Principe du PSTypeConverter

Par Laurent Dardenne, le 29/05/2017.

Conçu avec Powershell v5.1 Windows 7 64 bits.

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

Sources des démos :
https://github.com/LaurentDardenne/Tutorial/tree/master/Principe%20du%20PSTypeConverter/Sources

Chapitres

1 Contexte

Lors de mes tests sur le versionning d'instance de ce tutoriel, je n'ai pas abordé une possibilité de Powershell nécessitant l'usage du C#. L'objectif était de réhydrater un objet qui avait été sérialisé par le remoting.

Avant d'aborder cette possibilité basée sur la classe ***PSTypeConverter***, voyons ce qui diffère entre une conversion et une réhydratation.

1.1 Conversion ou réhydratation ?

Pour aborder le sujet je reprends l'exemple suivant PowerShell Custom Types, Type conversion and ETS - The Practical Administrator.

L'objectif de cet exemple est de faciliter l'écriture de code en transformant une chaine en un type particulier, mais il n'aborde pas la transformation d'une information sérialisée , reçue d'un traitement distant (WinRM), dans son type d'origine. On peut également vouloir couvrir ces deux aspects.

1.1.1 Conversion

Dans l'exemple cité l'auteur souhaite faciliter la saisie utilisateur en la transformant en interne.

La solution, comme indiquée ici, se situe dans la conception de la classe afin d'utiliser le mécanisme de conversion de Powershell.

Prenons l'exemple de la classe IPAddress, on peut créer une instance à partir d'une chaîne de caractères sans appeler explicitement un constructeur :

```
[System.Net.IPAddress]"192.168.1.1"
AddressFamily      : InterNetwork
IPAddressToString  : 192.168.1.1
…
```

La classe IPAddress ne propose pas de constructeur utilisant une chaîne de caractères, mais dispose d'une méthode statique *Parse()* :

```
[System.Net.IPAddress]::Parse("192.168.1.1")
```

En interne Powershell utilise cette méthode pour transformer la saisie dans le type attendu :

```
Trace-Command -PSHost -Name TypeMatch, TypeConversion -Expression {
[System.Net.IPAddress]"192.168.1.1"}
DÉBOGUER : TypeConversion Information: 0 :  Found "System.Net.IPAddress"
in the loaded assemblies.
DÉBOGUER : TypeConversion Information: 0 : Converting "192.168.1.1" to
"System.Net.IPAddress".
DÉBOGUER : TypeConversion Information: 0 : Parse result: "192.168.1.1".
```

1.1.2 Réhydratation

Dans ce cas l'objectif est de récupérer une information sérialisée dans le type d'origine :

```
Trace-Command -PSHost -Name TypeMatch, TypeConversion -Expression {
 $IP=Start-Job {
  [System.Net.IPAddress]"192.168.1.1"
 } |Receive-Job -Wait -AutoRemoveJob
 $IP
}
TypeConversion Information: Converting "192.168.1.1" to
"Microsoft.PowerShell.DeserializingTypeConverter".
TypeConversion Information: Custom type conversion.
TypeConversion Information:  Destination type's converter is PSTypeConverter.
TypeConversion Information:  Destination type's converter can convert from originalType.
TypeConversion Information:  Custom Type Conversion succeeded.
AddressFamily      : InterNetwork
IPAddressToString  : 192.168.1.1
…
```

Il s'agit aussi d'une conversion mais elle s'appuie sur un autre mécanisme que celui cité précédemment. Dans tous les cas WinRM émet un PSObject (hormis les types primitifs), mais côté récepteur on peut insérer une transformation de ce PSObject vers le type d'origine à l'aide d'un **PSTypeConverter** déclaré dans un fichier ETS (.ps1xml).

La classe *DeserializingTypeConverter* indiquée dans la première ligne de Trace-Command dérive de la classe **PSTypeConverter**.

