1[[System.Int32, mscorlib, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]..BuildObject
```

2 Créer un membre en lecture seule

La construction précédente protège le contenu du membre, mais pas le membre en lui-même, car celui-ci peut être supprimé de la liste des membres adaptés :

```
$UnObjet.psobject.Members.Remove('Nom')
$UnObjet.Nom='impossible'
```

```
. : La propriété « Nom » est introuvable sur cet objet ; assurez-vous qu'elle existe et qu'elle peut être définie.
```

Vous pouvez laisser cette possibilité si vous jugez que l'utilisateur/utilisatrice sait ce qu'il/elle fait, sinon utilisez une classe C# qui vous permettra de combiner la construction de membres statiques dont certains seront basés sur un PSObject. Seule l'instance liée à ces derniers sera protégée, ainsi il restera possible d'avoir une propriété de type PSObject sans aucun membre, mais qui sera toujours différent de ***$null***.

```
Add-Type @"
using System;
using System.Management.Automation;
namespace ClassesPS
{
    public class Personne
    {
        public Personne(String aNom)
        {
            nom= aNom;
            Informations= new PSObject();
            roinformations= new PSObject();
        }

        private readonly string nom;
        public string Nom
        {
            get { return nom; }
        }
        private readonly PSObject roinformations;
        public PSObject ROInformations
        {
            get { return roinformations; }
        }
        public PSObject Informations;
    }
}
"@
```

Créons une instance de cette classe :

```
$Personne=New-Object ClassesPS.Personne 'Dupont'
$Personne
```

```
Nom                 ROInformations      Informations
---                 --------------      ------------
Dupont
```

L'affichage des membres *ROInformations* et *Informations* nous indique que ce sont juste des PSObject sans membre :

```
$Personne.Informations -eq $null
```

```
false
```

```
$Personne.ROInformations -eq $null
```

```
false
```

La modification du membre *Informations* est possible :

```
$Personne.Informations=New-Object PSObject
```

Celle du membre *ROInformations* ne l'est pas :

```
$Personne.ROInformations=New-Object PSObject
```

```
. : « ROInformations » est une propriété ReadOnly.
```

En revanche il est possible de modifier sa structure, par exemple ajoutons un membre synthétique basé ***PSVariableProperty*** :

```
$path='Votre_Répertoire_Des_Sources'
Import-Module "$path\PsObjectHelper.psm1"

$Personne.ROInformations.PSObject.Properties.Add(
  (New-PSVariableProperty 'Age' 25 -ReadOnly)
)
$Personne
```

```
Nom                 ROInformations      Informations
---                 --------------      ------------
Dupont              @{Age=25}
```

A son tour le membre est en lecture seule :

```
$Personne.ROInformations.Age=20
```

```
. : Impossible de remplacer la variable Age, car elle est constante ou en lecture seule.
```

On peut donc considérer que notre objectif est atteint, à savoir construire un objet ayant des membres de type PSObject en lecture seule.

Attention, pour ce type de construction soyez attentif aux objets implémentant IDisposable.

Pour la notion de 'Disposable voir ce tutoriel :

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

Note :

Les variables utilisées pour la construction d'un membre de type **PSVariableProperty** peuvent être des paramètres d'une fonction :

```
Function New-ItemInformations{
#Crée un objet personnalisé avec les propriétés Size et Datas (en RO)
param(
   [Parameter(Mandatory=$True,position=0)]
  [int64]$Size,
   [Parameter(position=1)]
   [AllowNull()]
  [string[]]$Datas=@()
)
    #Les paramétres liés définissent les propriétés de l'objet
  $O=New-Object PSObject
  $O.PSObject.Properties.Add( (New-PSVariableProperty Size $Size -
ReadOnly) )
    #Protége les éléments du tableau
  $DatasRO=New-ArrayReadOnly ([ref]$Datas)
    #Protége la valeur du membre
  $O.PSObject.Properties.Add( (New-PSVariableProperty Datas $DatasRO -
ReadOnly) )
  $O.PsObject.TypeNames.Insert(0,"ItemInformations")
  $O
}#  New-ItemInformations

$path='Votre_Répertoire_Des_Sources'
Import-Module "$path\PsObjectHelper.psm1"

$o=New-ItemInformations 150 @(1..5)
```

2.1 Exemple de construction d'un objet imbriqué

Dans les sources vous trouverez les scripts suivants :

Hashtable Imbriquée.ps1	Construit un objet dont la structure est basée sur des hashtables imbriquées.
TransformHashtable Imbriquee.ps1	Construit un objet dont la structure est basée sur des PSObject imbriqués.

L'usage de hashtables permet d'accéder aux valeurs des membres en lecture/écriture, là où l'usage de membres de type PSVariableProperty permet de les protéger et de les contraindre sur le type si on le souhaite. Note : Ces exemples n'ont pas de gestion d'erreur. Les fonctions *New-Disk* et *New-LogicalDisk* ne déclarent pas de types sur leurs paramètres.

3 Accéder à un membre privé

On peut parfois vouloir interdire à l'utilisateur l'accès direct à une donnée d'une instance, mais autoriser son accès à partir de l'instance adapté. Dans ce cas on utilisera comme intermédiaire un membre de type **PSCodeProperty.**

Le code sera constitué de la déclaration de la classe *Personne* et de celle hébergeant les accesseurs de la propriété de type **PSCodeProperty** :

