L’usage du pipeline sous PowerShell

Par Laurent Dardenne, le 28 avril 2008.

Niveau

Si vous souhaitez utiliser au mieux les possibilités du langage de script dynamique de PowerShell il vous faudra approfondir les principes de fonctionnement du pipeline, c’est ce que je vous propose de faire dans ce tutoriel.

Chapitres

Pré-requis

Des notions de bases sur le langage de script de PowerShell.

Télécharger les scripts : ftp://www.ftp-developpez.com/laurent-dardenne/articles/Windows/PowerShell/Pipelining/fichiers/ScriptsPipelining.zip

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

Je tiens à remercier Guy Métayer et Vow pour leurs corrections orthographique.

1 Qu'est-ce qu'un pipeline ?

Comme nous l'indique ce dictionnaire en ligne (http://atilf.atilf.fr), un pipeline est une :

« Canalisation de gros diamètre servant au transport à grande distance de certains fluides (notamment carburants liquides, gaz naturel) et de certaines substances pulvérisées; p.méton., ensemble des canalisations et des installations permettant ce transport. »

Nous nous intéressons ici, dans un environnement informatique, à la mise en place de la canalisation et à l'élément transporté.

Sous PowerShell la mise en œuvre d'un pipeline se fait entre deux traitements, un émetteur et un récepteur.

Par défaut un cmdlet reçoit la ou les données nécessaires à son exécution *via* la ligne de commande et émet le résultat, au format texte, vers la console. Ce que fait l'exemple suivant utilisant l'alias ***dir*** :

```
PS C:\Temp> dir

   Répertoire : Microsoft.PowerShell.Core\FileSystem::C:\Temp

Mode          LastWriteTime    Length       Name
----          -------------    ------       ----
d----      08/04/2007   11:17                Adobe
d----      26/02/2008   19:03                badha5py.ce3
…
```

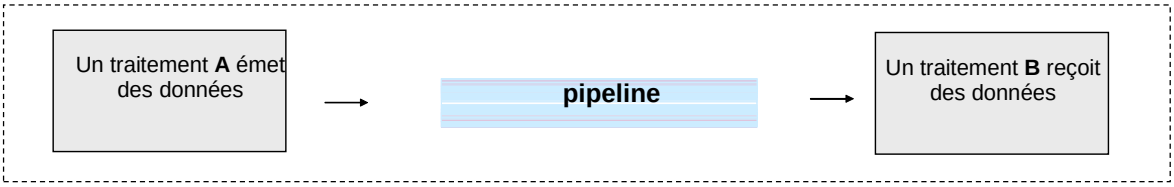
Note :

La redirection de la sortie d'un cmdlet vers un fichier, émet également du texte.

```
PS C:\Temp> dir > Resultat.txt
```

1.1 Segment d'un pipeline

Relions donc nos 2 traitements minimum pour constituer un pipeline :



Exemple :

```
Dir | Out-File "MonFichier"
```

Pour créer un pipeline on utilise le caractère '**|**', accessible par la combinaison de touches

Alt-Gr 6.

Ici le parseur de la ligne de commande détecte la présence du caractère pipe (|) puis demande au processeur de pipeline de construire une canalisation, plus exactement d'établir un canal de transmission, entre les 2 cmdlets. C'est ce qui permet d'envoyer les données résultant du traitement **A** vers le traitement suivant **B**. Notez que l'enchaînement des commandes liées se fait de gauche à droite. Il s'agit d'un mode de communication simplexe. Le traitement **B** n'envoie aucune donnée vers le traitement **A.**

Cet exemple nous indique que le résultat de l'exécution de l'alias *dir*, appelant ***Get-ChildItem***, est envoyé vers le cmdlet ***Out-File*** qui se charge d'écrire dans un fichier les données reçues.

Vu d'un cmdlet un objet pipeline (http://msdn2.microsoft.com/en-us/library/system.management.automation.runspaces.pipeline_members(VS.85).aspx) peut utiliser les trois flux d'informations suivants :

Input	Obtient l'écrivain de l'entrée du pipeline	PipelineWriter
Output	Obtient le lecteur de sortie du pipeline	PipelineReader
Error	Obtient le lecteur des erreurs du pipeline	PipelineReader

A noter qu'une affection de variable peut être vue comme l'enchaînement de commandes de droite à gauche puisqu'ici le dernier traitement *Where* renvoie le résultat dans la variable *$Var* :

```
$Var=Get-ChildItem|Where-Object {$_.Extension -match ".txt"}
```

Le host, par défaut la console PowerShell, est donc ici le lecteur de la sortie du pipeline du cmdlet ***Where-Object***.