En interne la conversion se fait par l'appel de la méthode suivante :

```
// src/System.Management.Automation/engine/serialization.cs
private static System.Net.IPAddress RehydrateIPAddress(PSObject pso)
{
  string s = pso.ToString();
  return System.Net.IPAddress.Parse(s);
}
```

La solution se situe ici, non plus dans la conception de la classe, mais dans celle de l'assembly en ajoutant une classe dédiée à la conversion afin de réhydrater l'objet dans le type d'origine. Ainsi on peut ajouter une transformation de type même si on ne dispose pas du code source de la classe ciblée par la réhydratation.

Notez que dans l'exemple de création d'adresse IP, l'objet reçu est bien du type [IPAddress] mais il est également adapté en un **PSObject** afin de contenir les propriétés additionnelles *RunspaceId* et *PSSourceJobInstanceId* ajoutés par le job.

Le code de l'exemple choisi utilise uniquement la classe *NetConverter* pour ses conversions.

2 Principe de base

J'ai repris le code d'origine en l'état, l'objectif de ce tutoriel étant d'étudier les mécanismes autour des PSTypeConverter, ce code doit être amélioré pour une réutilisation en production.

2.1 Sans converter

Utilisez la démo du répertoire *Sources\Without converter*, elle charge une classe puis crée un job qui émet une nouvelle instance du type chargé :

```
$o=.\Demo.ps1
$O.GetType().Fullname
System.Management.Automation.PSObject
```

Par défaut le type de l'objet reçu est PSObject, le cmdlet Get-Member nous informe que le typename d'origine est *GetAdmin.Server*. Une fois reçu en local son typename est *Deserialized.GetAdmin.Server* :

```
$O|gm
   TypeName: Deserialized.GetAdmin.Server
Name                MemberType   Definition
----                ----------   ----------
...
Name                Property     System.String {get;set;}
Network             Property     Deserialized.System.Collections.ArrayList
{get;set;}
```

Attention, notez que pour le type de la propriété *Network* le cmdlet Get-Member affiche *Deserialized.System.Collections.ArrayList* alors que son type réel est bien un *Arraylist* :

```
$o.Network.GetType().Fullname
System.Collections.ArrayList
```

Le mécanisme de réhydratation transforme les collections en un *Arraylist*.

Cette démo met en œuvre le comportement de sérialisation/désérialisation par défaut.

2.2 Sans converter mais avec la méthode Parse()

Pour en revenir à notre exemple, l'auteur souhaite faciliter la création d'une instance à partir d'une chaîne. Avec la démo précédente ce n'est pas possible :

```
[GetAdmin.Net]"eth0,192.168.1.1,255.255.255.0"
Cannot convert the "eth0,192.168.1.1,255.255.255.0" value of type
"System.String" to type "GetAdmin.Net".
```

Ouvrez une nouvelle session et utilisez la démo du répertoire *Sources\Without converter but with Parse*.

Cette version ajoute une méthode *Parse() :*

```
Add-Type -path ".\Adapters.dll"
[GetAdmin.Net]"eth0,192.168.1.1,255.255.255.0"
Interface IPAddress   Netmask
--------- ---------   -------
eth0      192.168.1.1 255.255.255.0
```

On constate que l'ajout de la méthode *Parse()* suffit pour faciliter la saisie utilisateur.

Attention le code de la méthode *Parse* est incomplet, il manque quelques contrôles sur le contenu de la chaine reçue.

Par contre, il y reste un problème avec le contenu de la propriété *Network* :

```
$o=.\Demo.ps1
$o.Network
GetAdmin.Net
GetAdmin.Net
GetAdmin.Net
```

Elle ne contient plus les informations d'origine mais seulement les noms des types d'origine.

On doit préciser la profondeur de sérialisation de notre classe qui imbrique un objet, la collection *Network*. A moins d'utiliser le cmdlet *Update-TypeData* (Powershell v3 et supérieure), L'ajout d'un fichier ps1xml du côté de l'émetteur ET du récepteur est nécessaire.

