Structures de données sous PowerShell

Par Laurent Dardenne, le 19 Janvier 2009.

Niveau

Débutant Avancé Confirmé

Comme tout langage informatique PowerShell propose des structures de données qui permettent de stocker et de retrouver une donnée particulière dans un ensemble de données, le mode d'accès à cette donnée différera selon le type de la structure.

Ce tutoriel aborde les plus courantes telles que les tableaux, les hashtables, les stringbuilder, les arraylist et les énumérations.

Les débutants y trouveront les manipulations de base autour de ces structures de données, les autres des détails d'implémentation et quelques opérations d'un niveau avancé.

Je tiens à remercier **shawn12** pour ses corrections orthographiques.

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

Chapitres

1 Une histoire de collections

PowerShell en utilise principalement deux, les tableaux et les tables de hachage ou hashtable. La structure de type tableau est la plus utilisée notamment dans le pipeline mais aussi comme résultat d'un traitement qu'il soit effectué par un cmdlet, une fonction ou un scriptblock.

Ces collections de données sont bien évidemment basées sur celles de .NET qui est le socle technique de PowerShell, la plupart possèdent donc certains comportements identiques.

Ce ou ces comportements sont liés à la notion d'interfaces qui ne décrivent aucune structure de donnée, mais définissent une aptitude à faire quelques choses, par exemple dans notre cas une itération sur un ensemble de données.

L'un des comportements les plus usités pour une collection est celui d'énoncer un par un ces éléments, sous .NET il est implémenté à l'aide de l'interface **IEnumerable** (http://msdn.microsoft.com/fr-fr/library/system.collections.ienumerable.aspx)

Avant d'approfondir cet aspect, voyons comment manipuler quelques structures telles que les tableaux et les hashtables.

2 Les tableaux

Les tableaux sont implémentés en utilisant le type **System.Array** du Framework (http://msdn.microsoft.com/fr-fr/library/system.array.aspx).

Vous noterez dans la définition C# de cette classe la présence de nombreuses interfaces :

```
public abstract class Array : ICloneable, IList, ICollection, IEnumerable
```

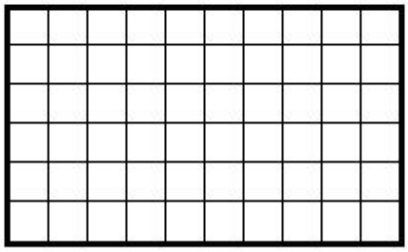
dont celle nommée **IEnumerable.** Chacune de ces interfaces ajoute un comportement spécifique, tel que l'énumération ou la copie d'objet avec **ICloneable.**

Le Framework .NET distingue 2 types de tableaux :

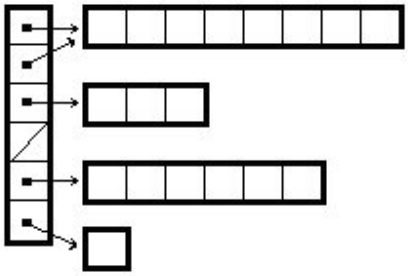
- les tableaux nommés SZ, unidimensionnels et débutants à un indice zéro, ceux de PowerShell sont par défaut de ce type,
- et les autres, c'est à dire :
  - les tableaux unidimensionnel, mais débutant à un indice différent de zéro (un indice négatif est aussi possible),
  - les tableaux multidimensionnels,
  - les tableaux de tableaux réguliers,
  - et les jagged array ou ragged array qui sont des tableaux de tableaux irréguliers c'est-a-dire qui n'ont pas le même nombre d'éléments dans toutes les directions.

Remarque : un tableau de tableaux contient des pointeurs sur d'autres tableaux alors qu'un tableau multidimensionnel n'en contient pas.

Exemple de tableau rectangulaire [6, 10] :



Exemple de tableau de tableaux référençant quatre tableaux horizontaux :



Notez que toutes les dimensions d'un tableau multidimensionnel seront de la même taille. Mais dans un tableau *jagged Array*, il est possible de référencer des tableaux de différentes tailles.

Sous .NET le type tableau SZ (unidimensionnel) débute à zéro et est optimisé. La zone mémoire allouée pour un tableau contient des informations supplémentaires qui sont :

- le nombre de dimensions (***Rank***),
- la borne inférieure du tableau (***LowerBound***),
- la longueur de chaque dimension (***GetLength***), c'est-à-dire le nombre d'éléments contenus dans la dimension spécifiée.

2.1 Les manipulations de bases

Voyons maintenant quelles opérations et syntaxes propose PowerShell autour des tableaux.

2.1.1 Création

Création d'un tableau vide :

```
$T=@()
```

D'un tableau à 1 élément contenant un entier et d'indice zéro :

```
$T=@(2)
 #ou
$T=,2
```

D’un tableau d'entiers de trois éléments d’indice zéro :

```
$T=1,2,3
 #ou
$T=1..3
```

D’un tableau de chaînes de quatre éléments d’indice zéro :

```
$T=“a”,“b”,”c”,”d”
[char[]]$T=“a”,“b”,”c”,”d”
```

Note : tous les tableaux créés précédemment sont des tableaux d’objet, hormis le dernier qui est un tableau de caractère, car on précise son type par le raccourcis *[char[]]*.

D’un tableau dans un tableau, les deux étant d’indice zéro :

```
$T=1,(2,3),4
```

Ici le premier élément est un objet de type entier, le second est un objet de type tableau.

D’un tableau d'objets d’indice zéro composé de trois éléments :

```
$T=(Get-process), (Get-date), (Dir c:\)
```

Le premier élément est un tableau contenant les processus actifs, le second est une date et le troisième un tableau de fichiers existant à la racine du disque C.

2.1.2 Accès aux éléments

Accès au deuxième élément d’un tableau, le premier se trouvant à l’indice zéro :

```
$T[1]
```

Au premier élément du second tableau :

```
$T[1][0]
```

Aux deux premiers éléments d’un tableau :

```
$T[0..1]
```

Remarque : on récupère ici un entier et un tableau d’entier.

Aux éléments indiqués, consécutifs ou non :

```
$T=1..10
$T[2,4,6,8,10]
```

Il est possible de combiner des étendues et des indices :

```
$T[1..4,6,8..10]
Cannot convert "System.Object[]" to "System.Int32".
```

L’usage de l’opérateur d’addition est nécessaire en lieu et place de la virgule :

```
$T[2..5+6+8..10]
```

Cette syntaxe peut également être utilisée lors de la création d’un tableau :

```
[int32[]]$i=2..5+6+8..10
```

On peut donc accéder aux éléments d’un tableau en utilisant un tableau en tant qu’indice :

```
$T[$i]
```

2.1.3 Tests à l'aide d'opérateur

La plupart des opérateurs peuvent être utilisés sur un tableau.

Ce tableau contient-il la valeur 3 ?

```
$T –contains 3
```

Est-ce que ce tableau ne contient pas la valeur 11 ?

```
$T –notcontains 11
```

Retourne tous les éléments égaux à 4 :

```
$T –eq 4
```

Retourne tous les éléments inférieurs à 3 :

```
$T –lt 3
```

Retourne les éléments débutant par les lettres "Po*"

```
$T= "Power","Shell","Peek","POKE"
$T –like "Po*"
 #recherche en tenant compte de la casse
$T –clike "Po*"
```

Les opérateurs **match** et **replace** se basent quant à eux sur une expression régulière :

```
 #recherche les éléments débutant par le caractère P
$T -match "^P"
 #recherche les éléments finissant par le caractère E
$T -match "E$"
 #Recherche les éléments contenant la chaîne “ee”
$T -match "ee"
#recherche les éléments ne finissant pas par le caractère E
$T -notmatch "E$"
#recherche les éléments dont le second caractère est la lettre O
$T -match "^.{1}o"
 #Expression identique, mais avec un opérateur sensible à la casse
$T -cmatch "^.{1}o"
 #Création d’un nouveau tableau dont les éléments débutant par “po”
 #sont remplacés par “to”. L’ancien tableau est détruit par le GC.
$T=$t -replace "^po","to"
$T
```

Il existe une variable impactant certaines opérations sur les tableaux, il s'agit de la variable **$OFS** (*Ouptut Field Separator*)**.** Elle contient un séparateur de champs, constitué de zéro ou plusieurs caractères, qui est appliqué lors de la conversion d'un tableau en une chaîne de caractères. Par défaut PowerShell applique un caractère espace, de plus il ne crée pas cette variable dans le provider de variable lors de son exécution.

