Gestion des path sous PowerShell.

Par Laurent Dardenne, le 28/10/2014.

Niveau

Conçu avec Powershell v3 sous Windows Seven.

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

Les fichiers sources :
http://ottomatt.pagesperso-orange.fr/Data/Tutoriaux/Powershell/GestionDesPathSousPowershell/Sources.zip

Chapitres

1 Une évidence

Par habitude, l'usage du terme lecteur nous fait référencer ceux associés au système de fichiers. Que ce soit dans un shell ou une application, la source de données importe peu puisque nous n'en avons qu'une à notre disposition proposant des lecteurs. Ici un lecteur sera toujours une unité de disque Windows, son support, son format et son emplacement physique n'y changera rien, car les mécanismes de redirection sont pris en charge par le système.

Par exemple la notion de chemin (path) associée à celle de lecteur, référence toujours le système de fichiers. La gestion d'erreur doit prendre en compte celles récurrentes comme les problèmes de droits d'accès, de noms de chemin invalide, des possibilités du support physique, difficile d'écrire sur un CDROM, voire des erreurs d'écriture dues à un support défectueux ou indisponible.

Si ce n'est qu'une évidence, pourquoi donc la rappeler ? Vous pourrez constater à la lecture de ce tutoriel que cette évidence est trompeuse sous Powershell et qu'il est préférable d'étudier de plus près ses notions de lecteur et de chemins revisitées.

2 Principe

PowerShell nous offre la possibilité d'employer un ensemble cohérent de cmdlets pour parcourir et naviguer dans des sources de données hiérarchisée ou non, le système de fichier, la base de registre, Active Directory ou toutes autres sources de données disposant d'un provider :

http://laurent-dardenne.developpez.com/articles/Windows/PowerShell/Introduction/#L3-3

Sous Powershell l'accès aux données d'un provider nécessite la création d'un PSDrive, un lecteur Powershell.

Un PSDrive référence un emplacement d'une source de données, ce peut être la racine d'une arborescence, par exemple le PsDrive ***C:***\ du FileSystem ou ***Variable:***\ du provider de variable Powershell.

Pour le système de fichier il est possible de créer des noms de lecteur en dehors de l'étendue de lettres ***A-Z***, par exemple 'Psionic' :

```
cd psionic:\
PsIonic:\
PsIonic:\> Dir

    Répertoire : G:\PS\PsIonic

Mode      LastWriteTime         Length Name
----      -------------         ------ ----
d----      04/05/2014   17:12   <DIR>  Bin
```

L'affichage renvoie bien vers un disque logique du FileSystem.

Ceci permet par exemple de référencer indirectement l'emplacement d'un répertoire projet sans le coder en dur dans un script :

```
#<INCLUDE %'PsIonic:\Tools\New-PSPathInfo.ps1'%>
```

Sous réserve de déclarer ce lecteur dans le profile Powershell :

```
 #PSDrive sur le répertoire du projet
$null=New-PsDrive -Name PsIonic -Root 'G:\PS\PsIonic' -PSProvider
FileSystem -Scope Global
```

Si on souhaite que le lecteur reste accessible une fois le script terminé, on utilisera la portée globale.

Note : Cette possibilité existait déjà sous MS-DOS à l'aide de la commande Subst.

3 La notion de chemin sous Powershell

Dans le fichier *about_Path_Syntax.txt* de la documentation de Powershell, le terme conteneur est utilisé afin de généraliser le concept porté par le mot de répertoire ou de directory qui sont liés au système de fichiers.

« Tous les éléments peuvent être identifiés de façon unique par leur nom de chemin d'accès.