La démo suivante **doit être exécutée en dotsource**, car le cmdlet *Update-TypeData* à une portée locale :

```
$O=. '.\Demo with SerializationDepth.ps1'
```

La modification de la profondeur de sérialisation permet de récupérer les informations d'origine :

```
$O.Network
Interface IPAddress   Netmask
--------- ---------   -------
ns0       192.168.1.1 255.255.255.0
ns0       192.168.1.2 255.255.255.0
ns0       192.168.1.3 255.255.255.0
```

En revanche le type de l'information n'est pas celle d'origine :

```
$O.Network[0].GetType().Fullname
System.Management.Automation.PSObject
$O.Network[0].PStypenames
Deserialized.GetAdmin.Net
Deserialized.System.Object
```

On reçoit un PSObject désérialisé.

2.2.1 Profondeur de sérialisation (SerializationDepth)

SerializationDepth *spécifie le niveau de sérialisation d'un objet.*

- *Une valeur de 0 sérialise l'objet, mais pas ses propriétés.*
- *La valeur de 1, sérialise l'objet et ses propriétés.*
- *Une valeur de 2 sérialise l'objet, ses propriétés et tous les objets dans les valeurs de propriété*
- *Les valeurs supérieures correspondent au niveau d'imbrication souhaité.*

Voici le contenu du fichier de type utilisé dans la démo précédente :

```
<Types>
  <Type>
    <Name>GetAdmin.Server</Name>
    <Members>
      <MemberSet>
        <Name>PSStandardMembers</Name>
        <Members>
          <NoteProperty>
              <Name>SerializationDepth</Name>
              <Value> 2 </Value>
          </NoteProperty>
        </Members>
      </MemberSet>
    </Members>
  </Type>
</Types>
```

La commande suivante retrouve les informations ETS d'un type :

```
Get-TypeData -TypeName GetAdmin.Server|Select-Object *
```

Notez que cette configuration concerne avant tout la sérialisation elle peut donc être nécessaire dans un contexte n'utilisant pas le remoting.

2.2.1.1 Profondeur de sérialisation par défaut

Pour cet exemple on doit d'abord supprimer les informations de type précédemment chargés :

```
Remove-TypeData -TypeName GetAdmin.Server
$Server=.\NewServer.ps1
$S  = [System.Management.Automation.PSSerializer]::Serialize($Server)
$Ds = [System.Management.Automation.PSSerializer]::Deserialize($S)
$S
```

```
…
<ToString>GetAdmin.Server</ToString>
 <Props>
   <S N="Name">Server1</S>
   <Obj N="Network" RefId="1">
     <TN RefId="1">
       <T>System.Collections.ArrayList</T>
       <T>System.Object</T>
     </TN>
     <LST>
       <S>GetAdmin.Net</S>
       <S>GetAdmin.Net</S>
       <S>GetAdmin.Net</S>
     </LST>
   </Obj>
…
```

Vous pouvez également constater le nombre de PSObject créés en traçant la désérialisation :

```
Trace-Command -PSHost -Name TypeMatch,TypeConversion,InternalDeserializer
-Expression {
 $Ds = [System.Management.Automation.PSSerializer]::Deserialize($S) }
```

```
…
InternalDeserializer Information:  WriteLine   Read PSObject with refId: 0
InternalDeserializer Information:  WriteLine   Read PSObject with refId: 1
InternalDeserializer Information:  WriteLine   Found container node List
InternalDeserializer Information:  WriteLine   Primitive Knowntype Element S
InternalDeserializer Information:  WriteLine   Processing start node S
InternalDeserializer Information:  WriteLine   Primitive Knowntype Element S
…
```

Deux objets sont créés, l'objet serveur et la collection, les instances de la classe *GetAdmin.Net* sont représentées dans le type primitif *String* (Primitive Knowntype Element S).