Exemple d'usage de la variable automatique **$OSF** :

```
$OFStmp=$OFS
```

```
$OFS="+"
$T=1..10
 #affiche le contenu du tableau en tant que chaîne de caractères
"$T"
#affiche : 1+2+3+4+5+6+7+8+9+10
invoke-expression "$T"
$OFS=$OFStmp
```

2.1.4 Opérations diverses

Les cmdlets **Where-Object** (raccourci **?**) et **Foreach-Object** (raccourci **%**) seront très utiles lorsque les opérateurs classiques ne pourront être employés :

```
$R=1..10
 #Filtre les nombres pair puis les additionne
$R|? { !($_ -band 1)}|% {$Res+=$_}
$Res
30
```

Retrouver le premier élément répondant à un critère spécifique :

```
$R=(1,3,5,7,2,4,6,8,10)
$Result,$Count=0,0
$Result=$R|? { $Count++; !($_ -band 1)}|% {Break}
"Count={0} Result={1}" -F $Count,$Result
Count=5 Result=0
```

Attention, ici l'instruction *Break* n'opère aucune affectation puisqu'elle stoppe avant son terme le pipeline en cours d'exécution ; L'affectation doit donc impérativement la précéder :

```
$Result,$Count=0,0
$R|? { $Count++; !($_ -band 1)}|% {$Result=$_;Break}
"Count={0} Result={1}" -F $Count,$Result
Count=5 Result=2
```

Pour retrouver le dernier élément répondant à un critère spécifique, à la différence de l'opération précédente on parcourt ici la totalité de la collection :

```
$Result,$Count=0,0
 #renvoi le dernier élément impair
$R|? { $Count++; $_ -band 1}|% {$result=$_}
```

Savoir si tous les éléments répondent à une condition :

```
$R=1..10
$R|? {$All++;$_ -le 10 }|% -begin {$All,$Count=0,0} -process
{[void]$count++} -end {$Count -eq $All}
True
$R=1..11
```

```
$R|? {$All++;$_ -le 10 }|% -begin {$All,$Count=0,0} -process
{[void]$count++} -end {$Count -eq $All}
False
```

Certaines méthodes statiques de la classe **System.Array** peuvent être utilisées, par exemple pour inverser le contenu d'un tableau :

```
$T=@(1,2,3)
$T
[System.Array]::Reverse($T)
$T
```

Ou pour trier son contenu :

```
# Tri les 5 derniers éléments
[system.Array]::Sort($T,5,5)
$T
$T=Dir *.*|% {$_.Name} ;$T
 #Tri sur l'intégralité du tableau
[system.Array]::Sort($T);$T
```

Concaténer 2 tableaux :

```
$T=@(1,2,3)
$T2=@(4,5,6)
$Total=$T+$T1
```

Ajouter un élément à un tableau existant

```
$Total+=7
```

Attention cette opération peut s'avérer très pénalisante, par exemple au sein d'une boucle, car sous .NET un tableau n'est pas réellement redimensionné, à la place on crée un nouveau tableau puis on y recopie le contenu du tableau d'origine et le nouvel élément, enfin l'ancien tableau est détruit. Cette destruction répétitive peut surcharger le garbage collector (GC). Pour éviter ceci lors de traitements itératifs, mieux vaut utiliser un objet de type **ArrayList**.

Notez la différence de concaténation entre l'usage de la virgule et celle de l'opérateur d'addition :

```
$T1=10,20
$T2=30,40
$T3=$T1,$T2
$T3.Length
2
$T3[1]
30
40
$T3[3]
#RAS
```

```
$T3=$T1+$T2 #une réaffectation crée un nouveau tableau
$T3.Length
4
$T3[1]
20
$T3[3]
40
$T3
```

L'affichage du tableau **$T3** est identique dans les 2 cas et ne permet pas de déterminer s'il s'agit d'un tableau simple ou d'un tableau de tableau, les éléments sont tout simplement affichés les uns à la suite des autres.

2.1.5 Forcer la prise en compte d'un tableau dans le pipeline

Par défaut le pipeline manipule un tableau élément par élément :

```
$T=1..4
$T| % {$_ -is [array]}
False
False
False
False
```

Si on souhaite envoyer dans le pipeline un objet tableau et ne pas y énumérer ses éléments on utilisera la virgule comme ceci :

```
,$T| % {$_ -is [array]}
True
```

Pour les tableaux de tableau le principe est identique, à noter qu'en cas d'itération sur les éléments on récupérera bien un objet tableau, ce qui est normal :

```
$T1=10,20
$T2=30,40
$T3=$T1,$T2
$T3| % {$_ -is [array]}
True
True
,$T3| % {$_ -is [array]}
True
```

Dans une fonction on utilisera le même principe pour retourner un tableau et pas un simple objet :

```
 Return ,$T3
#ou encore
 ,$T3
```

2.1.6 Quelques détails

Reprenons la dernière création du tableau **$T** afin de détailler cet objet. Quel est son type ?

```
$T|gm
TypeName: System.Int32
```

Le cmdlet **Get-Member** peut nous induire en erreur, car ici il référence l'objet envoyé dans le pipeline qui est un des éléments du tableau et pas l'objet tableau. Pourtant, le type de **$T** est un bien un tableau :

```
$T.GetType()
IsPublic  IsSerial   Name          BaseType
True      True       Object[]      System.Array
```

Certes un tableau qui contient des objets de n'importe quel type, mais pas un tableau d'entier uniquement. Pour préciser ce type il nous faut utiliser la syntaxe suivante :

```
[int[]] $T=@(1,2,3)
```

Où *int[]* précise un tableau d'entier et *[int[]]* un raccourci de type utilisé par PowerShell. Ces raccourcis pouvant être par exemple :

```
[Int[]]     #Tableau d'entier
[String[]]  #Tableau de chaine de caractéres
[byte[]]    #Tableau d'octets
[single[]]  #Tableau de nombres flottants simples précision
[double[]]  #Tableau de nombres flottants double précision
[type[]]    #Tableau de types
[array[]]   #Tableau de tableau
[decimal[]] #Tableau de nombres décimaux
[char[]]    #Tableau de caractères
[bool[]]    #Tableau de booléens
```

Essayons d'accéder au type adapté (http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/CreationDeMembresSynthetiquesSousPowerShell/) *via* la propriété *PSBase* :

```
$T.PSBase|gm
```

Ceci nous affiche bien les propriétés de la classe **Sytem.Array,** on peut donc récupérer le nombre de dimensions, le nombre maximum d'éléments, l'indice de début et l'indice de fin sans pour autant préciser la propriété *PSBase* **:**

```
$T.Rank
1
$T.Length
4
$T.GetLowerBound(0)
0
$T.GetUpperBound(0)
3
```

Il est possible de créer un tableau en appelant directement le constructeur de la classe **System.Array** afin de préciser la taille maximum :

```
$T=new-object String[] 4
1..5|ForEach {$T[$_]=$_}
Array assignment failed because index '4' was out of range.
```

Cette création fixe la taille maximum et permet donc de contrôler les affectations en dehors des dimensions du tableau, le code suivant redimensionne le tableau à chaque itération :

```
 # Tableau de zéro élément
$T=@()
 # Suppression et reconstruction du tableau $T à chaque itération
1..5|ForEach {$T+=$_}
```

Toutes les membres de la classe **Array** ne sont pas accessibles, notamment ceux utilisant les génériques :

```
$T.PSBase|gm
 #Affichage de toutes les membres
[System.Array].GetMembers()|Select Name,membertype –unique

 #filtre les propriétés d'un objet. En natif
$T.Psbase|gm -member Properties
 # via la réflexion .NET
[System.Array].GetMembers()|Where {$_ -is `
[System.Runtime.InteropServices._PropertyInfo]}
```

Je vous laisse consulter le SDK pour le détail de ces méthodes.

2.2 Tableau multidimensionnel

Créons un tableau à plusieurs dimensions avec la syntaxe suivante :

```
$T=@((1..5),(6..10))
```

Ceci malheureusement ne crée pas le tableau attendu puisque l'on crée ainsi un tableau à une dimension de 2 éléments contenant chacun un tableau.

On doit utiliser le cmdlet **New-Object** en précisant un raccourci de type :

```
  #crée un tableau à deux dimensions de 5 éléments contenant
  #chacun un entier
$T=new-object 'int32[,]' 2,5
$T[0,0]=1+(0*5)
$T[1,0]=1+(1*5)
$T.Count
 #récupère le nombre de dimensions
$T.Rank
 #récupère le nombre d'éléments de la première dimension
```

```
$T.psbase.GetLongLength(0)
2
 #récupère le nombre d'éléments de la seconde dimension
$T.psbase.GetLongLength(1)
5
$T
 #ici l'accès à l'indice à l'aide d'une variable doit se faire par 'paire'
$Indice=1,4
$T[$indice]=10
$T[(0,1),$indice] #affiche 2 éléments

 #Raz du tableau $T
0..1|Foreach { $i=$_; 0..4|Foreach { $T[$i,$_]=0} }
 #Ou mieux
[System.Array] ::Clear($T,0,$T.Count)
```

Notez que toutes les dimensions d'un tableau multidimensionnel seront de la même taille.

2.3 Tableau de tableaux

À la différence d'un tableau multidimensionnel, un tableau de tableaux peut référencer des tableaux de tailles différentes :

```
$T=(1),(2,3),(4,5,6)
$T[0]
$T[1]
$T[1,0]
```

La dernière instruction nous affiche non pas le premier élément du second tableau, mais l'intégralité du second tableau et celle du premier :

```
2
3
1
```

Pour ce type de tableau on utilisera la notation suivante :

```
$T[1][0]
```

Il y a un léger problème avec la déclaration du tableau **$T** :

```
$T[0][0]
Unable to index into an object of type System.Int32.
$T[0].GetType()
IsPublic IsSerial Name                 BaseType
-------- -------- ----                 --------
True     True     Int32                System.ValueType
```

Comme indiqué précédemment le premier élément du tableau n'est pas considéré comme un tableau, mais comme un objet, on doit donc forcer la déclaration d'un tableau en utilisant la virgule (,) :

```
$T=(,1),(2,3),(4,5,6)
$T[0].GetType()
IsPublic IsSerial Name                BaseType
-------- -------- ----                --------
True     True     Object[]            System.Array
```

Les syntaxes suivantes sont autorisées :

```
$T[0,1]       # affiche les deux premiers tableaux
$T[2][1..2]   # affiche les deux premiers éléments du troisième tableau
```

Mais la combinaison des deux précédentes ne l'est pas :

```
 #affiche les deux premiers éléments des deux derniers tableaux
$T[1..2][0,1]
 #pour cela on utilisera le cmdlet Foreach
$T[1..2]|Foreach-Object  {$_[0,1]}

 #Tableau d'objets de différents types
$T=(,1),(dir c:\),(get-content C:\boot.ini)
…
```

2.4 Construction d'un tableau avec un indice de début supérieure à zéro

Pour construire un tel tableau on doit faire appel aux méthodes de la classe **System.Array** :

```
$LowerBounds= @(5) #déclare 1 dimension et indique le premier indice
$Lengths=@(10)    # déclare la longueur du tableau [5..15]
$Nz=[System.Array]::CreateInstance([Byte], $Lengths, $LowerBounds)
$nz[0]
 #RAS
$Nz[0]=10
Array assignment failed because index '0' was out of range.
20..29|% -begin {$I=5} -process{$Nz[$I]=$_;$I++}
$nz[5]
20
$nz[15]
#RAS
```

Notez que les accès aux indices hors du tableau **$Nz** ne provoquent pas d'erreur à la différence d'une affectation.