1.2 Enchaînement de segments de pipeline

Une canalisation est constituée fréquemment d'un enchaînement de plusieurs segments de pipeline :

```
Get-ChildItem *.*|where-object {$_.PSIsContainer -eq 0}|foreach
{$_.get_Fullname(), $_.LastWriteTime}|Out-File “MesFichiers”
```

Cet enchaînement implique que le traitement **B** est récepteur du traitement **A** et émetteur vers le traitement **C**, et ainsi de suite. Les cmdlets lisent les objets dans le pipeline tant qu’ils en existent, les modifient ou non, puis les propagent ou non selon qu’ils répondent aux critères demandés.

Le cmdlet *Where-Object* peut recevoir des centaines de données mais n’en émettre qu’une ou deux.

Le plus souvent c’est le cmdlet destinataire (**D**) qui se chargera du format d’affichage des données reçues. Vous pouvez bien sûr influencer son comportement soit par l’usage de paramètres :

```
|Out-File –Witdh 256 “MesFichiers”
```

soit en insérant en amont un ou plusieurs cmdlets (**B** et **C**) modifiant les données émises par le cmdlet (**A**).

Notez que dans une canalisation le nombre de segment n'est pas limité.

Pour chaîner des commandes dans un pipeline, un cmdlet doit pouvoir y émettre des données. Par exemple *Out-File* ne peut être utilisé comme émetteur dans un pipeline, l’instruction suivante n’émet aucune donnée vers le cmdlet *Sort*:

```
Out-File test.txt |Sort
```

Sa documentation en ligne nous indique également qu’il n’accepte pas l'entrée de pipeline pour lier le paramètre *–filepath* :

```
Help Out-File -full
```

…
PARAMÈTRES
-filePath <string>
Spécifie le chemin d'accès au fichier de sortie.

Requis ? true
Position ? 1
Valeur par défaut
Accepter l'entrée de pipeline ? false
Accepter les caractères génériques ? false
…
Out-File envoie des données**, mais il n'émet pas d'objets de sortie**. Si vous dirigez la sortie de Out-File vers Get-Member, Get-Member indique qu'aucun objet n'a été spécifié.

Voir aussi les documents suivants présents dans le répertoire d’installation de PowerShell :

C:\WINDOWS\system32\windowspowershell\v1.0\fr\

- about_pipeline.help.txt
- userguide.rtf (cf. *Pipeline d'objets*, pg 28)

2 Type de donnée transportée

Le nom du cmdlet *Where-Object* précédemment utilisé nous informe qu'un cmdlet opère sur des objets .NET et non pas du texte comme les autres shell l'ont fait jusqu'à maintenant (on pourrait bien évidemment utiliser un mécanisme de sérialisation pour obtenir un comportement identique).

Dés que l'on utilise un pipe sous PowerShell les données transitent sous forme d'objet qui sont tous des instances de la classe PSObject (http://msdn2.microsoft.com/en-us/library/system.management.automation.psobject.aspx). Cette classe encapsule chaque instance d'objet .NET manipulée sous PowerShell.

Comme tout est objet sous .NET chaque cmdlet peut, à l'aide du système de réflexion (http://emerica.developpez.com/dotnet/reflection/introduction/csharp/), accéder à la totalité des informations d'une instance de classe. C'est ce que fait le cmdlet *Where-object* lorsqu'on lui demande de filtrer tous les objets reçus d'après la valeur d'une propriété d'une classe.

```
where-object {$_.PSIsContainer -eq 0}
```

Les objets émis dans un pipe peuvent être de classe différente, par exemple un fichier et un répertoire, en revanche les propriétés indiquées sur la ligne de commande devront être communes aux classes utilisées.

Dans un des précédents exemples nous avons pu voir la présence répétée des caractères '**$_** '

```
Get-ChildItem *.*|where-object {$_.PSIsContainer -eq 0}
```

Ils représentent une variable, créé automatiquement par PowerShell, contenant l'objet courant émit dans le canal de transmission. Son usage en dehors d'un segment de pipeline provoquera une erreur.

Il n'est pas nécessaire de préciser la classe de l'objet contenu dans cette variable automatique, tout ce que l'on sait est qu'il possède au moins une propriété *PSIsContainer*. L'exemple suivant ne provoque aucune erreur et ne produit aucun résultat :

```
Get-Process|where-object {$_.PSIsContainer -eq 0}
```

Un objet processus ne contient aucune propriété nommée *PSIsContainer*. Dans ce cas c'est à vous de vous assurer de la cohérence de vos instructions.

Note

Un cmdlet peut très bien modifier un objet en lui ajoutant ou masquant des propriétés ou encore le transformer comme le fait le cmdlet *Select-Object :*

```
Get-ChildItem|Where-Object {$_.Extension -match ".txt"}|select Name
```

2.1 Les consommateurs de pipeline

La grammaire (http://blogs.msdn.com/powershell/archive/2006/05/10/594535.aspx) de PowerShell nous indique que de nombreuses instructions du langage utilisent le principe du pipeline.