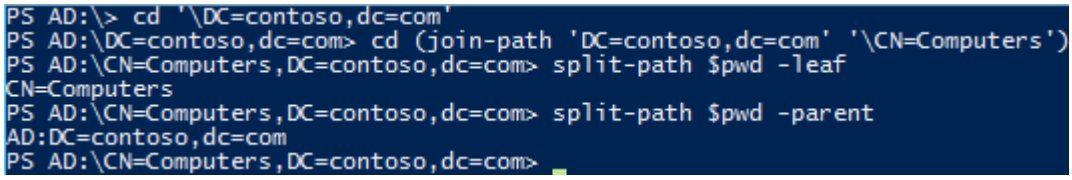
Un nom de chemin d'accès est une combinaison du nom d'élément, du conteneur et des sous-conteneurs qui constituent l'emplacement de l'élément, et du lecteur Windows PowerShell par lequel s'effectue l'accès aux conteneurs.

Dans Windows PowerShell, les noms de chemin d'accès sont de deux types : complet et relatif. Un nom de chemin d'accès complet est constitué de tous les éléments qui composent un chemin d'accès. La syntaxe suivante présente les éléments d'un nom de chemin d'accès complet :*

[<provider>::]<drive>:[\<container>[\<subcontainer>...]]\<item>

L'espace réservé <provider> fait référence au fournisseur Windows PowerShell à travers lequel vous accédez au magasin de données. »

***** Le provider fournit les séparateurs de chemin d'accès, ici aussi l'habitude nous fait considérer la barre oblique comme le séparateur par défaut. Un exemple avec le provider ***ActiveDirectory*** :



Note : Une URL sera considérée comme un nom de chemin :

```
Split-Path -Path 'http://blogs.msdn.com/b/Powershell' -Leaf
```

```
Powershell
```

```
Split-Path -Path 'http://blogs.msdn.com/b/Powershell' -Qualifier
```

```
http:
```

Comme il n'existe pas de provider '*WEB*', certaines opérations échoueront :

```
Split-Path -Path 'http://blogs.msdn.com/b/Powershell' -Resolve
```

```
Split-Path : Lecteur introuvable. Il n'existe aucun lecteur nommé « http ».
```

3.1 Exemples de noms de chemins d'accès relatifs et complets

Certains providers autorisant le déplacement dans les conteneurs, Powershell reprend la notion de chemin relatif. Ces exemples supposent que le répertoire de travail actif est *C:\Windows*.

Symbole	*Description*	*Chemin d'accès relatif*	*Chemin d'accès complet*
.	Indique l'emplacement de travail courant	.\System	C:\Windows\System
..	Indique le parent de l'emplacement courant	..\Program Files	C:\Program Files
\	Indique la racine du lecteur courant	\Program Files	C:\Program Files
[aucun]	Aucun caractère spécial. Référence le conteneur courant.	System	C:\Windows\System
~	Indique le home directory du provider courant.	Cd C:\ Cd ~	Chemin contenu dans : *(Get-PSProvider FileSystem).Home*

Pour le dernier symbole, chaque provider utilise sa propriété *Home*, mais tous ne le configurent pas lors du chargement :

```
Get-PsProvider|Select Name,Home
```

```
Name                 Home
----                 ----
Alias
Environment
FileSystem           C:\Users\Laurent
Function
Registry
…
```

Pour le FileSystem le contenu de *Home* est égale à *$Env:HOMEPATH*.

L'utilisation du symbole ~ peut donc déclencher une exception sur certains providers :

```
cd hklm:
cd ~
```

```
Set-Location : L'emplacement d'origine de ce fournisseur n'est pas défini.
```

L'affectation d'une valeur par défaut pour la propriété *Home* peut se faire ainsi :

```
(Get-PsProvider 'Registry').Home ='HKLM:\SOFTWARE'
cd ~
```

```
HKLM:\SOFTWARE
```

4 Les types de chemin

Les classes dotNet ou les APIs win32 ne connaissent que des noms de chemin 'classiques', les type de chemins de Powershell ne sont pas supportés, hormis ceux compatibles avec le FileSystem.