2.5 Construction d'un tableau avec un indice négatif

Sous PowerShell la construction de ce type de tableau n'est pas autorisée, car les indices négatifs signalent une position relative, par exemple -1 indique le dernier élément d'un tableau :

```
$T=@("Premier","Deux","Trois","Quatre")
$T[-1]
Quatre
$T[-2]
Trois
$T[-($T.Length)] #$T[-4]
Premier
```

On peut ainsi récupérer les n derniers éléments d'un tableau :

```
$T[-1..-2]
Quatre
Trois
```

La construction d'un tel tableau se fera à l'aide d'une manipulation spécifique :

```
$LowerBounds= @(-256) #déclare 1 dimension et indique le premier indice
$Lengths=@(256)    # déclare la longueur du tableau [-256..-1]
$Ng=[System.Array]::CreateInstance([Byte], $Lengths, $LowerBounds)
```

Le type du tableau contient désormais une étoile indiquant un indice négatif :

```
$Ng.GetType()
IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Byte[*]                                  System.Array
$Ng.GetLowerBound(0)
-256
$Ng.GetUpperBound(0)
-1
```

Essayons d'accéder au premier élément du tableau puis de lui affecter une valeur :

```
$Ng[-1]
 #RAS
$Ng[-1]=10
Array assignment failed because index '-1' was out of range.
```

Pour ce type de tableau l'accès à un indice négatif génère une erreur, pour manipuler ce type de tableau sous PowerShell on doit utiliser les méthodes *SetValue* et *GetValue* de la classe **System.Array** :

```
$Ng.SetValue(10,-1)
Exception calling "SetValue" with "2" argument(s): "Impossible d'élargir
du type source au type cible, car le type source n'est pas un type
primitif ou la conversion ne peut pas être effectuée."
```

Pour régler ce problème on doit caster la valeur dans le type précisé dans l'appel de *CreateInstance*, à savoir le type **Byte** :

```
$Ng.SetValue([Byte]10,-1)
$Ng.getValue(-1)
10
```

Pas très pratique, mais bon à savoir si un assembly externe utilisait une telle construction.

2.6 Créer un tableau en lecture seule

Ce code permet de créer un tableau en lecture seule à partir d'un tableau existant *via* l'appel de la méthode générique ***AsReadOnly***. Il utilise quelques méthodes du système de réflexion de .NET :

```
  #Déclare un tableau, il doit être typé
[Int32[]]$Tableau=1..10
  #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([System.Int32])
  #Appel la méthode générique créée qui
  #renvoi une collection en lecture seule, du type :
  # [System.Collections.ObjectModel.ReadOnlyCollection``1[System.Int32]]
$TableauRO=$MethodeGenerique.Invoke($null,@(,$Tableau))

$Tableau[0]=10
$TableauRO[0]=10
#Unable to index into an object of type System.Collections.ObjectModel...

  #Les 2 tableaux pointent sur le même contenu
$Tableau
$TableauRO

  #Crée un tableau unique en lecture seule
$TableauRO=$MethodeGenerique.Invoke($null,@(,$Tableau.Clone()))
$Tableau[0]=255
  #Les 2 tableaux ne pointent plus sur le même contenu
$Tableau
$TableauRO
```

```
 #Crée une variable en lecture seule sur un tableau unique
 #en lecture seule
Set-Variable -name TReadOnly -value `
 ($MethodeGenerique.Invoke($null,@(,$Tableau.Clone()))) -option Constant

$TReadOnly=$null
#Cannot overwrite variable TReadOnly because it is read-only or constant.
```

3 Les chaines de caractères

Le type *string* est un type particulier sous .NET car il est immuable comme le type tableau, ce qui est cohérent puisqu'une chaîne est basée sur un tableau de caractères :

```
$Str="string"+" .NET"
$Str[7..$Str.Length]
.
N
E
T
$Str[0].GetType()
IsPublic   IsSerial   Name        BaseType
--------   --------   ----        --------
True       True       Char        System.ValueType
```

Les éléments peuvent être vu de différentes manières :

```
 #Chaîne de caractères
$Str
 #Tableau de caractères
[char[]]$Str
 #Tableau d'octets, nécessite un double cast
[byte[]] [char[]] $Str
 #ou
$str[0..$str.Length] | Foreach { [byte] $_ }
```

L'accès à un caractère d'une chaîne est donc possible en lecture, mais pas en écriture :

```
$Str[0]=[char]::ToUpper($Chaine[0])
Impossible d'indexer dans un objet de type System.String.
```

C'est en cela qu'une chaîne de caractères est immuable, car on ne peut modifier son contenu sans recréer une nouvelle chaîne.

3.1 La classe StringBuilder

Le type *string* étant immuable les opérations de modification peuvent s'avérer pénalisante en termes de performance.

Pour optimiser ces opérations, on utilisera la classe **System.Text.StringBuilder** qui représente une chaîne mutable (http://msdn.microsoft.com/fr-fr/library/system.text.stringbuilder(VS.80).aspx).

En interne un *StringBuilder* manipule un tableau de char, sans provoquer d'allocation de nouvel objet sur le tas, et renvoie par l'appel de la méthode *ToString* une référence sur ce tableau.

Un exemple d'utilisation :

```
 #Création à partir d'une chaîne existante
$StrB = new-object System.Text.StringBuilder $Str
[void]$StrB.Append(" ")
 #Ajoute 5 caractères
for ($i=65; $i -lt 70; $i++)
 {   [void]$StrB.Append([char]$i) }
 #Affiche la chaîne
$StrB.ToString()
[void]$StrB.Replace(".NET","PowerShell")
$Str=$StrB.ToString()
$Str
string PowerShell ABCDE
```

Ici l'accès à un caractère est possible en lecture et en écriture :

```
$StrB.Chars(0)=[char]::ToUpper($StrB.Chars(0))
$StrB.ToString()
String PowerShell ABCDE
```

Puisque la plupart des méthodes de la classe **StringBuilder** renvoient une instance de type StringBuilder il est possible d'enchaîner les appels :

```
$StrB.replace($SB.Chars(0),[char]::ToUpper($SB.Chars(0)),0,1).ToString()
 #ou
$Str=(new-object System.Text.StringBuilder $Str).`
Replace(".NET","PowerShell").ToString()
```

4 La classe ArrayList

Les problèmes de performance liés aux tableaux peuvent être résolus en utilisant la classe

ArrayList (http://msdn.microsoft.com/fr-fr/library/system.collections.arraylist.aspx).

Celle-ci facilitera les opérations de suppression d'élément et de redimensionnement, car sa capacité augmente automatiquement au fur et à mesure des besoins, évitant ainsi une surcharge de travail au GC.

```
 #Initialise la collection à 5 éléments
$Liste = New-Object System.Collections.ArrayList(5)
```

L'ajout d'un élément se fait par l'appel à la méthode **Add** qui renvoi la position de l'élément dans la liste :

```
$NumeroInsertion=$Liste.Add(10)
$Liste[$NumeroInsertion]
 #[Void] indique de ne pas considérer la valeur de retour
 #on évite ainsi une opération d'affectation
[Void]$Liste.Add(9)
```

La réinitialisation se fait par l'appel à la méthode *Clear* :

```
 #Vide la liste des erreurs PowerShell
$Errors.Clear()
```

Les méthodes *RemoveXxxx* supprime un élément ou une plage d'éléments :

```
$Liste.Clear()
1..10|Foreach {[void]$Liste.Add($_)}
 #Supprime dans la liste le premier élément correspondant à 5
$Liste.Remove(5)
 #Supprime le cinquième élément
$Liste.RemoveAt(5)
 #Supprime deux éléments à partir de l'index zéro
$Liste.RemoveRange(0,2)
```

On peut utiliser cette classe pour supprimer des éléments d'un tableau :

```
$T=1..20
$L=New-Object System.Collections.ArrayList(20)
 #Ajoute le tableau
$L.AddRange($T)
 #Supprime les 10 dernier éléments
$L.RemoveRange(10,10)
 #Réaffecte le résultat
$T=$L.ToArray()
```

Bien évidemment, cette approche n'est pertinente qu'au sein d'une boucle, car on peut réaliser la même chose nativement en une seule ligne :

```
$T=$T[0..9]
```

Je vous laisse consulter le SDK qui détaille les autres méthodes proposées par cette classe.

5 Les énumérations

Une énumération est constituée d'un ensemble de constantes nommées. Chaque type d'énumération a un type sous-jacent qui peut être n'importe quel type intégral sauf *char*.

Par défaut le type sous-jacent des éléments d'une énumération est **Int**. Par défaut, le premier élément a la valeur 0, et chaque élément suivant à la valeur n+1.