Note : Vous trouverez le détail de ces éléments XML dans le fichier *[MS-PSRP].pdf*

2.2.1 Des types presque primitifs

Attention à l'interprétation :

```
$j=Start-job {
 $List = new-object 'Collections.Generic.List[Int]'
 $List.Add(10)> $null
 $List.Add(-5)> $null
 ,$List
}
Sleep -S 4
$B=Receive-Job -id $j.id
```

Dans ce cas l'affichage peut prêter à confusion puisque l'information *TypeName* référence un type désérialisé :

```
,$B|gm
```

```
   TypeName: Deserialized.System.Collections.Generic.List`1[[System.Int32, mscorlib,
Version=4.0.0.0, Culture=neutral,PublicKeyToken=b77a5c561934e089]]
```

mais celui-ci propose des méthodes :

```
Name                MemberType          Definition
----                ----------          ----------
Add                 Method              int Add(System.Object value), int
IList.Add(System.Object value)
AddRange            Method              void AddRange(System.Collections.ICollection c)
```

Powershell réhydratant les collections dans un Arraylist :

```
$B.GetType()
```

```
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     ArrayList                                System.Object
```

Ceci est donc possible :

```
$B.Add('Test')>$null
```

2.2.1.1 Modification de la profondeur de sérialisation

Cet exemple défini une profondeur de 2 afin de sérialiser le tableau et les objets qu'il contient :

```
Update-TypeData -PrependPath '.\GetAdmin.Server.Type.ps1xml'
$S= [System.Management.Automation.PSSerializer]::Serialize($Server)
$S
```

```
…
   <ToString>GetAdmin.Server</ToString>
   <Props>
     <S N="Name">Server1</S>
     <Obj N="Network" RefId="1">
       <TN RefId="1">
         <T>System.Collections.ArrayList</T>
         <T>System.Object</T>
       </TN>
       <LST>
         <Obj RefId="2">
           <TN RefId="2">
             <T>GetAdmin.Net</T>
             <T>System.Object</T>
           </TN>
           <ToString>GetAdmin.Net</ToString>
           <Props>
             <S N="Interface">ns0</S>
             <Obj N="IPAddress" RefId="3">
               <TN RefId="3">
                 <T>System.Net.IPAddress</T>
                 <T>System.Object</T>
               </TN>
               <ToString>192.168.1.1</ToString>
               <Props>
                 <I64 N="Address">16885952</I64>
…
```

En traçant l'opération de désérialisation vous constaterez la création de nombreux PSObject ainsi que les conversions des adresses IP :

```
…
DEBUG: TypeConversion Information: 0 : Converting "192.168.1.2" to
   "Microsoft.PowerShell.DeserializingTypeConverter".
DEBUG: TypeConversion Information: 0 : Custom type conversion.
DEBUG: TypeConversion Information: 0 : Destination type's converter is PSTypeConverter.
DEBUG: TypeConversion Information: 0 : Destination type's converter can convert from
   originalType.
DEBUG: TypeConversion Information: 0 : Custom Type Conversion succeeded. …
```

La méthode *Serialize(object source)* utilise par défaut une profondeur de 1, la méthode *Serialize(object source, int depth)* permet de spécifier une autre valeur.

Sachez que :

- l'ajout d'un tel fichier .ps1xml modifie le comportement de la méthode *Serialize(object source)*,
- le cmdlet Import-Clixml est également concerné.

Il semble que la valeur par défaut dépende du contexte d'utilisation …

2.3 Avec un converter pour la classe Net

Voyons maintenant comment retrouver le type d'origine des objets la collection *Network* du précédent exemple. On parle dans ce cas de réhydratation dans le type d'origine.

Utilisez la démo du répertoire *Sources\With Net converter*.

```
$O=.\Demo.ps1
$O.Network[0].GetType().Fullname
GetAdmin.Net
$O.Network[0].PStypenames
GetAdmin.Net
System.Object
```

L'ajout d'un PSTypeConverter transforme les informations reçues dans leur type d'origine.