Par exemple l'appel suivant nous indique que le fichier n'existe pas :

```
[IO.File]::ReadAllBytes('C:\temp\notexist')
```

```
ReadAllBytes : Exception lors de l'appel de « ReadAllBytes » avec « 1 » argument(s) : « Impossible de trouver le fichier 'C:\temp\notexist'. »
```

Le chemin est considéré comme syntaxiquement valide, l'exception déclenchée étant ***System.IO.FileNotFoundException.***

Par contre celui-ci déclenche une exception ***System.NotSupportedException***:

```
[IO.File]::ReadAllBytes('MyDrive:\temp\notexist')
```

```
ReadAllBytes : Exception lors de l'appel de « ReadAllBytes » avec « 1 » argument(s) : « Le format du chemin d'accès donné n'est pas pris en charge. »
```

A la différence des classes dotNet et des APIs win32, PowerShell propose pour tous les providers d'autres constructions de nom de chemin.

4.1 Drive-Qualified Path (Chemin qualifié par un lecteur)

DriveName:\Container\Subcontainers\Item

Un chemin qualifié par un lecteur est une combinaison du nom du lecteur PowerShell, du nom du conteneur et/ou la liste de sous-conteneur et du nom d'élément (un item).

Un lecteur cible un fournisseur afin d'accéder à une source de données. La documentation parle de magasin de données (data store) alors que la notion de provider souhaite justement s'affranchir du matériel afin de se concentrer sur les données. Powershell accède à tous lieux de stockage, chacun ayant sa propre méthode de rangement, laquelle ne nous intéresse pas. Seul le provider connait cette méthode. En revanche la structure des données manipulées nous intéresse.

Exemples :

```
C:\Windows
Cert:\LocalMachine
AD:\CN=Computers,DC=Contoso,DC=Com
IIS:\sites\Default Web Site
```

4.2 Provider-Qualified Path (Chemin qualifié par un provider)

ProviderName::Container\Subcontainers\Item

ModuleName\ProviderName::\Container\Subcontainers\Item

Un chemin qualifié par un nom de provider accède directement à une ressource sans utiliser la notion de lecteur, on qualifie le chemin à l'aide du nom du provider, suivi de deux point ' ::' et du chemin d'accès.

Par exemple, on peut référencer une ruche en construisant le chemin suivant :

```
Registry::\HKEY_CLASSES_ROOT
```

En cas de conflit sur un nom de provider, il est possible de préfixer le nom de chemin par le nom du module hébergeant le provider :

```
Microsoft.PowerShell.Core\Registry::\HKEY_CLASSES_ROOT
```

L'usage de ce type de chemin pour un accès relatif est possible avec le provider FileSystem, car celui-ci reconnait en interne les entrées '**.**' et '**..**' :

```
Get-ChildItem FileSystem::C:\Windows\..
```

```
    Répertoire : Microsoft.PowerShell.Core\FileSystem::C:\
…
```

Mais ces entrées ne seront pas reconnues avec d'autres :

```
Get-ChildItem Registry::HKEY_LOCAL_MACHINE\software\Classes\..
```

```
dir : Impossible de trouver le chemin d'accès « HKEY_LOCAL_MACHINE\software\Classes\.. », car il n'existe pas.
```

La localisation peut toutefois utiliser ces noms :

```
HKLM:\SOFTWARE\Classes\> cd hklm:\SOFTWARE\Classes\
```

```
HKLM:\SOFTWARE\Classes\
```

```
HKLM:\SOFTWARE\Classes\> set-location ..

HKLM:\SOFTWARE>
```

4.2.1 Notes

Pour ce type de chemin, l'exemple précédent (*FileSystem::C:\Windows\..*) semble contradictoire avec la définition qui en est donnée.