Voici la déclaration d'une énumération contenant les noms anglais des jours de la semaine :

```
public enum DayOfWeek {
        Sunday = 0,
```

```
        Monday = 1,
        Tuesday = 2,
        Wednesday = 3,
        Thursday = 4,
        Friday = 5,
        Saturday = 6,
    }
```

Sous PowerShell V1 il n'est pas possible de créer des énumérations, le type **enum** en C#, en revanche on est amené à manipuler celles du Framework à l'aide de la classe nommée **System.Enum** (http://msdn.microsoft.com/fr-fr/library/system.enum.aspx).

L'affichage de tous les éléments de l'énumération *DayOfWeek* se faisant ainsi :

```
[System.Enum]::GetNames([System.DayOfWeek])
```

Selon les cas on souhaitera accéder à un élément d'une énumération soit par son nom soit par sa valeur :

```
 #Accès par le nom
[int]$TypeAce = [System.Security.AccessControl.AceType]"AccessDenied"
 #Ou
$NomType="AccessDenied"
[int]$TypeAce =[System.Security.AccessControl.AceType]$NomType
 #Ou encore
[int]$TypeAce = [System.Security.AccessControl.AceType]::AccessDenied
$TypeAce
1
 #Accès par le nom
[String]$TypeAce =[System.Security.AccessControl.AceType]1
AccessDenied
```

Les valeurs d'une énumération peuvent être combinées :

```
[int]$AccessMask =
  [System.Security.AccessControl.FileSystemRights]"ReadAndExecute, Synchronize"
1179817
 #Affichage de la valeur en hexadécimal
"{0:x}" -f $AccessMask
00120OA9
 #Affichage de la valeur en chaîne
[String]$AccessName = [System.Security.AccessControl.FileSystemRights]$AccessMask
$AccessName
ReadAndExecute, Synchronize
```

5.1 Quelques fonctions utiles

Le raccourci *[Type]* permet de récupérer le type d'une classe .NET, cette information est nécessaire pour utiliser le système de réflexion. Consultez le tutoriel C# suivant pour plus d'informations (http://emerica.developpez.com/dotnet/reflection/introduction/csharp/) .

Pour récupérer un nom d'après une valeur :

```
Function Get-EnumName
{ #Renvoi le nom d'un membre d'une énumération
  # Ex : Get-EnumName ([TYPE][System.Security.AccessControl.AceFlags]) 2
  param ([Type]$Type , [byte] $Valeur )
  [System.Enum]::GetName($Type,$Valeur)
}
```

Pour récupérer une valeur d'après un nom :

```
Function Get-EnumValue
{ #Renvoi la valeur d'un membre d'une énumération
  #Sensible à la casse
  # Ex 1 : Get-EnumValue ([TYPE][System.Security.AccessControl.AceFlags])`
  #        "ContainerInherit"
  # Ex 2 :
  #   Récupère le type de l'énumération
  #    [Type] $Tf=[System.Security.AccessControl.AceFlags]
  #    Get-EnumValue $Tf "ContainerInherit"
  param ([Type]$Type , [String] $Nom )
 ([System.Enum]::Parse($Type,$Nom)).value__
}
```

On peut créer un tableau contenant les noms des éléments d'une énumération :

```
$T=[System.IO.FileMode]|gm -static -membertype property|% {$_.Name}
$T
Append
Create
CreateNew
Open
OpenOrCreate
Truncate
 #Ou encore
[System.Enum]::GetValues([System.IO.FileMode])
```

Dans les sources proposées, vous trouverez une fonction autorisant la création d'énumération à la volée. Voir le script ***New-Enum.ps1***

Voir aussi :

PowerShell V2 CTP2, making Custom Enums using Add-Type

http://thepowershellguy.com/blogs/posh/archive/2008/06/02/powershell-v2-ctp2-making-custom-enums-using-add-type.aspx

6 Les hashtables

Ce qu'en dit le SDK :

« Une hashtable représente une collection de paires clé/valeur qui sont organisées en fonction du code de hachage de la clé. Chaque élément est une paire clé/valeur stockée dans un objet DictionaryEntry. Une clé ne peut pas être $null, contrairement à une valeur.

…

Les objets de clé doivent être immuables tant qu'ils sont utilisés comme clés dans Hashtable.

Quand un élément est ajouté à Hashtable, il est placé dans un compartiment basé sur le code de hachage de la clé. Les recherches suivantes de la clé utiliseront le code de hachage de cette clé pour ne rechercher que dans un compartiment précis, ce qui réduira substantiellement le nombre de comparaisons de clés nécessaire pour trouver un élément.

…

La capacité de Hashtable correspond au nombre d'éléments que peut contenir Hashtable. Lorsque des éléments sont ajoutés à Hashtable, la capacité augmente automatiquement par réallocation. »

L'accès dans une hashtable, à la différence d'un tableau, se fait *via* une clé qui est dans la majorité des cas une chaîne de caractères. Les indices d'un tableau ne peuvent être que numériques. De plus, cette clé documente les manipulations des éléments d'une collection.

Vous noterez dans la définition C# de cette classe la présence de nombreuses interfaces dont celle nommée **IEnumerable** :

```
public class Hashtable : IDictionary, ICollection, IEnumerable,
ISerializable, IDeserializationCallback, ICloneable
```

6.1 Les manipulations de bases

6.1.1 Création

Création d'une table de hachage vide :

```
$h =@{}
```

Créer et initialiser une table de hachage avec deux éléments :

```
$h =@{Nom="Dardenne";Prénom="Laurent"}
```

Affecter 3 à la clé nommée "Nom" :

```
$h.Nom = 3
 #ou
$h["Nom"] = 3
```

La valeur peut être de n'importe quel type.

Créer une hashtable en précisant une taille initiale :

```
$h=New-object [System.Collections.Hashtable] 10
```

6.1.2 Accès aux éléments

Retourne la valeur de la clé nommée "Nom" :

```
$h.Nom
 #ou
$h["Nom"]
```

A la différence d'un tableau, le pipeline traite une hashtable comme un objet et pas comme une collection d'éléments :

```
$h|% {$_.value}
#ras
```

On doit donc ici utiliser non pas la virgule, mais la méthode **GetEnumerator()**

```
$h.GetEnumerator()|% {$_.value}
8
Foo
```

La présence de cette méthode est imposée par l'usage de l'interface **IEnumerable**.

6.1.3 Ajouter des éléments

Ajouter une clé dynamiquement

```
$h =@{}
$h.Clé="foo"
$h.foo=8
 #accés indirect à la clé nommée foo
$h[$h.Clé]
```

6.2 Les membres de la classe hashtable

Dans un premier temps, deux propriétés sont à connaître : *Keys* et *Values*.

La propriété *Keys* renvoie une collection contenant les clés de la hashtable :

```
$h =@{foo=1;bar=2}
$h.Keys
Bar
foo
```

La propriété *Values* renvoie une collection contenant les valeurs de la hashtable :

```
$h.Values
2
1
```

On peut déjà noter qu'à l'inverse d'un tableau, l'ordre d'itération n'est pas celui de l'insertion.

Affichons le type des éléments contenus dans une hashtable :

```
$h.GetEnumerator()|% {$_.GetType()}
IsPublic IsSerial Name                     BaseType
-------- -------- ----                     --------
True     True     DictionaryEntry          System.ValueType
True     True     DictionaryEntry          System.ValueType
```

6.2.1 La classe DictionaryEntry

Cette classe définit une paire clé/valeur, sa propriété *Key* obtient ou définit la clé et sa propriété *Value* obtient ou définit la valeur.

On retrouve bien nos données, mais cette fois élément par élément et non plus par les collections respectives accessibles *via* la hashtable :

```
$h.GetEnumerator()|% {$_.Key}
bar
foo
$h.GetEnumerator()|% {$_.Value}
2
1
```

Arrêtons-nous maintenant sur quelques méthodes de la classe Hashtable.

6.2.2 Méthodes de la classe Hashtable

Ajouter un élément avec la clé et la valeur spécifiée dans une hashtable :

```
$h.Add("Nom","Valeur")
```

Supprimer tous les éléments d'une hashtable :

```
$h.Clear()
```

Créer une copie partielle d'une hashtable :

```
$h2=$h
 #$h2 pointe sur les mêmes données que $h
$h2=$h.Clone()
 #$h2 pointe sur ces propres données recopiées à partir de $h
```

Déterminer si une hashtable contient une clé spécifique :

```
$h.Contains(2)
False
```

```
$h.Contains("Bar")
True
$h.ContainsKey(2)
False
$h.ContainsKey("Bar")
True
```

Déterminer si une hashtable contient une valeur spécifique.

```
$h.ContainsValue(2)
True
$h.ContainsValue("Bar")
False
```

Récupérer un itérateur :

```
$h.GetEnumerator()
```

Supprimer un élément ayant le nom de clé spécifiée.

```
$h.Remove("Bar")
```

6.2.3 La classe **SortedList**, une hashtable ordonnée

Pour créer une hashtable ordonnée, on peut utiliser la classe **SortedList** (http://msdn.microsoft.com/fr-fr/library/system.collections.sortedlist.aspx) :

```
Function PrintKeysAndValues( [System.Collections.SortedList] $myList )  {
"`t-KEY-`t-VALUE-"
  for ( $i = 0; $i -lt $myList.Count; $i++ )  {
   "`t{0}:`t{1}" -F $myList.GetKey($i), $myList.GetByIndex($i)
  }
"inverse"
  for ( $i =$myList.Count-1; $i -gt -1 ; $i-- )  {
   "`t{0}:`t{1}" -F $myList.GetKey($i), $myList.GetByIndex($i)
  }
 }
$mySL = new-object System.Collections.SortedList
$mySL.Add("First", "Hello");
$mySL.Add("Second", "World");
$mySL.Add("Third", "!");

 # Displays the properties and values of the SortedList.
"mySL"
"  Count:    {0}" -F  $mySL.Count
"  Capacity: {0}" -F  $mySL.Capacity
"  Keys and Values:"
PrintKeysAndValues  $mySL
```

6.3 Imbrication de structure

Il est possible de créer une structure de donnée évoluée à partir d'une hashtable, ce qui peut faciliter certaines opérations.