Ce peut être un cmdlet, une fonction, un filtre, une expression :

```
1..10|Echo
```

Ou encore une boucle foreach (à ne pas confondre avec Foreach-Object) :

```
Foreach ($P in (Get-Process|Where {$_.ProcessName -eq "svcHost"}))
 { Echo $P}
```

Pour cet exemple le pipeline s'exécute en totalité avant que l'itération débute.

Note:

Les applications externes, par exemple *ipConfig.exe*, produisent uniquement des données qui sont de type texte.

3 Usage de pipeline dans un filtre ou une fonction

Comme l'indique Bruce Payette (http://www.manning.com/payette/), un des auteurs de PowerShell, la différence la plus importante entre une fonction et un filtre est d'ordre sémantique, car toute émission d'un résultat via du code, sauf de rares exceptions, se fait dans un pipe s'il en existe un.

L'usage dans une fonction de *write-output 10* ou *Return 10* est équivalente.

Un filtre traite les objets reçus du pipeline un par un alors qu'une fonction exécute **par défaut** son traitement pour l'ensemble des données reçues dans le pipe. Dans ce cas on attend la fin du traitement contenu dans cette fonction pour passer à l'étape suivante du pipeline.

Nous verrons plus avant une autre implémentation qui permettra d'éviter ce comportement par défaut.

Mais voyons déjà le fonctionnement d'un filtre.

3.1 Filtre

Le mot clé **filter** crée un filtre permettant de manipuler les objets émis dans le pipe au travers de la variable automatique **$_** qui est en lecture/écriture :

```
filter process-r-x
{
    $_.processname -like "[r-x]*"
}
```

La variable **$_** sera renseigné uniquement par la mise en place d'un pipeline, par défaut son contenu est égal à *$Null.*

Hors pipeline l'appel de **Process-r-x** renvoie *$False :*

```
PS C:\Temp> $_.processname -like "[r-x]*"
False
PS C:\Temp> $null -like "[r-x]*"
False
```

L'usage de ce filtre dans un segment de pipeline renverra uniquement *True* ou *False* :

```
PS C:\Temp> Get-Process | process-r-x
False
True
…
```

Pour propager la transmission de l'objet courant on doit tout simplement prendre en charge sa réémission en « l'écrivant » dans le pipeline :

```
filter process-r-x
{
  If ($_.processname -like "[r-x]*")
   {
    Write-Host "Réémet l'objet filtré."
    $_ # Ré-émission de l'objet courant dans le pipeline
  }
}
```

L'objectif d'un filtre est de propager dans le pipe les objets répondant à certains critères de sélection et seulement ceux-ci. On doit donc s'assurer de filtrer les objets dés que possible afin d'optimiser le pipeline.

Ce filtre ne peut pas être utilisé avec le cmdlet *Where-Object* comme nous l'indique la documentation de PowerShell :

```
Get-Process | where {process-r-x}
```

Car ce cmdlet attend comme paramètre un ***scriptblock*** (ou bloc de code) ce que n'est pas un filtre ou une fonction.

Une fonction est un bloc de code nommé, à la différence d'un scriptblock qui est un bloc de code anonyme (une lambda expression d'après Bruce Payette).

Voyons cela dans le provider des fonctions qui contient également les filtres :

```
PS C:\Temp> function:
PS Function:\> dir process-r-x
```

```
CommandType     Name                Definition
-----------     ----                ----------
Filter          process-r-x         process {...
```

```
PS Function:\> $F=dir process-r-x
PS Function:\> $F.Definition
```

```
process {
 $_.processname -like "[r-x]*"
}
```

Nous verrons un peu plus loin la signification du bloc ***process.***

Notez que ce filtre peut être remplacé par une fonction sans que cela modifie le comportement. Nous verrons plus tard comment l'implémenter.

Pour finir on peut très bien utiliser un filtre pour modifier l'affichage :

```
filter Affiche
{
  If ($_.processname -like "[r-x]*")
   { write-host $_.processname –f Green }
 else
```

```
   { write-host $_.processname –f Red }
}
Get-Process | affiche
```

Attention :

Le cmdlet **Write-Host** écrit toujours sur la console à la différence du cmdlet **Write-Output** qui lui écrit sur la sortie standard qui peut donc être connectée à un pipeline.