Dans une nouvelle session exécutez les commandes suivantes :

```
cd Alias:

Alias:\

Get-PSDrive|Remove-PSDrive
```

```
Remove-PSDrive : Impossible de supprimer le lecteur « Alias », car il est en cours d'utilisation.
```

```
Get-PSDrive
```

```
Name           Used (GB)     Free (GB) Provider      Root            CurrentLocation
----           ---------     --------- --------      ----            ---------------
Alias                                  Alias
```

```
Dir FileSystem::C:\Windows
```

```
    Répertoire : Microsoft.PowerShell.Core\FileSystem::C:\Windows
..
```

Une fois tous les lecteurs Powershell supprimés, le chemin qualifié par le provider pointe toujours sur les données souhaitées. La confusion vient du fait que le lecteur 'C' sous Powershell est redéclaré lors du démarrage du provider FileSystem, l'habitude nous fait penser qu'il s'agit d'une seule et même chose.

Il vous faut donc comprendre qu'ici le lecteur Windows C: référence un chemin d'accès existant en dehors de la session Powershell, tout comme la ruche de l'utilisateur nommée *HKEY_CURRENT_USER* :

```
dir Registry::\HKEY_CURRENT_USER\Software
```

```
Hive: \HKEY_CURRENT_USER\Software
…
```

Ce type de chemin pointe directement sur un élément ou un conteneur d'une source de données sans nécessiter la présence d'un lecteur Powershell. Donc, hormis pour le provider FileSystem, la présence d'un nom de lecteur dans ce type de chemin provoquera une erreur :

```
Dir registry::HKCU:\Software
```

```
Dir : Impossible de trouver le chemin d'accès « HKCU:\Software  », car il n'existe pas.
```

Ici le provider de registry cherche à accéder à une donnée dans la ruche '**HKCU:**' qui, comme nous le précise Powershell, n'existe pas.

Pour clore cette note, la différence entre les deux chemins inexistants suivants :

```
Dir FileSystem::Z:\Windows
```

```
Dir : Impossible de trouver le chemin d'accès « Z:\Windows », car il n'existe pas.
```

```
Dir Z:\Fichiers
```

```
Dir : Lecteur introuvable. Il n'existe aucun lecteur nommé « Z ».
```

est que le message d'erreur référence pour le premier un chemin d'accès direct d'un élément et pour le second un chemin d'accès *via* un lecteur.

Les exceptions déclenchées sont respectivement **ItemNotFoundException** et **DriveNotFoundException.** Dans le premier cas je peux supposer, à tort, que si l'item n'existe pas je peux le créer alors que dans le second ce ne sera pas possible sans le lecteur.

Une subtilité sémantique à connaitre.

4.3 Provider-Direct Path (chemin direct de provider)

ProviderName::\ServerName\Path\Item

ModuleName\ProviderName::\ServerName\Path\Item

Permet à un provider PowerShell l'accès à une ressource distante, le nom du chemin est passé directement au provider. Celui-ci doit intégrer ce type d'accès, par exemple un chemin UNC :

```
Cd C:\
Dir \\Localhost\C$
Dir \\Localhost\PublicShareName\Path
```

Par contre ce type de chemin ne sera pas supporté par d'autre provider :

```
Cd Variable:\
Dir \\Localhost\C$
```

Dir : Impossible de trouver le chemin d'accès « \\Localhost\C$ », car il n'existe pas.

Pour régler ce problème il suffit de préfixer le chemin avec le nom du provider ciblé :

```
Cd Variable:\
Dir FileSystem::\\Localhost\C$
```

Répertoire : Microsoft.PowerShell.Core\FileSystem::\\Localhost\C$
…

4.4 Provider-Internal Path (chemin interne de provider)

Il s'agit du chemin indiqué après les deux point "::" dans un nom de chemin qualifié provider. Seul le provider ciblé sait l'interpréter.

Par exemple pour :

```
FileSystem::C:\Windows
```

le chemin d'accès interne du provider est :

```
C:\Windows
```

Pour :

```
Registry::HKEY_CURRENT_USER\Software
```

le chemin d'accès interne du provider est :

```
HKEY_CURRENT_USER\Software
```

Note :

Windows PowerShell Provider Capabilities
http://msdn.microsoft.com/en-us/library/ee126189%28v=vs.85%29.aspx

Designing Your Windows PowerShell Provider
http://msdn.microsoft.com/en-us/library/ee126198%28v=vs.85%29.aspx

5 Autres points à considérer

5.1 Les caractères génériques

Appelé Jokers et quelques fois ‘globbing’, tous les providers n’implémentent pas cette capacité.