Dans une hashtable on associe une clé unique à un objet, cet objet pouvant être un type simple, un entier, une chaîne de caractères, mais aussi un type plus complexe tel qu'un tableau, etc.

Le principe est d'imbriquer une hashtable en tant que valeur d'une entrée d'une hashtable. Ceci permet de créer les structures suivantes :

```
   # ------- Structure n° 1
 EtatCivil=
   Nom
   DateNaissance
   Adresse[]
   # ------- Structure n° 2
 Profile=
   Script
   Quota
   # ------- Structure n° 3 combinant les 2 précédentes
Utilisateur =
 EtatCivil
 Profile
```

Cette structure étant similaire au type ***record*** du langage Pascal/Delphi ou **struct** du C#.

Sa mise en œuvre se fait de la manière suivante :

```
$Utilisateur=@{}
$Utilisateur.EtatCivil=@{Nom="";DateNaissance="";Adresse=(new-object
String[] 4)}
```

On crée une structure de donnée basée sur une hashtable, la valeur de son élément nommé *EtatCivil* est également une hashtable (premier niveau d'imbrication) qui contient un élément nommé *Adresse* dont le type est un tableau (second niveau d'imbrication).

Notez que le type de donnée de chaque entrée n'est pas figé et que l'appel du constructeur de la classe **System.Array** évite la construction suivante :

```
 Adresse=@($null,$null,$null,$null)
  # --- Idem mais avec un tableau à 2 dimensions, de 2 sur 2.
#$Utilisateur.EtatCivil=@{Nom="";Age=0;DateNaissance="";Adresse=(new-`
object "String[,]" 2,2)}
```

Le mode d'accès aux éléments suit celui du type sous-jacent, par exemple

$Variable.NomDeClé.NomDeClé ou *$Variable.NomDeClé[IndiceDeTableau]*

```
$Utilisateur.EtatCivil.Nom="Durand"
$Utilisateur.EtatCivil.DateNaissance=([DateTime]::Now).AddMonths(12*-25)
$Utilisateur.EtatCivil.Adresse[1]="15 rue du faubourg "
$Utilisateur.EtatCivil.Adresse[2]="Saint-Honoré"
```

```
$Utilisateur.EtatCivil.Adresse[3]="75008 Paris"
  #--- On crée une seconde hashtable
$Utilisateur.Profile=@{}
  #---Recherche uniquement les fichiers du répertoire courant
$Files=Dir *.*|Where {!$_.PSIsContainer}
```

Notez que l'on peut ajouter au fur et à mesure des champs sans passer par une initialisation :

```
$Utilisateur.Profile.Script=$Files[1]
$Utilisateur.Profile.Quota=200
```

Ce qui nous donne à l'affichage :

```
$Utilisateur
Name                Value
----                -----
Profile             {Script, Quota}
EtatCivil           {Adresse, DateNaissance, Nom}

$Utilisateur.EtatCivil
Name                Value
----                -----
Adresse             {, 15 rue du faubourg , Saint-Honoré, 75008 Paris}
DateNaissance       16/10/1983 19:56:44
Nom                 Durand
```

6.3.1 Une étape supplémentaire

On peut aller plus loin dans la conception en ajoutant des membres synthétiques à chaque structure. Bien que la construction d'objet personnalisé soit également une autre solution.

```
  #-- Le contenu est un scriptbloc calculant l'âge
$Utilisateur.EtatCivil.Age={ ((Get-date) -`
$Utilisateur.EtatCivil.DateNaissance).TotalDays /365 -as [int]}
  #-- Exécute le scriptbloc pour calculer l'âge.
&$Utilisateur.EtatCivil.Age
  #-- Supprime la clé Age
$Utilisateur.EtatCivil.Remove("Age")


  #-- On ajoute un membre synthétique de type méthode
  #sur l'objet référencé par la clé EtatCivil
$Utilisateur.EtatCivil=$Utilisateur.EtatCivil|Add-member ScriptMethod Age`
{ ((Get-date) - $this.DateNaissance).TotalDays /365 -as [int]} -Passthru
$Utilisateur.EtatCivil|gm |Sort name
$Utilisateur.EtatCivil.Age()
```

Ce dernier appel étant plus pratique que celui d'un scriptblock.

Puisque notre objet **$Utilisateur** n'est pas un type personnalisé il n'est pas possible d'ajouter un nom de type dans la collection *PsObject.TypeNames*, on ajoute donc un membre synthétique afin de porter cette information (si besoin est) :

```
$Utilisateur.Profile=$Utilisateur.Profile|Add-member NoteProperty
PSTypeName  "Profile" -Passthru
$Utilisateur.Profile.PSTypeName
```

L'accès dynamique reste possible à la condition de placer le nom de la clé entre guillemets :

```
#Accès par une variable
$Clé1="EtatCivil"
$Clé2="Profile"
$Clé1,$Clé2|% {Write-Host "`r`n------Clé : $_ ---"; $Utilisateur."$_"}
```

Si vous souhaitez dupliquer des informations à partir d'une telle structure, utilisez la méthode ***clone*** de l'objet ciblé, car une simple affectation copiera une référence identique à l'objet ciblé.

C'est-à-dire qu'une affectation recopie l'adresse de l'objet, mais ne crée pas un nouvel objet :

```
$Copie=$Utilisateur.EtatCivil
$copie

Name                Value
----                -----
Adresse             {, 15 rue du faubourg , Saint-Honoré, 75008 Paris}
DateNaissance       17/10/1983 18:08:01
Nom                 Durand


$Copie.Nom="Dupond"
$Utilisateur.EtatCivil.Nom
 #--On modifie le même objet
Dupond
$Clone=$Utilisateur.EtatCivil.Clone()
 #--On modifie un objet différent, mais au contenu identique
$clone.Nom="Duchemin"
$Utilisateur.EtatCivil.Nom
Dupond
$clone.Nom
Duchemin
```

Il reste un petit souci, à savoir que les champs contenant d'autres structures ne sont pas clonés :

```
$Utilisateur.EtatCivil.Adresse[3]="78100"
$clone

Name              Value
----              -----
Adresse           {, 15 rue du faubourg , Saint-Honoré, 78100}
DateNaissance     17/10/1983 18:08:01
```

```
Nom                 Duchemin
```

Il faudrait donc mettre en place un système de copie complète (*Deep-Copy*) afin de copier tous les champs contenant un objet qui contient lui même un objet, etc :

```
 $Clone.Adresse=$Utilisateur.EtatCivil.Adresse.Clone()
```

6.3.2 Construction dynamique d'une hashtable

Voici un exemple à l'aide du cmdlet **Foreach**. Dans le bloc *begin* on crée une hashtable, puis dans le bloc *process* on ajoute une entrée dont la clé est un nom de l'énumération **[Environment+SpecialFolder]** et enfin on affecte à cette entrée la valeur du répertoire associé :

```
$Utilisateur.$($_.NomPropriété)=Contenu
[System.Enum]::GetNames([Environment+SpecialFolder])|`
  % -begin{$SpecialFolder =@{}} `
    -process{$SpecialFolder.$_=[System.Environment]::GetFolderPath($_)}

$SpecialFolder
Name                    Value
----                    -----
Favorites               C:\Documents and Settings\Laurent\Favoris
LocalApplicationData    C:\Documents and Settings\Laurent\Local Settings\Application Data
MyPictures              C:\Documents and Settings\Laurent\Mes documents\Mes images
...
```

Il existe d'autres possibilités autour de collections génériques, mais sous PowerShell v1 ce n'est pas le plus aisé à utiliser, la V2 devrait améliorer ce point. C'est d'autant plus regrettable, car il existe une librairie de collections proposée par Wintellect qui offre des fonctionnalités intéressantes, liste triée, liste sans doublon, liste en lecture seule, ensemble, etc.

À noter également cette astuce de Bruce Payette autour d'une hashtable et de l'affectation multiple (http://blogs.msdn.com/powershell/archive/2007/02/06/powershell-tip-how-to-shift-arrays.aspx)

Voir aussi :

-Au cœur des dictionnaires en .Net 2.0 **(**niveau avancé**)**

http://mehdi-fekih.developpez.com/articles/dotnet/dictionnaires/

-Utiliser des clés composées dans les dictionnaires **(**niveau confirmé**)**

http://www.e-naxos.com/Blog/post/2008/09/30/Utiliser-des-cles-composees-dans-les-dictionnaires.aspx (Les sources du présent article contiennent la dll compilé.)