```
public class Personne
{
    public Personne(String aNom)
    {
        nom= aNom;
        Datas= new PSObject();
    }

    private readonly string nom;
    public string Nom
    {
        get { return nom; }
    }

    protected internal PSObject Datas;
}
```

Dans cet exemple la propriété *Informations* est remplacée par celle nommée *Datas*.

Rappel sur les modificateurs d'accès :

Protected : Seul le code de la même classe ou du même struct, ou d'une classe dérivée de cette classe, peut accéder au type ou au membre.

Internal : Tout code du même assembly, mais pas d'un autre assembly, peut accéder au type ou au membre.

Pour notre exemple, le code de la classe *Personne* et celui de la classe implémentant les accesseurs de la propriété de type **PSCodeProperty** doivent être dans le même assembly.

Code de la classe servant ‘d’intermédiaire’ **:**

```
public class AdapterCodeProperty
{
    public static Object DatasGet(PSObject psobject)
    {
        return ((Personne)psobject.BaseObject).Datas;
    }

    public static void DatasSet(PSObject psobject, Object value)
    {
        //L'objet imbriqué peut être manipulé par ailleurs
        if ((psobject == null) || (psobject.BaseObject == null) )
         throw new ArgumentNullException();

         //Régle de gestion
        if  (value == null)
         throw new ArgumentNullException();

          //On s'assure de toujours manipuler un PSObject
     ((Personne)psobject.BaseObject).Datas=PSObject.AsPSObject(value);
    }
}
```

Pour cet exemple, le code des accesseurs manipule le paramètre ‘value’ comme une instance de la classe **Personne**, bien évidemment l’accès direct à la collection *properties* d’un simple PSObject est tout à fait possible et ce sans transtypage.

La classe **AdapterCodeProperty** déclare deux méthodes statiques qui doivent respecter la règle suivante : Le type de l'objet renvoyé par le getter doit être du même type que celui du paramètre ‘value’ du setter.

3.1 Ajout du membre de type CodeProperty

Une fois le code de ces deux classes compilé :

```
$path='Votre_Répertoire_Des_Sources'
. .\AdapterCodeProperty.ps1
```

Il reste à associer les accesseurs à un membre personnalisé d'une instance adaptée de la classe *Personne* (un PSObject donc) :

```
$Personne=New-Object 'ClassesPS.Personne' 'Dupont'
```

On doit d'abord récupérer les informations des accesseurs :

```
$InformationGetter=[ClassesPS.AdapterCodeProperty].GetMethod('DatasGet')
$InformationSetter=[ClassesPS.AdapterCodeProperty].GetMethod('DatasSet')
```

Puis déclarer le nouveau membre les utilisant :

```
Add-Member -InputObject $Personne -MemberType CodeProperty -Name
Informations -Value $InformationGetter -SecondValue $InformationSetter
```

Le PSObject renvoyé par le getter autorise toujours la déclaration de membres contraint :

```
$Personne.Informations.PSObject.Properties.Add( (New-PSVariableProperty
'Adresse' '25 chemin des Dames') )
```

Le setter filtre, selon la règle de gestion implémentée, la nouvelle valeur et la transforme si besoin :

```
$Personne.Informations=$null
```

```
. : Exception lors de la définition de « Informations » : « La valeur ne peut pas être null. »
```

```
$Personne.Informations=10
```

```
Informations            Nom
------------            ---
10                      Dupont
```

```
$Personne.Informations=new-object psobject -Property @{Age=25}
```

```
Informations            Nom
------------            ---
@{Age=25}               Dupont
```

Si on supprime le membre ***Informations***, seul l'accès aux données est concerné, les données ne sont pas supprimées.

3.2 Cas des collections

Le paramètre ‘value’ du setter peut être une collection, dans ce cas les opérateurs d’affectation suivants sont pris en compte.

Sous réserve que la propriété pointée par le code des accesseurs est d’un type collection et plus PSObject :

= valeur	Remplace le contenu de la collection avec la nouvelle valeur.
= @(valeur,…)	Remplace le contenu de la collection existante avec la nouvelle collection.
+= valeur	Concatène la nouvelle valeur à la collection existante. *Dans ce cas value contient les anciens éléments et le nouveau.*
+= @(valeur,…)	Concatène la nouvelle collection à la collection existante. *Dans ce cas value contient les anciens éléments et les nouveaux.*

Je vous laisse tester le comportement des opérateurs d’affectation restant.

4 Conclusion

Ce type de construction d’objet permet d’utiliser les possibilités du langage C# tout en laissant à l’utilisateur/trice la possibilité d’ajouter des membres synthétique.

Ce principe de construction ayant un coût d’occupation mémoire, il ne peut convenir dans tous les cas. Par exemple si vous devez protéger plusieurs milliers d ‘objets, l’usage d’une classe C# est préférable.

Depuis la version 3, Powershell est basé sur la DLR, celle-ci permettra peut être des évolutions sur ce type de construction.