Modifions notre filtre :

```
filter process-r-x
{
  Write-Output "Réémet le process $_"
  $_ # Ré-émission dans le pipeline de l'objet courant
}
```

Ce dernier exemple émet dans le pipeline un objet de la classe *System.Diagnostics.Process* **ET** une chaîne de caractère :

```
PS C:\Temp>  $A=Get-Process
PS C:\Temp>  $P=$A[0]|process-a-m
PS C:\Temp>  $P
```

```
Réémet le process System.Diagnostics.Process (explorer)
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    160       5     2432       4692    32     0,11   3792 explorer
```

```
PS C:\Temp> $P.count
```

```
2
```

Notez que s'il y a plus d'un objet un pipeline renvoie son résultat final dans une collection d'objet de type tableau :

```
PS C:\Temp> $P -is [Array]
```

```
True
```

Par défaut PowerShell référence l'espace de nom *System*, la syntaxe [Array] est donc un raccourci de [System.Array].

3.2 Fonction

Une fonction peut se composer, outre sa section param(), de trois blocs de code : *begin*, *process* et *end*. De plus elle a accès, comme les scripts et les scriptblocs, à la variable automatique ***$input***, qui est un énumérateur sur le ou les objets reçus dans le pipe.

- ✓ Le bloc **Begin** est exécuté une seul fois lorsque la fonction est appelée.
- ✓ Le bloc **Process** est exécuté pour chaque objet envoyé dans le pipeline. Un filtre ne possède que ce bloc.
- ✓ Le bloc **End** est appelé une seul fois lorsque la fonction se termine. C'est le bloc par défaut, c'est-à-dire que si vous ne définissez aucun de ces blocs, la fonction se comportera comme si elle possédait un seul bloc de type **End**.

L'usage de ces trois blocs permet à une fonction de se comporter comme un cmdlet (http://msdn2.microsoft.com/en-us/library/system.management.automation.cmdlet(VS.85).aspx) qui dispose des méthodes *BeginProcessing*, *ProcessRecord* et *EndProcessing*.

Note : Le cmdlet **Foreach-Object** permet également l'usage de ces blocs :

```
1..3|Foreach-Object {"Début"} {$_} {"Fin"}
```

Voir aussi :

La version 2 de PowerShell proposera la notion de ScriptCmdLet (http://community.bartdesmet.net/blogs/bart/archive/2008/03/22/windows-powershell-2-0-feature-focus-script-cmdlets.aspx) .

Webcast MSDN : Creating Windows PowerShell V2 Script Cmdlets. (http://msevents.microsoft.com/CUI/WebCastEventDetails.aspx?EventID=1032371230&EventCategory=3&culture=en-US&CountryCode=US)

3.3 A propos des itérateurs

La variable automatique ***$input*** implémente l'interface .NET *IEnumerator* (http://msdn2.microsoft.com/fr-fr/library/system.collections.ienumerator_members.aspx) qui propose les méthodes *MoveNext*, *Reset* et la propriété *Current*. Les instructions PowerShell Foreach et Switch dispose également, dans leur bloc d'exécution, de variable automatique de même type respectivement nommé **$foreach** et **$switch**.

Ces énumérateurs permettent d'accéder à l'intégralité des objets contenus dans ces variables automatiques. Notez qu'un itérateur ne possède pas de propriété indiquant le nombre d'éléments présents dans cette collection.

3.4 Visualisation de l'exécution des différents blocs

Déclarons trois fonctions identiques permettant de visualiser l'exécution des différents blocs :

```
Function Fonction1
{   #ici on initialise les variables avant la
    #mise à disposition du premier objet
  Begin { Write-Host "Begin Fonction1" -f Green }
    #Traitement effectué pour chaque objet reçu
  Process{ Write-Host "`tProcesss Fonction1" -F Yellow
    $_
  }
    #Finalisation du traitement
  End {  Write-Host "End Fonction1" -f Cyan }
}
```

Puis exécutons-les dans un pipeline :

```
1..4|fonction1|fonction2|fonction3
```

```
Begin Fonction1
Begin Fonction2
Begin Fonction3
      Processs Fonction1
      Processs Fonction2
      Processs Fonction3
1
      Processs Fonction1
      Processs Fonction2
      Processs Fonction3
2
      Processs Fonction1
      Processs Fonction2
      Processs Fonction3
3
      Processs Fonction1
      Processs Fonction2
      Processs Fonction3
4
End Fonction1
End Fonction2
End Fonction3
```

Cela nous permet de voir que les blocs *Begin* sont exécutés en premier, les uns à la suite des autres et dans l'ordre de leurs déclarations.

Ensuite les blocs *Process* sont exécutés, les uns à la suite des autres et dans l'ordre de leurs déclarations. La console étant le dernier consommateur du pipeline elle affiche l'objet sous forme de texte une fois l'exécution de la dernière fonction effectuée.

Et enfin les blocs *End* sont exécutés les uns à la suite des autres et dans l'ordre de leurs déclarations.