Cette version n'utilisant plus la méthode *Parse()* le mécanisme de conversion utilise à la place la classe **NetConverter** déclarée dans le fichier *Adapters.cs*.

```
Trace-Command -PSHost -Name TypeMatch,TypeConversion -Expression {
  [GetAdmin.Net]"eth0,192.168.1.1,255.255.255.0"}
TypeConversion Information: Custom type conversion.
TypeConversion Information: Destination type's converter is PSTypeConverter.
TypeConversion Information: Destination type's converter can convert from
originalType.
TypeConversion Information: Custom Type Conversion succeeded.
Interface IPAddress   Netmask
--------- ---------   -------
eth0      192.168.1.1 255.255.255.0
```

2.4 Retrouver la liste des types réhydratés nativement

On doit interroger une hashtable privée :

```
 # Test portable d'une version
 # cf. https://github.com/PowerShell/PowerShell/issues/1618


If (($PSVersionTable.PSVersion -as [version]) -ge '6.0.0')
{$FieldName='s_converter' }
Else
{$FieldName='converter' }
$DTCnvtr =New-Object Microsoft.PowerShell.DeserializingTypeConverter
$Property =
[Microsoft.PowerShell.DeserializingTypeConverter].getfield("converter",
"nonpublic,static")
$Converter = $Property.getvalue($DTCnvtr)
$Converter.GetEnumerator() |sort key
```

3 Ajouter un PSTypeConverter

Utilisez la démo du répertoire *Sources\With Net AND Server converter*.

Un convertisseur de type Windows PowerShell est nécessaire lorsque Windows PowerShell ne peut, à l'aide de ses convertisseurs standard, effectuer une conversion de type pour une classe cible.

La classe PSTypeConverter a deux différences principales par rapport à la classe TypeConverter. Il peut être appliqué à une famille de types (comme tous les types dérivés de System.Enum) et pas seulement un type comme dans TypeConverter.

Pour ce faire les méthodes *ConvertFrom* et *CanConvertFrom* reçoivent *destinationType* qui définit le type vers lequel nous convertissons *sourceValue.*

Pour personnaliser la conversion de type d'une classe cible on doit dériver une classe à partir de la classe abstraite **PSTypeConverter** et ce pour chaque classe que l'on souhaite réhydrater.

Pour une classe, dérivée ou non, contenant des propriétés de type scalaires ou des propriétés de 'type primitif' un seul convertisseur de type est nécessaire, pour une agrégation, un objet contenant d'autres type d'objets, un convertisseur de type par classe est nécessaire.

Usage des méthodes :

- CanConvertFrom : indique si l'on peut convertir l'objet *spécifié vers le type que le convertisseur gère,*
- ConvertFrom : convertit un *objet en une instance du type que le convertisseur gère,*
- CanConvertTo : indique si l'on peut convertir *l'objet que le convertisseur gère vers le type spécifié (*cette méthode est appelée en interne avant *ConvertTo*),
- ConvertTo : indique si l'on peut convertir *l'objet que le convertisseur gère vers le type spécifié.*

Chacune de ces méthodes propose deux surcharges, la première pour traiter un type *Object* la seconde pour traiter un type *PSObject.*

Pour le détail des appels entre l'une ou l'autre vous pouvez consulter la méthode IsCustomTypeConversion.

Je n'ai pas abordé ici le scénario où l'on déclare les deux surcharges.

3.1 Associer le PSTypeConverter à sa classe cible

Pour que Powershell puissent utiliser notre classe dérivée, on doit l'associer à sa ou ses classes cibles.

3.1.1 TypeConverterAttribute

La documentation MSDN nous indique qu'il est possible de placer un attribut en lieu et place d'un fichier de type .ps1xml :

```
    [TypeConverter(typeof(GetAdmin.SiteConverter))]
    public class Site
    {…
```

Dans ce cas la valeur de la profondeur de sérialisation par défaut sera de 0 (*sérialise l'objet, mais pas ses propriétés)* :

```
Update-TypeData -TypeName GetAdmin.Site -DefaultDisplayPropertySet Name
Get-TypeData -TypeName GetAdmin.Site|Select SerializationDepth
SerializationDepth
------------------
                 0
```

Un fichier de type est donc nécessaire pour modifier cette valeur à 1 (*sérialise l'objet et ses propriétés)*, car il n'existe pas d'attribut ***SerializationDepth*** pouvant être associé à la classe.