Wildcard	Description	Exemple	Correspond	Ne correspond pas
*	Fait correspondre **zéro, un ou plusieurs** caractères.	a*	A, ag, aigle	banane
?	Fait correspondre **exactement un** caractère à la position spécifiée.	?n	an, en, un	ban
[-]	Fait correspondre **une plage de** caractères.	[a-p]oupe	coupe, loupe, poupe	soupe
[ ]	Fait correspondre **les** caractères spécifiés.	[cl]oupe	coupe, loupe	hook

Notez que le paramètre *–Filter* présent sur certains cmdlets utilise une syntaxe spécifique au provider ciblé.

5.1.1 Méthodes liées

Pour manipuler ces jokers Powershell propose la classe **Management.Automation.WildcardPattern**, notamment les méthodes statiques *ContainsWildcardCharacters*, *Escape* et *UnEscape*.

```
[Management.Automation.WildcardPattern]::Escape('Frm[AZ]')
```

```
Frm`[AZ`]
```

```
[Management.Automation.WildcardPattern]::UnEscape('Frm`[AZ`]')
```

```
Frm[AZ]
```

```
[Management.Automation.WildcardPattern]::ContainsWildcardCharacters('Frm`[AZ`]')
```

```
False
```

```
[Management.Automation.WildcardPattern]::ContainsWildcardCharacters('Frm[AZ]')
```

```
True
```

```
[Management.Automation.WildcardPattern]::ContainsWildcardCharacters('Frm[AZ')
```

```
True
```

Note : dans le dernier exemple la présence d’un seul caractère ‘**[**‘ ou ‘**]**‘ renvoie le résultat *$true*.

Voir aussi : http://www.powershellmagazine.com/2012/10/09/manipulating-wildcards/

5.2 Différence entre Path et LiteralPath

L'implémentation des caractères génériques précédents à un effet de bord sur la recherche d'item.

Par exemple l'usage d'un nom de fichier en paramètre d'un cmdlet tel que *Fichier[1].txt* fait que le parseur Powershell déclenche une recherche 'élargie', ceci est dû à la présence des caractères **[** et **]** :

```
fsutil.exe file createnew C:\Temp\Fichier[1].txt 20
```

```
Le fichier C:\Temp\Fichier[1].txt est créé
```

```
Dir C:\Temp\Fichier[1].txt
```

La commande ***Dir*** ne renvoie aucun fichier, même pas celui que l'on vient de créer.

Constatons le déclenchement du globbing :

```
fsutil.exe file createnew C:\Temp\Fichier1.txt 20
```

```
Le fichier C:\Temp\Fichier1.txt est créé
```

```
Dir C:\Temp\Fichier[1].txt
```

```
    Répertoire : C:\temp

Mode        LastWriteTime         Length Name
----        -------------         ------ ----
-a---       07/10/2014    12:41   20     Fichier1.txt
```

Pour éviter ce comportement on peut soit échapper les caractères générique :

```
dir -path 'C:\Temp\Fichier``[1``].txt'
```

```
Fichier[1].txt
```

Dans la première édition de son livre 'Powershell in Action', Bruce Payette indique que le premier caractère d'échappement est éliminé par l'interpréteur et le second par le provider.