L'usage de segment de pipeline n'est donc pas l'équivalent d'appels de fonctions dans un langage de script conventionnel tel que :

```
Fonction1(Fonction2(Fonction3(Objet) ) )
```

Maintenant réécrivons la seconde fonction de manière «classique» :

```
Function Fonction2-1
{
    Write-host "`tDonnées dans la fonction2-1 : $input" -F Yellow
    $input
}
```

Cette réécriture ne permet plus l'accès à la variable automatique **$_**, car elle n'est plus disponible, en revanche celle nommée **$input** est accessible :

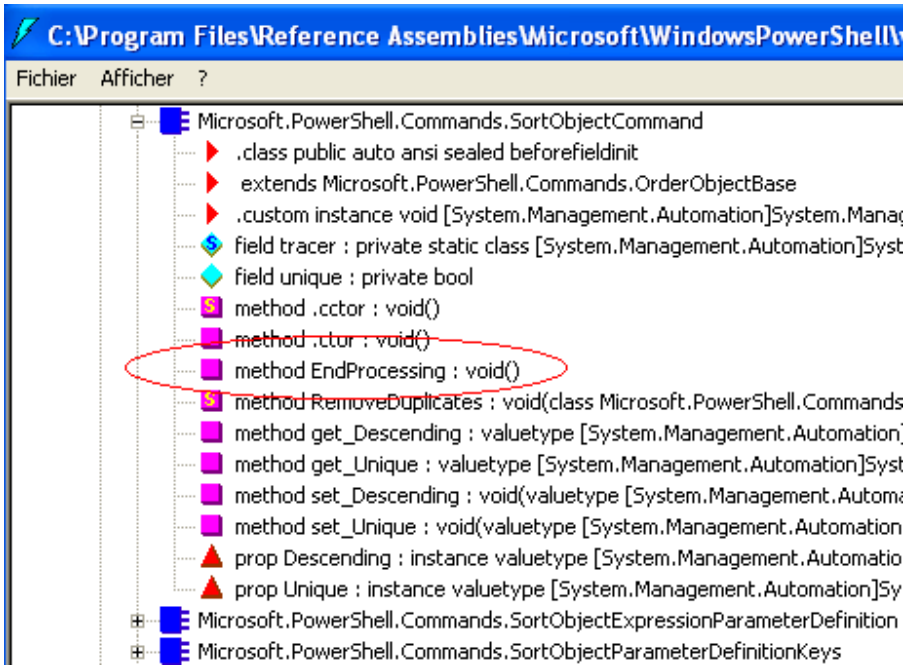
```
1..4|fonction1|fonction2-1|fonction3
```

```
Begin Fonction1
Begin Fonction3
      Processs Fonction1
```

```
      Processs Fonction1
      Processs Fonction1
      Processs Fonction1
End Fonction1
      Données dans la fonction2-1  : 1 2 3 4
End Fonction3
```

Notez que suite à ces modifications certains blocs *End* et *Process* ne sont pas exécutés tout comme le bloc *Process* de la fonction3

On voit que sans préciser aucun des blocs, une fonction implémente par défaut le bloc *End*. De plus elle affiche une seule fois une collection d'objets au lieu d'énumérer son contenu objet par objet. Cette fonction se comporte donc comme le cmdlet **Sort-Object** qui n'implémente que la méthode *EndProcessing*, comme on peut le voir sous IlDasm :



Pour le second point, l'arrêt de l'exécution du bloc *Process* de la fonction 3, cela est dû au fait que dans notre fonction nous affichons les données reçues **ce qui provoque leurs consommation par l'appel, interne au parseur de PowerShell, à l'itérateur *$input*.**

Pour corriger cela nous devons réinitialiser l'itérateur par un appel à la méthode ***Reset*** :

```
Function Fonction2-2
{  Write-host "`tDonnées dans la fonction2-2 : $input" -F Yellow
    $input.Reset()
    $input
}
```

Voyons le résultat :

```
1..4|fonction1|fonction2-2|fonction3
Begin Fonction1
```

```
Begin Fonction3
	Processs Fonction1
	Processs Fonction1
	Processs Fonction1
	Processs Fonction1
End Fonction1
	Données dans la fonction2-2 : 1 2 3 4
	Processs Fonction3
1
	Processs Fonction3
2
	Processs Fonction3
3
	Processs Fonction3
4
End Fonction3
```

Ainsi le bloc process de la fonction3 reçoit bien les données provenant de la fonction2-2.