7 À propos des types valeur et des types référence

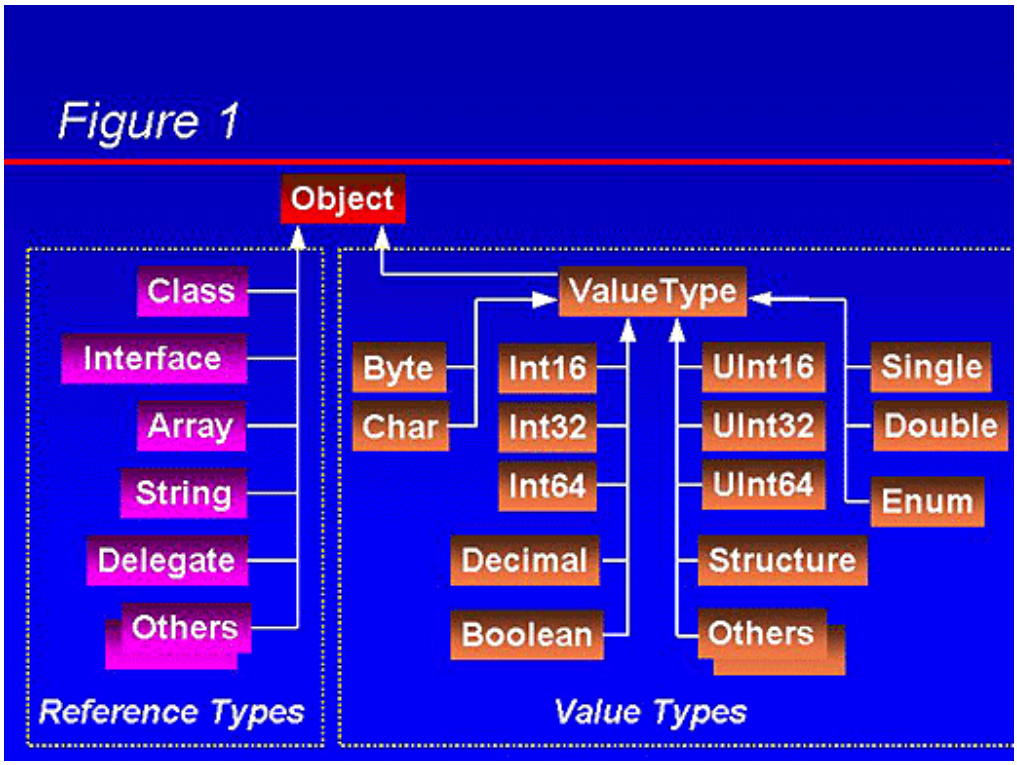
Comme nous venons de le voir certains types de donnée ont un comportement différent lors d'une recopie, il faut savoir que sous .Net les données sont de deux types :

- **type valeur**

  Les types valeur contiennent directement leurs données ; les instances de types valeur sont allouées sur la pile (durée de vie potentiellement courte).

- **type référence**

  Les types référence stockent une référence à l'adresse mémoire de la valeur et sont alloués sur le tas managé (durée de vie potentiellement longue).



Un type référence est un pointeur, mais sous .NET, à la différence de Win32, l'adresse sur laquelle il pointe peut changer (mais pas le contenu de l'objet) ceci étant dû à la réorganisation mémoire interne au Framework.

C'est pourquoi le type de l'objet a une incidence sur la copie. Pour un type valeur, créé dans la pile, la recopie suivante est valide :

```
[int32] $I=5
[int32] $J=$I
$I ;$J
5
5
```

```
$I=9
$I ;$J
9
5
```

La variable $J est bien un objet distinct de $I.

Pour un type référence, créé dans le heap (zone d'allocation mémoire), l'affectation suivante n'est pas une copie valide :

```
$T=1..10
$T2=$T
$T2[0]=20
$T
```

Le premier élément du tableau $T2 est également le premier élément du tableau $T puisque $T et $T2 pointe sur la même zone mémoire, avec une variable de type référence on utilisera la méthode *Clone* :

```
$T.Equals($T2)
True
$T=1..10
$T2=$T.Clone()
$T.Equals($T2)
False
$T2[0]=20
$T
```

Note :

Bien que le type **String** soit un type référence, la recopie crée un pointeur référençant la même chaîne on a donc en interne deux pointeurs pour une seule chaîne, mais dés que l'on modifie son contenu par l'un des deux pointeurs le Framework modifiera l'adresse du dit pointeur en créant une nouvelle chaîne (http://msdn.microsoft.com/fr-fr/library/system.string.isinterned.aspx).

7.1 Manipuler une référence, le raccourci [REF]

Dans d'autres circonstances, on aimerait manipuler une référence et pas une valeur :

```
function ChangeArray([int32[]] $T)
{
  [System.Array]::Reverse($T)
  Write-host ("Type PSreference : {0}" -F ( $T -is `
                              [System.Management.Automation.psreference]))
}
$R=10..20
ChangeArray($R) ;$R
```

Dans l'exemple précédent l'argument $T, de type tableau d'entier, est une copie du tableau $R ce qui fait que la modification n'est pas reportée dans le tableau d'origine, si on souhaite travailler sur le même objet on utilisera l'approche suivante :

```
function ChangeArray2([REF] $T)
{
  # On accède à la valeur de la référence par la propriété value
  [System.Array]::Reverse($T.Value)
  Write-host ("Type PSreference : {0}" -F ( $T -is `
                                [System.Management.Automation.psreference]))
}
ChangeArray2([ref]$R) ;$R
```

Un des précédents exemples remaniés :

```
[int32] $I=5
$J=[REF] $I
$I ;$J
5
5
$I=9
$I ;$J.Value
9
9
```

On utilise ici une variable de type valeur (**$I**) et un pointeur (**$J**) sur la variable (**$I**).

Interprétation du résultat de l'appel à la méthode *GetType* :

```
$I.GetType()
IsPublic IsSerial Name      BaseType
-------- -------- ----      --------
True     True     Int32     System.ValueType #Type valeur
$R.GetType()
IsPublic IsSerial Name      BaseType
-------- -------- ----      --------
True     False    Object[]  System.Array    #C'est donc un type référence
```

Voir aussi

- Document de Bruce Payette sur les REF : http://www.manning.com/payette/wpia_errata.pdf
- **Introduction à la gestion mémoire,** à propos des notions de pile et de tas : http://hal.archives-ouvertes.fr/docs/00/06/53/26/PDF/oz_gestion_memoire_master.pdf
- **Types**, détails des catégories du système de types C# : http://msdn.microsoft.com/fr-fr/library/3ewxz6et(VS.80).aspx

8 Création dynamique d'un type struct à l'aide de C#

Comme nous l'indique la documentation du SDK .NET :

*« Un type **struct** est un type valeur qui est utilisé en général pour encapsuler des petits groupes de variables connexes, telles que les coordonnées d'un rectangle ou les caractéristiques d'un élément dans un inventaire. »*

Le Framework utilise ce type de structure par exemple dans la classe ***DateTimeOffset***. On peut donc choisir ce type en lieu et place d'une structure imbriquée basée sur une hashtable.

Mais ceci nécessite de connaître la syntaxe de base du C# puis de générer du code dynamiquement. Sachez que par défaut les compilateurs C#, Visual Basic .NET et C++ appliquent la valeur de disposition *Sequential* aux structures, c'est à dire :

```
[StructLayout(LayoutKind.Sequential)]
Struct MonNom {....}
```

Cette information précise comment sont *marshallée* les données et pas comment elles sont stockées, voir http://msdn.microsoft.com/fr-fr/library/04fy9ya1.aspx .

L'ordre de déclaration des champs, dans une structure, n'est donc pas important dans ce cas :

```
  #Définition PowerShell de la structure nommée Disque
 Disque=@{
           Nom=[String]
           Partitions="Partition[]"
         }
```

sauf pour la déclaration du constructeur pour qui l'ordre peut ne pas correspondre :

```
  // Définition C# du code de la structure Disque
 public struct Disque {
  public Partition[] Partitions;
  public System.String Nom;

  public Disque (Partition[] partitions,System.String nom) {
    Partitions = partitions;
    Nom = nom;
 }
```

```
 #Affichage du constructeur via Get-Constructor
Disque(
       Partition[] partitions,
       String nom,
)
```

Ceci est du au fait qu'une hashtable n'est pas ordonnée.

Si utilise la version originale du script (http://poshcode.org/190) permettant cette construction, on ne peut pas créer de structures utilisant d'autres structures, à moins de le modifier pour générer un assembly externe autorisant ainsi de référencer une autre *struct*. Dans ce cas, le nom de l'assembly doit avoir l'extension .DLL et son chemin pointer de préférence sur *%Temp%*.

Vous trouverez dans le fichier *New-Struct.ps1* une possible solution modifiant le script précité.

À l'origine le script attend un nom de classe et une hashtable contenant le nom du champ ainsi que son type, voici un exemple d'appel :

```
New-Struct Song @{
 Artist=[string];
 Name=[string];
 Length=[TimeSpan];
}
```

Cette approche permet de valider sous PowerShell l'existence du type de chaque champ, la version modifiée déplace ce contrôle dans l'appel de CodeDOM et elle n'attend plus qu'un seul paramètre de type hashtable, chaque entrée de cette hashtable contient la définition d'une *struct*, cette définition étant elle-même une hashtable (comme à l'origine).

Voici quelques exemples d'utilisation, d'abord les possibles erreurs :

```
$Structs=@{D=@{Artist=[string]};
           A=@{Numero=[Int32];A="D"}}
New-Struct $Structs
```

Les noms de membres doivent être différents de leur type englobant.

```
$Structs=@{D=@{Artist=[string]};
           E=@{Numero=[Int32];Album="A"}}
New-Struct $Structs
```

Le type ou le nom d'espace de noms 'A' est introuvable, une directive using ou une référence d'assembly est-elle manquante ?

```
$Structs=@{D=@{Artist=[string]};
           E=@{Numero=[Int32];Album="d"}}
New-Struct $Structs
```

En C# les noms d'identificateur sont sensibles à la casse, l'identificateur 'd' est différent de 'D'.