Attention toutefois à l'usage de guillemets double qui modifient l'analyse :

```
dir -path "C:\Temp\Fichier``[1``].txt"
```

```
dir : Impossible de trouver le chemin d'accès « C:\Temp\Fichier`[1`].txt », car il n'existe pas.
```

```
dir -path "C:\Temp\Fichier````[1````].txt"
```

```
Fichier[1].txt
```

Soit, si le cmdlet le propose, passer la valeur du nom de chemin *via* le paramètre *–LiteralPath* :

```
dir -LiteralPath 'C:\Temp\Fichier[1].txt'
```

```
Fichier[1].txt
```

L'usage de ce paramètre indique au code du cmdlet d'utiliser le nom de chemin de manière littéral, il n'y a aucune interprétation des caractères génériques, là où par convention le paramètre *-Path* les interprète.

Ce qui fait qu'une recherche de fichiers sur ce nom de répertoire :

```
'C:\temp\frm[az]\*'
```

ne peut utiliser -Literalpath et nécessite d'échapper les caractères '[' et ']', mais pas les caractères '*' et '?'.

5.2.1 Implémenter la gestion des paramètres Path et LiteralPath

Voici une solution basée sur une fonction avancée :

```
Function Test {
 [CmdletBinding(DefaultParameterSetName="Path")]
 param(
    [parameter(Mandatory=$True,ValueFromPipeline=$True,
ParameterSetName="Path")]
   [string]$Path,
     [parameter(Mandatory=$True,ValueFromPipeline=$True,
ParameterSetName="LiteralPath")]
   [string]$LiteralPath,
   $ParamNull='Valeur par défaut'
  )
Begin {   Write-Warning "Begin ParamNull=$ParamNull"}
Process {
   $isLiteral = $PsCmdlet.ParameterSetName -eq 'LiteralPath'
   Write-Warning "Literal Path ? $isLiteral"
   if ($isLiteral)
   {
     $EscLP =[Management.Automation.WildcardPattern]::Escape($LiteralPath)
     $PsCmdlet.InvokeProvider.Item.Exists($EscLP,$false,$false)
   }
   else
   { $PsCmdlet.InvokeProvider.Item.Exists($Path,$false,$false) }
  Write-Warning "`tProcess ParamNull=$ParamNull"
 }
}
New-Item -ItemType Directory -Path 'C:\Temp\Test[' -ea SilentlyContinue
```

Notez que l'usage de l'API du provider, à savoir la méthode *Exists*, ne nécessite qu'un caractère d'échappement.

L'implémentation du paramètre *LiteralPath* permet à la fonction d'utiliser des noms de fichiers contenant des caractères génériques utilisés par PowerShell :

```
Test-Path -LiteralPath 'C:\Temp\Test['
```

```
True
```

```
Test-Path 'C:\Temp\Test['
```

```
Test-Path : Impossible de récupérer les paramètres dynamiques pour l'applet de commande.
Les caractères génériques spécifiés ne sont pas valides : Test[
```

Le code en lui-même est compréhensible et son usage aisé :

```
$ParamNull="Scope de l'appelant"
 #---Liaison par la ligne de commande
Test -LiteralPath 'C:\Temp\Test['
```

```
Begin ParamNull=Valeur par défaut
Test -Path C:\Temp
Begin ParamNull=Valeur par défaut
```

Le seul problème, mais pas des moindres et que ce type de construction sous Powershell V2 est buguée. L'usage du pipeline met en évidence ce bug :

```
  #---Liaison par le pipeline
'C:\Temp\Test['|Test -LiteralPath {$_}
Begin ParamNull=Valeur par défaut
'C:\Temp'|Test
Begin ParamNull=Scope de l'appelant
```

Pour le contourner on doit redéclarer la gestion de la valeur par défaut dans le bloc ***begin*** :

```
begin {
  if (-not $PSBoundParameters.ContainsKey('ParamNull'))
  { $ParamNull='Valeur par défaut'} #bug v2 : parameters-initialization
  Write-Warning "Begin ParamNull=$ParamNull"
}
```

Ce bug est corrigé sous PS v3.

5.2.2 Propager l'usage de LiteralPath

Dans le cas où un cmdlet utilise le paramètre *-LiteralPath* dans un pipeline, on doit le propager au cmdlet suivant.