3.1.1 Déclarer le converter dans un fichier de type

En interne ces fichiers vont renseigner une table de type utilisé lors de la sérialisation /désérialisation des objets. D'après les commentaires du code source de Powershell les informations suivantes sont utilisées lors de de sérialisation :

- SerializationMethod
- SerializationDepth
- SpecificSerializationProperties

Et celles-ci sont utilisées lors de la désérialisation :

- TargetTypeForDeserialization
- TypeConverter

Précédemment nous avons utilisé ce type de fichier, pour cet exemple il doit contenir deux entrées, une pour la sérialisation et une pour la désérialisation.

1. On associe la classe converter à la classe qu'il cible et on précise la profondeur :

```
<Types>
    <Type>
        <Name>GetAdmin.Net</Name>
        <TypeConverter>
            <TypeName>GetAdmin.NetConverter</TypeName>
        </TypeConverter>
        <Members>
            <MemberSet>
                <Name>PSStandardMembers</Name>
                <Members>
                    <NoteProperty>
                        <Name>SerializationDepth</Name>
                        <Value>1</Value>
                    </NoteProperty>
                </Members>
            </MemberSet>
        </Members>
    </Type>
```

2. Puis on associe le nom du type Powershell (PSTypeName) au type réel :

```
    <Type>
        <Name>Deserialized.GetAdmin.Net</Name>
        <Members>
            <MemberSet>
                <Name>PSStandardMembers</Name>
                <Members>
                    <NoteProperty>
                        <Name>TargetTypeForDeserialization</Name>
                        <Value>GetAdmin.Net</Value>
                    </NoteProperty>
                </Members>
            </MemberSet>
        </Members>
    </Type>
</Types>
```

Pour les autres éléments, tel que *SerializationMethod*, vous trouverez des informations sur la page d'aide du cmdlet Update-Typedata , le fichier *types.ps1xml* livré avec Powershell contient des exemples de déclarations de ces éléments ou encore ce texte du blog Powershell.

La démo *Sources\With Net,Server AND Site converter* reprend le principe exposé ici, elle ajoute un niveau d'imbrication, la notion de site regroupant des serveurs au sein d'une collection générique. Le code utilise Log4net pour tracer les différents appels du PSTypeConverter.

3.1.2 Conversion d'une famille de types

Le code de la classe DeserializingTypeConverter propose une implémentation de gestion d'un PSConverter pour plusieurs types.

3.1.3 Implémentations avec une classe Powershell native

Ce bug sur Github relate l'impossibilité actuelle de créer un PSConverter à partir d'une classe Powershell.

3.2 Membres personnalisés

Il y a un cas à considérer qui est la gestion des membres personnalisés sur un objet à réhydrater. Le PSTypeConverter doit renvoyer à partir d'un PSObject un objet du type attendu sinon Powershell déclenche une exception. On peut donc recevoir un PSObject et mais pas en retourner un.

Dans le cas où on ajoute un membre additionnel, dans un job par exemple, et puisqu'on ne réhydrate que les propriétés du type on perd le membre additionnel :

```
$s= New-Server -Name Server1 –Network …
Add-Member -InputObject $S -MemberType Noteproperty -Name MyMember `
           –Value 'Test' -Passthru
```

Si on utilise un fichier de type identique dans l'appelant et l'appelé, les membres seront bien ajoutés mais avec leur valeur par défaut ou $null.

Une solution, je ne sais à ce jour s'il en existe une autre à l'aide d'API, est de retourner un PSObject dans lequel on insère en tant que membre l'objet à réhydrater puis de lui ajouter les membres additionnels :

```
[psobject]@{Server=$S;MyMember='Test'}
```

Ce que fait en quelque sorte le cmdlet Start-Job lors de la réception d'un objet, voir le chapitre *1.1.2 Réhydratation*

Voir aussi.