Maintenant, voyons la création de structure syntaxiquement correcte (sous PowerShell) :

```
 #2 structures "basic"
New-Struct @{Song=@{Artist=[string]};
             Album=@{Numero=[Int32];titre=[String]}}
```

Une fois ces classes créées on peut les instancier :

```
$r=new-object song
$t=new-object Album
$r ; $t
```

Voyons ensuite une structure imbriquée, notez que le type A ne peut être utilisé comme type, c'est-à-dire [A], puisqu'il n'existe pas encore sous PowerShell, mais il existera lors de la compilation via CodeDOM :

```
$Structs=@{A=@{Artist=[string]};B=@{Numero=[Int32];Album="A"}}
New-Struct $Structs
$a=new-object A
$b=new-object b
$a ; $b
$b.Album
```

Creusons le sujet en utilisant des structures imbriquées utilisées par certaines API Win32 :

```
$R=@{
     Rect=@{
       Left=[UInt32];
       Top=[UInt32];
       Right=[UInt32];
       Bottom=[UInt32];
     };
    Windowinfo=@{
       cbSize=[UInt32];
       rcWindow="Rect";
       rcClient="Rect";
       dwStyle=[UInt32];
       dwExStyle=[UInt32];
       dwWindowStatus=[UInt32];
       cxWindowBorders=[UInt16];
       cyWindowBorders=[UInt16];
       atomWindowType=[UInt16];
       wCreatorVersion=[UInt16];
    }
}
New-Struct $R
$Wi=new-object WindowInfo
$Wi
$Wi.RcClient
```

Utilisons dans notre structure un champ de type tableau de structure :

```
$DisqueInfo=@{
               Partition=@{
                    Type=[Int32];
                    Taille=[Long];
               };
```

```
                Disque=@{
                     Nom=[String]
                     Partitions="Partition[]"
                }
            }
New-Struct $DisqueInfo
```

Affichage de la liste des constructeurs d'une classe, pour rappel sous .NET tout est classe, on utilisera le script suivant : http://blogs.msdn.com/powershell/archive/2008/09/01/get-constructor-fun.aspx

```
Get-Constructor Disque
 #L'affectation de $null reste possible lors de la création
$D=new-object Disque $null,"C"
 #Constructeur par défaut, tous les champs sont à $null
$D=new-object Disque


Get-Constructor Partition
 #Créations imbriquées
$D2=new-object Disque @( (new-object Partition 20Gb,1),(new-object
Partition 10Gb,1) ),"C"
$d2
$D2.Partitions
$D2.Partitions[0].Taille
```

Les limites de la solution

Redéfinition, dans la même session, d'une structure existante en mémoire :

```
$DisqueInfo=@{
            Partition=@{
                    Taille=[Long];
                   };
            Disque=@{
                    Numero=[Int32]
                    Partitions="Partition[]"
                   }
          }
New-Struct $DisqueInfo
```

L'appel de ***New-Struct*** ne pose pas de problème, mais le code n'est pas remplacé, car la nouvelle structure est, selon la démarche choisie :

- soit créée en mémoire dans un autre assembly,

- soit créée dans une DLL externe différente de la première (par défaut le nom est aléatoire), de plus une fois une DLL chargée dans un domaine d'application .NET, on ne peut plus la décharger.

```
Get-Constructor Disque
 #L'appel suivant échoue, car l'appel pointe sur le constructeur de la première déclaration
$D2=new-object Disque @( (new-object Partition 20Gb),(new-object Partition 10Gb) ),1
$D=new-object Disque $null,"C"
```

Affichons toutes les classes créées *à la volée* :

```
 #Récupère, du domaine d'application courant (la session PowerShell),
 # tous les assemblies chargés
[appdomain]::currentdomain.GetAssemblies()

  #Le code compilé à la volée ne défini pas de numéro de version
 [appdomain]::currentdomain.GetAssemblies()|`
  #On Filtre les assemblies compilés dynamiquement
 ? {$Asm=$_.GetName(); $Asm.version.tostring() -eq "0.0.0.0"}|`
  # Affiche le nom de l'assembly, généré automatiquement,
  # et le nom des types qu'il contient
 % {$_.GetTypes()|Group {$asm.Name}}
```

Pour différencier les assemblies il faut compiler les structs dans un espace de nom :

```
  $code = @"
using System;
namespace $MonNom {
  ## ICI LE CODE EST IDENTIQUE ##
  "`n  }`n }`n }"
  })
"@
```

Ensuite on accédera aux types par :

```
$D=new-object NOM1.Disque $null,"C"
$D2=new-object NOM2.Disque $null,"C"
```

Pour la création de DLL ajoutez dans le code les lignes suivantes et les paramètres adéquats dans l'entête de la fonction :

```
        $assemblies = @("System.dll", $dllName)
         # Add Referenced Assemblies for the type of a member of a struct
        if ($references.Count -ge 1)
        {
         $assemblies +=$References
```

```
        }
        ...
        $compilerParameters.GenerateInMemory = $true
         #management for assemblies in-memory
        if ($MyAssemblyName -ne [String]::Empty)
         {$compilerParameters.OutputAssembly = $MyAssemblyName}
```

Dans la ctp2 de PowerShell version 2, le cmdlet **Add-Type** utilise le même principe et semble avoir les mêmes restrictions tout en offrant de meilleurs contrôles sur les cas d'erreurs.

9 Comportement commun avec des types différents

Les interfaces permettent, entre autres, la manipulation d'objets de classes différentes, par convention le nom d'une interface débute par la lettre **I.**

Malheureusement sous PowerShell V1 l'usage des interfaces n'est pas facilité, voir ce post (http://www.eggheadcafe.com/software/aspnet/32708023/access-to-interfaces-mem.aspx).

À défaut, affichons, pour une liste de types, la liste des interfaces implémentées :

```
[System.Array], [System.Collections.Hashtable], [System.Enum],
[System.Collections.ArrayList], [System.Collections.SortedList]|`
% {Write-host $_.Name -f green; $_.GetInterfaces()}
```

Pour afficher les méthodes d'une interface :

```
[system.Array].GetInterfaceMap([System.Collections.IEnumerable])|FL
```

Maintenant affichons pour chaque interface ses méthodes et les types qui l'implémentent :

```
[System.Array], [System.Collections.Hashtable], [System.Enum],
[System.Collections.ArrayList], [System.Collections.SortedList] |`
 Foreach {
    $local:Type=$_;
     #Retrouve les membres de chaque interface
    $_.GetInterfaces()|% {$local:Type.GetInterfaceMap($_)}| Select
TargetType,InterfaceType,InterfaceMethods
    #regroupement des types par interface
 } |Group InterfaceType|Sort count -desc|% {
       #Affiche le nom de l'interface et les types l'implémentant
       # puis les méthodes de l'interface
      Write-host $("{0}. {1} types : " -F $_.Name,$_.count) -nonewline
      $OldOfs=$OFS
      $OFS=","
         # ou TargetType.FullName
       Write-host "$($_.group|% {$_.TargetType.Name})" -fore DarkGray
       Write-host "`t Méthodes : " -nonewline
```

```
        Write-host "$($_.group[0].InterfaceMethods|% {$_.Name})" -fore
DarkYellow
        Write-host
      $OFS=$OldOfs
     }
```

9.1 La boucle Foreach

Ne confondez pas la boucle *ForEach* avec l'alias du cmdlet **ForEach-Objet**, si ces deux approches opèrent une itération, seul le cmdlet propose un paramétrage avancé et ne fait pas partie du langage.

L'interface **IEnumerable** expose un énumérateur, de type *IEnumerator*, qui supporte une simple itération. C'est pourquoi certains types d'objet peuvent être utilisés au sein d'une boucle *ForEach* :

```
$T=1..10
foreach($I in $T) {Write-host $i}
```

Sur une hashtable on procédera différemment :

```
$h =@{foo=1;bar=2}
foreach($I in $h) {Write-host $i}
System.Collections.DictionaryEntry System.Collections.DictionaryEntry
```

Comme nous l'avons vu précédemment une hashtable présente en quelque sorte une collection de collections, on doit donc préciser sur laquelle itérer :

```
foreach($I in $h.Values) {Write-host $i}
 #Ou
foreach($I in $h.Key) {Write-host $i}
```

L'instruction *Foreach* appel en interne la méthode **GetEnumerator()** si l'objet n'implémente pas l'interface **IEnumerable** l'instruction ne fait rien :

```
foreach($I in [System.DayOfWeek]) {Write-host $i}
System.DayOfWeek
```

Dans ce cas elle appelle par défaut la méthode *ToString()* qui renvoie un élément, le nom du type.

Il nous faut donc rechercher un moyen de récupérer une collection à partir de cet objet :

```
foreach($I in [System.Enum]::GetNames([System.DayOfWeek])) {Write-host $i}
```

Ce qui correspond à notre besoin, car la méthode System.Enum.GetNames renvoie un tableau de chaîne de caractères :

```
[System.Enum]::GetNames|Select Value
Value
-----
static System.String[] GetNames(Type enumType)
```

Les interfaces peuvent donc être utiles pour filtrer, dans une collection d'objets de différentes classes, ceux supportant tel ou tel traitement/comportement.