Nous pouvons bien évidement transmettre dans le pipe d'autres types de donnée :

```
1,2,(4..5),255|Fonction1
 $H.Premier="Un"
 $H.Deuxieme="Deux"
 $H.Troisieme=@(1,2,3) # Un tableau est un objet
 $H
 $H|Fonction1 # On transmet un object contenant d'autre objet
 $H.Values| Fonction1 # On transmet plusieurs objects

"Test de chaine"|Fonction1
[char[]]"Test de chaine"|Fonction1

$Tab=@((1),(1,2),(1,2,3)) #Tableau de tableaux
$Tab| Fonction1
```

3.5 Accès aux données dans les différents blocs

Pour étudier ce sujet nous utiliserons une fonction composée des 3 blocs. Voir le script ..\Scripts\VisualisePipe.ps1. Chaque bloc affichera les mêmes informations, à savoir le type et le contenu des variables automatiques **$_** et **$input** :

```
Function VisualisePipe
{ #Affichage d'information sur les objets transmis dans un pipeline
 begin{
  Write-Warning "Begin"
  $NomType=$input.GetType().Fullname
  Write-host "Nom du type de la variable automatique `$input : $NomType)"
  Write-host "Vérification de la déclaration de la variable automatique
`$input : $(dir variable:i*)"
```

```
  if ($_ -eq $null)
   {Write-host "Pas de donnée issue du pipe " -f red }
  else {Write-host "Donnée issue du pipe : $_" -f green }

  if ($input.Movenext() -eq $false)
   {Write-host "Pas de donnée issue de l'énumérateur." -f red }
  else
   {Write-host "Donnée issue de l'énumérateur : $($input.Current)" -f
green }
      $IsPremier=$true
…
}
```

Affichons le résultat :

```
1..3| VisualisePipe
```

```
AVERTISSEMENT : Begin
Nom du type de la variable automatique $input : System.Collections.ArrayList+ArrayListEnumeratorSimple
Vérification de la déclaration de la variable automatique $input : System.Management.Automation.PSVariable
Pas de donnée issue du pipe
Pas de donnée issue de l'énumérateur.
AVERTISSEMENT : Processs
Nom du type de la variable automatique $input : System.Collections.ArrayList+ArrayListEnumeratorSimple
        Donnée issue du pipe : 1
        Type de la donnée :System.Collections.ArrayList+ArrayListEnumeratorSimple
1
        Donnée issue de l'énumérateur : 1
AVERTISSEMENT : Processs
        Donnée issue du pipe : 2
        Type de la donnée :System.Collections.ArrayList+ArrayListEnumeratorSimple
2
        Donnée issue de l'énumérateur : 2
AVERTISSEMENT : Processs
        Donnée issue du pipe : 3
        Type de la donnée :System.Collections.ArrayList+ArrayListEnumeratorSimple
3
        Donnée issue de l'énumérateur : 3
AVERTISSEMENT : End
Nom du type de la variable automatique $input : System.Array+SZArrayEnumerator
        Donnée issue du pipe : 3
3
        Pas de donnée issue de l'énumérateur.
```

On peut voir que le contenu de ces deux variables automatiques diffère selon les blocs :

- ✓ Dans le bloc **begin** aucune variable n'est disponible. C'est un bloc d'initialisation.
- ✓ Dans le bloc **process** la variable ***$input*** contient l'objet courant tout comme la variable ***$_***. Ces variables sont à ***$null*** à la fin de l'exécution de ce bloc.
- ✓ Dans le bloc **end,** et seulement si le bloc **process** est aussi déclaré, la variable ***$_*** contient le dernier objet reçu dans le pipe, objet identique au dernier reçu dans le bloc **process**. La variable ***$input*** ne contient aucune donnée. C'est un bloc de finalisation.

  Ce comportement diffère si on ne déclare aucun bloc ou seulement le bloc **end**, dans ce cas la variable ***$_*** ne contient aucune donnée et ***$input*** contient la totalité des objets envoyés dans le pipe.

Je vous laisse vérifier ces informations à l'aide des fonctions suivantes :

VisualisePipe : implémente les trois blocs Begin, Process et End.

VisualisePipe2 : implémente le bloc End uniquement.

VisualisePipe2-1 : implémente que les blocs Process et End.

VisualisePipe3 : n'implémente aucun bloc. Equivaut à la fonction VisualisePipe2.

Note :

La variable ***$input*** n'est jamais $***null*** mais peut contenir une collection vide, dans ce cas un appel à *MoveNext* ne renvoie aucune donnée. De plus l'accès à la propriété *Current* ne peut se faire qu'après un appel à *MoveNext*.

3.6 Type de donnée des variables $input et $_

Vous aurez noté que le type de l'énumérateur est différent selon le bloc dans lequel on l'utilise. Voyons le détail à l'aide des filtres puis des fonctions suivantes.

```
Filter FiltreTst {$input.GetType(); Write ("-" * 80)}
1..3|FiltreTst
```

Renvoie 3 éléments de type:

IsPublic	IsSerial	Name	BaseType
False	True	ArrayListEnumeratorSimple	System.Object

```
Filter FiltreTst {$_.GetType();Write ("-" * 80)}
1..3| FiltreTst
```

Renvoie 3 éléments de type:

IsPublic	IsSerial	Name	BaseType
False	True	Int32	System.ValueType

```
function FctTest (){$_.GetType();Write ("-" * 80)}
1..3| FctTest
```

Renvoie l'erreur:

« Vous ne pouvez pas appeler de méthode sur une expression ayant la valeur Null. »

Ici le symbole **$_** n'est pas supporté, essayons avec **$input**

```
function FctTest (){$input.GetType();Write ("-" * 80)}
1..3| FctTest
```

Renvoie 1 élément de type:

IsPublic	IsSerial	Name	BaseType
False	True	SZArrayEnumerator	System.Object

```
function FctTest {process{$_.GetType();Write ("-" * 80)}}
1..3| FctTest
```

Renvoie 3 éléments de type:

IsPublic	IsSerial	Name	BaseType
False	True	Int32	System.ValueType

Le résultat du dernier exemple, utilisant une fonction, est donc identique au filtre utilisé précédemment :

```
Filter FiltreTst {$_.GetType();Write ("-" * 80)}
```

Notes

L'ajout d'un test d'appartenance de classe ($input -is [array]) ou ($_-is [array]) renvoie toujours False.

La classe ArrayListEnumeratorSimple est interne à la BCL 2.0 (http://msdn2.microsoft.com/en-us/netframework/aa569603.aspx) et est utilisée par la classe ArrayList (cf. ArrayList.cs). Elle implémente l'interface *IEnumerator*.

4 Visualisation du flux de traitement des objets

Comme nous pouvons créer des filtres associons leurs une autre technique proposé par PowerShell, l'ajout dynamique de membres. Dans cet exemple nous ajouterons une propriété contenant l'heure, les minutes, les secondes ainsi que les millisecondes. Cet exemple nous confirmera l'intérêt et les pièges que les blocs d'une fonction peuvent apporter dans le développement de script.

Le cmdlet utilisé pour ajouter des propriétés se nomme **Add-Member :**

```
   ObjetConcerné|Add-Member TypeDeMembre NomDePropriété Contenu
```

Ce qui nous donne au sein de notre fonction, où seule la partie **process** nous intéresse :

```
function Add-TimeStamp($Nom) {
# Ajoute une propriété de nom $Nom contenant la date formatée
  process
   {
     $_|Add-Member NoteProperty $Nom ("{0:HH:mm:ss:ffff}" -f (get-date))
     $_ # On renvoie l'objet modifié
     # L'usage de Write-Host reste bien entendu possible ici
   }
}
```

Exécutons maintenant cette suite d'instructions :

```
Get-Process|Add-TimeStamp Début|select processname,Début| Sort
ProcessName|foreach {start-sleep -m 10; $_}|Add-TimeStamp Fin
```

Notre pipe récupère tous les process, ajoute à chaque objet une propriété nommée *Début*, tri les process à l'aide de la propriété ajoutée, effectue une pause de 10 millisecondes puis retransmet chaque objet reçu. On termine par l'ajout d'une seconde propriété nommée *Fin*.

Ce qui renvoie quelque chose comme ceci :

```
ProcessName                Début              Fin
```

```
-----------                    -----                  ---
explorer                       15:36:01:3281          15:36:01:5937
...
winlogon                       15:36:01:3750          15:36:02:4062
```

Vous remarquerez que l'heure de début pour le processus *Winlogon* est antérieure à l'heure de fin du processus *explorer*. Ce qui voudrait dire que le pipeline traite dans chaque segment les objets par groupe avant de les transmettre au segment suivant.

En réalité le pipeline transmet chaque objet de segment à segment sans attendre que la totalité des objects soit traités dans chaque segment. Ce qui se passe dans notre exemple précédent est que le cmdlet **Sort-Object**, situé en milieu de pipeline, doit récupérer une collection pour effectuer son traitement, il provoque donc dans le pipeline une rétention du flux.

Une meilleure approche, si toutefois votre traitement le permet, est de placer ce type de cmdlet à la fin du traitement de votre pipeline :

```
Get-Process|Add-TimeStamp Début|select processname,Début|foreach {start-
sleep -m 10; $_}|Add-TimeStamp Fin|Sort Début,Fin
```

Dans ce cas les objets sont bien émis les uns à la suite des autres :

```
ProcessName                    Début                  Fin
-----------                    -----                  ---
explorer                       16:16:08:6093          16:16:08:6250
...
winlogon                       16:16:09:3125          16:16:09:3281
```

Note :

Dans notre exemple l'ajout « à la volée » de propriétés sur des objets s'applique uniquement sur les instances concernées et pas sur toutes les instances de la classe. Pour ce faire vous devez créer un fichier de formatage `.ps1xml` (http://blogs.msdn.com/powershell/archive/2006/06/21/more-how-does-powershell-formatting-really-work.aspx)