Comment savoir si un objet est une collection :

```
Tab=@(1,2);$I=10
   # On teste si l'objet implémente l'interface IEnumerable
$Tab -is [System.Collections.IEnumerable]
$I -is [System.Collections.IEnumerable]
```

Est-ce qu'un objet implémente une interface générique d'un type donné ?

```
$Str="Test"
 #Test si $Str implémente System.IEquatable<String>
$Str -is [System.IEquatable``1[System.String]]
True
 #Test si $Str implémente System.IEquatable<Int32>
$Str -is [System.IEquatable``1[System.Int32]]
False
```

Une seule itération sur plusieurs collections

```
foreach($I in $h,$t) {Write-host $i}
foreach($I in 1..10+50..60) {Write-host $i}
 #erreur car la seconde collection n'est du même type
foreach($I in 1..10+50..60,$h) {Write-host $i}
 #dans ce cas, on force le type de la première collection
foreach($I in [object[]](1..10+50..60),$h) {Write-host $i}
```

Ceci est possible, car l'instruction *ForEach* supporte le pipeline :

```
foreach ($i in get-childitem | sort-object length)
{ $i ;  $sum += $i.length }
"Taille totale {0} kilo octets" -F ($Sum / 1kb)
```

9.2 Gestion des collections d'un objet COM

Les objets COM présentent leurs collections au travers de propriété, pourtant sous PowerShell leurs accès ont été adaptés :

```
 #Crée un objet Explorer
$ShellExp = new-object -comObject Shell.Application
$ShellExp|gm
 #Affiche toutes les Fenêtres
$ShellExp.Windows()
  #Les quatre appels suivants provoquent une erreur
   $ShellExp.Windows[0]
   $ShellExp.Windows(1)
   $ShellExp.Windows.Item(1)
   $ShellExp.Windows.Item[1]
 #Affiche la Fenêtre zéro
```

```
$ShellExp.Windows().Item(0)
```

Sous Excel la propriété *WorkSheet* est documentée comme étant une collection malgré cela on doit y accéder via la propriété nommée *Item* :

```
$excel = new-object -com Excel.Application
$classeurSource="$PWD\Test.xls"
$MonClasseur = $excel.Workbooks.Open($classeurSource)
  #Les deux appels suivant provoquent une erreur
   $MonClasseur.Worksheets[1]
   $MonClasseur.Worksheets(1)
 #Accéde a la première feuille XL
$MonClasseur.Worksheets.Item(1)
$Excel.Quit()
 #Force la libération des ressources
$null=[System.Runtime.InteropServices.Marshal::ReleaseComObject($MonClasseur)]
$null=[System.Runtime.InteropServices.Marshal]::ReleaseComObject($excel)
[System.GC]::Collect()
```

Notez qu'ici l'accès est légèrement différent du premier exemple.

Toujours sous Excel, certaines collections peuvent débuter à un indice égal à 1 :

```
 #Liste des Excel à modifier
$excel = new-object -comobject excel.application
$excel.visible = $true
$excelfile ="$PWD\Classeur3.xls"
$classeur = $excel.workbooks.open($excelfile)
$xlOLELinks=2
 #L'appel peut renvoyer $Null et non pas une liste vide si
 # le fichier n'a pas de liaisons
$links = $classeur.LinkSources($xlOLELinks)
$Links.gettype()
#IsPublic IsSerial Name                                     BaseType
#-------- -------- ----                                     --------
#True     True     Object[*]                                System.Array
$Excel.Quit()
 #Force la libération des ressources
…
```

Voir aussi :

http://support.microsoft.com/kb/317109
http://www.microsoft.com/technet/scriptcenter/resources/pstips/nov07/pstip1130.mspx

9.3 Comparer des objets

On souhaite parfois connaître les différences existantes entre 2 ensembles de données, le cmdlet **Compare-Object** permet cette opération, il compare un ensemble de référence avec un second ensemble :

```
$reference=1..3+8
$second=1..6
Compare-Object -r $reference -di $second -IncludeEqual
InputObject  SideIndicator
 ----------- -------------
           1 ==
           2 ==
           3 ==
           4 =>
           5 =>
           6 =>
           8 <=
```

Le résultat est un tableau d'objets personnalisés correspondant aux éléments des 2 tableaux. Chaque objet personnalisé contient la valeur à comparer et le résultat de la comparaison.

Dans notre exemple les valeurs 1 à 3 sont communes aux deux ensembles, les valeurs 4 à 6 n'existent pas dans l'ensemble de référence et la valeur 8 n'existe pas dans le second ensemble.

Par défaut ce cmdlet n'affiche que les éléments différents, d'où l'usage du paramètre *IncludeEqual*. Pour afficher uniquement les éléments appartenant aux deux ensembles on combinera les paramètres suivants :

```
Compare-Object -r $reference -di $second -ExcludeDifferent –IncludeEqual
InputObject  SideIndicator
 ----------- -------------
           1 ==
           2 ==
           3 ==
```

Comme indiqué dans ce post (http://dmitrysotnikov.wordpress.com/2008/06/06/compare-object-gotcha/) ce cmdlet contient un piège qui est que par défaut il compare chaque objet avec les 5 éléments précédents et les 5 suivants. Pour des ensembles de taille plus importante que l'exemple précédent, il faut préciser le paramètre *–SyncWindow* sinon le résultat est faussé :

```
$reference=1..15+18
 #Les éléments des 2 ensembles sont triés
$second=1..30
Compare-Object -r $reference -di $second  -IncludeEqual
 #Les éléments des 2 ensembles ne sont pas triés
$second=30..1
Compare-Object -r $reference -di $second  -IncludeEqual
```

```
 #On élargi la fenêtre de recherche
$Sync=[math]::truncate([Math]::Max($reference.count,$second.Count)/2)
Compare-Object -r $reference -di $second  -IncludeEqual -SyncWindow `
$Sync|sort inputobject
```

Ensuite on peut ventiler le résultat dans des listes :

```
Function Répartir ($list)
{
  $Res=1|Select Identique,NotInReference,NotInSecond
  $Res.Identique=New-object System.Collections.ArrayList 25
  $Res.NotInReference=New-object System.Collections.ArrayList 25
  $Res.NotInSecond=New-object System.Collections.ArrayList 25
  foreach ($Item in $List)
  {
   switch ($Item.SideIndicator)
   {
    '==' {$CurrentArray="Identique"}
    '<=' {$CurrentArray="NotInSecond"}
    '=>' {$CurrentArray="NotInReference"}
   }
  [void]$Res.$CurrentArray.Add($item)
  }
 $Res
}

$Resultat=Répartir(Compare-Object -r $reference -di $second -IncludeEqual)
$Resultat|`
 Get-Member -MemberType *Property -Name "*"|`
 %{Write-host "$("`t"*6)$($_.Name)" -fore Yellow;$Resultat.$($_.Name) }
```

Ou encore utiliser plus simplement le cmdlet **Group-Object** :

```
$Res=Compare-Object -r $reference -di $second -IncludeEqual|`
 group SideIndicator
$Res|%{Write-host "$("`t"*6)$($_.Name)" -fore Yellow;$_.group}
```

Notez que le signe == indique la présence d'un même objet dans les deux ensembles, mais n'indique pas obligatoirement leurs égalités. C'est vrai pour un type scalaire (date, entier,..) mais pas pour un type contenant de nombreuses propriétés. On peut utiliser le paramètre *–property*, dans ce cas on récupère un objet personnalisé possédant non plus une propriété *InputObject*, du type des objets à comparer, mais une propriété de même nom et du même type que celle indiquée. Il y a donc une perte d'informations dans ce cas puisqu'on ne dispose plus de référence sur l'objet d'origine.

9.4 Trier des objets

Le Framework dotNET propose la capacité de tri sur des objets de même classe en s'appuyant sur les interfaces ***IComparer*** et ***IComparable.*** Les possibilités intrinsèques de PowerShell ne permettent pas d'ajouter ce comportement sur les types de constructions personnalisés que nous avons vu précédemment. Pourtant, le tri d'un tableau d'objet implémentant une de ces interfaces reste possible.

Dans un prochain tutoriel j'aborderai une solution utilisant des délégués, des génériques et des interfaces.

En attendant, voir ce post :

How do I find an item (in an array) based on one of it's properties (PS v2 ctp2) ?

http://huddledmasses.org/custom-icomparers-in-powershell-and-add-type-for-v1/

Utilise le script suivant : http://poshcode.org/720

10 Conclusion

Vous avez pu voir tout au long des précédentes pages que l'on dispose sous PowerShell de structures de données de base qui peuvent être manipulées simplement tout en offrant des possibilités avancées, qui elles nécessitent toutefois d'approfondir les classes .NET sous-jacentes.

Les opérations de base telles que la création, l'insertion, la suppression restent aisées en revanche cela l'est moins dès que l'on souhaite effectuer des tris, des recherches ou des comparaisons.

PowerShell est donc victime de son succès, car les concepteurs ne pensaient peut-être pas qu'il allait être adopté aussi rapidement par les développeurs ou tout du moins poussé dans ces limites, la version 2 devrait améliorer certains points.

Comme pour la création d'objet personnalisé il est souhaitable de combiner le plus souvent possible du code compilé et du code PowerShell.

11 Liens

Notez que l'espace de nom *System.Collections* propose d'autres types de collections telles que les piles (*Queue, Stack*) ou les tableaux de bits (*BitArray*) :

http://msdn.microsoft.com/fr-fr/library/system.collections(VS.80).aspx

La grammaire de PowerShell vl

http://www.manning.com/payette/AppCexcerpt.pdf

Comparing Array

http://keithhill.spaces.live.com/blog/cns!5A8D2641E0963A97!6159.entry

PowerShell's -EQ operator: reference equality vs value equality

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

Une implémentation similaire à l'opérateur SQL *Join* à l'aide d'opérateur PowerShell

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

Éléments d'algorithmique, par Jean Berstel
http://www-igm.univ-mlv.fr/~berstel/Elements/

Data Structures and Algorithms Using C#
http://www.cambridge.org/aus/catalogue/catalogue.asp?isbn=9780521670159&ss=exc