5 Pipeline et scriptblock

Un bloc de script peut également utiliser les clauses begin, end et process :

```
$ScriptBlock = {Begin {$X=0} Process{$X+=$_} End {Write-host $X}}
```

Ici aussi les règles identiques aux fonctions s'appliquent. Pour exécuter ce scriptbloc on utilise le caractère éperluette **&**, aussi appelé *et commercial :*

```
3..5|&$ScriptBlock
```

L'utilisation des variables automatique est identique :

```
$ScriptBlock = {
     foreach ($Process in $input)
      {  $Process.ProcessName }
}
```

L'appel de ***&$ScriptBlock,*** tout comme l'appel à la méthode *Invoke()*, se fait dans une nouvelle portée (scope) à la différence de l'appel ***.$ScriptBlock***

L'appel peut se faire sans la déclaration de variable intermédiaire :

```
Get-Process | &{ begin{"*Debut*"} process{$_.ProcessName} end{"*Fin*"} }
```

Un scriptblock peut déclarer et utiliser des paramètres :

```
$ScriptBlock = {
 param([Int] $Increment)
  Begin {$X=0}
  Process{$X+=$_+$Increment}
  End {Write-host $X}
}
```

Et enfin sachez qu'un scriptbloc peut être utilisé comme paramètre d'une fonction, d'un cmdlet, d'un filtre ou d'un scriptbloc.

5.1 Exemple

Cet exemple a été proposé sur un forum US par Keith Hill coordinateur du projet open-source PowerShell Community Extensions (http://www.codeplex.com/PowerShellCX), projet proposant de nombreux cmdlets.

```
filter Invoke-NullAlt ([scriptblock] $expression, [scriptblock]
$executeIfNull, [bool] $debug = $false)
{
   if ($debug) {"`$expression is `"$expression`""}
   if ($expression -ne $null)
    {
      $result = &$expression
      if ($debug) {"`$exprResult is `"$result`""}
      if ($result -ne $null)
       { $result }
      else
      { &$executeIfNull }
    }
   else
     { &$executeIfNull}
}
```

On déclare un alias pour ce filtre

```
set-alias ?? Invoke-NullAlt
```

Enfin l'appel du code utilisant des scriptblock en paramètre

```
get-process | ?? {$_.Description} {"* Description inconnue *"}
```

Ce filtre affichera la valeur de la propriété *description* de chaque processus mais si sa valeur est ***$null*** il affichera « * Description inconnue * ».

6 A suivre

Nous avons pu voir que le pipelining est bien une des pièces maîtresse de ce nouveau shell de Microsoft et qu'il influencera notre manière d'écrire des scripts sous PowerShell.

Faute de temps je n'ai pas abordé les principes du binding de paramètres ni la propagation et la gestion des exceptions.

Pour ce dernier point les exceptions non-critiques provoqueront, par défaut, un affichage sur la console quant aux exceptions critique elles annuleront le pipeline en cours en appelant auparavant les blocs **End** déclarés.

Un pipeline spécial est disponible pour remonter les erreurs qui peuvent être bloquantes ou pas, dans ce dernier cas le traitement se poursuit après l'affichage d'un message d'erreur. Le Cmdlet **Write-Error** écrit sur ce pipeline d'erreur.

7 Liens

De l'usage du pipeline pour optimiser les performances de PowerShell

http://janel.spaces.live.com/blog/cns!9B5AA3F6FA0088C2!304.entry

Optimizing Performance of Get-Content for Large Files

http://keithhill.spaces.live.com/blog/cns!5A8D2641E0963A97!756.entry

1-Flexible pipelining with ScriptBlock Parameters

http://blogs.msdn.com/powershell/archive/2006/06/23/643674.aspx

2-$MindWarpingPower = $Cmdlets + $ScriptBlock_Parameters

http://blogs.msdn.com/powershell/archive/2008/04/21/mindwarpingpower-cmdlets-scriptblock-parameters.aspx

Partying with Join-Path

http://blogs.msdn.com/powershell/archive/2007/06/29/partying-with-join-path.aspx

Les pipelines en C#

Les itérateurs de C#2

http://www.dotnetguru.org/articles/dossiers/iteratorcsharpv2/CS2_Iterators_FR.htm#_Toc84480268

Pipelines Using Iterators, Lambda Expressions and Extension Methods in C# 3.0

http://www.clariusconsulting.net/blogs/kzu/archive/2008/01/20/PipelinesUsingIteratorsLambdaExpressionsExtensionMethods.aspx

Creating a Console Application that Adds Commands to a Pipeline

http://msdn2.microsoft.com/en-us/library/ms714593.aspx