Prenons le cas suivant :

```
"Fichier A.txt"|Set-content -path c:\temp\A.txt
"Fichier [A].txt"|Set-content -LiteralPath c:\temp\[A].txt
```

La commande suivante ne pointe pas sur le bon chemin :

```
C:\temp> Resolve-Path -LiteralPath [a].txt | Get-Content
Fichier A.txt
```

En traçant la liaison de paramètre du cmdlet Get-Content :

```
Trace-command parameterbinding {Resolve-Path -LiteralPath [a].txt | Get-
Content -LiteralPath {$_}} -PSHost
```

On constate que c'est le paramètre ***-Path*** qui est renseigné, dans ce cas le cmdlet interprète le nom reçu comme un nom de fichier contenant des jokers et renvoie le fichier 'A.txt' correspondant à la recherche '[A].txt'.

Ceci peut être vérifié en utilisant une ou toutes les sources de traces suivantes :

```
Trace-Command PathResolution,FileSystemProvider,LocationGlobber {Resolve-
Path -LiteralPath [a].txt | Get-Content} -PSHost
```

Pour propager le nom littéral on doit utiliser la construction suivante :

```
C:\temp> Resolve-Path -LiteralPath [a].txt | Get-Content -LiteralPath {$_}
Fichier [A].txt
```

5.2.3 Limite des jeux de paramètres

Attention, le fait de déclarer les deux jeux de paramètres '*Path'* et '*Literalpath'*, rend difficile l'ajout d'autres jeux de paramètre incluant l'usage des paramètres –Path ou –LiteralPath.

Car dans ce cas la règle d'unicité d'un jeu de paramètre se trouve brisée. Une solution est de coder les exclusions dans le corps du script/fonction.

Voir aussi :

http://www.vistax64.com/powershell/16338-parametersetname-how-handle-multiple-mutual-exclusions.html

5.3 Les caractères autorisés dans les noms d'item

Selon le provider, le nommage des items peut varier. Par exemple celui de la Registry autorise le caractère étoile **'*' :**

```
Get-ChildItem -LiteralPath  HKLM:\SOFTWARE\Classes\*
```

```
   Hive: HKEY_LOCAL_MACHINE\SOFTWARE\Classes\*

Name                           Property
----                           --------
OpenWithList
…
```

Alors que son usage sur le FileSystem déclenchera une exception :

```
Get-ChildItem -LiteralPath  C:\Windows\*
```

```
Get-ChildItem : Caractères non conformes dans le chemin d'accès.
```

La validité d'un nom d'item dépend donc du provider ciblé.

5.4 Noms de fichier réservés

Les noms suivants sont réservés :

CON	Keyboard and display
PRN	System list device, usually a parallel port
AUX	Auxiliary device, usually a serial port
CLOCK$	System real-time clock
NUL	Bit-bucket device
LPT1	First parallel printer port
LPT2 à LPT9	…
LPT3	Third parallel printer port
COM1	First serial communications port
COM2 à COM9	..

Leur usage déclenchera une exception :

```
C:\> "Test"|Set-Content Aux
```

```
set-content : FileStream n'ouvre pas les périphériques Win32, tels que des partitions de disque et des lecteurs de bande. Évitez d'utiliser "\\.\" dans le chemin.
```

Les caractères interdits dans un nom de fichier sont également gérés :

```
"Test"| Set-Content -path 'C:\Temp>Test.txt'
```

```
Set-Content : Caractères non conformes dans le chemin d'accès.
```

5.5 Couplage de cmdlet avec un provider

Certains cmdlets peuvent nécessiter que la localisation courante soit sur un provider particulier, par exemple ***Out-File*** référence en interne le FileSystem.

L'exemple suivant utilisant implicitement le répertoire courant provoque une exception :

```
cd hklm:
"Test"| Out-File Log.Txt
```

```
Out-File : Impossible d'ouvrir le fichier, car le fournisseur actuel (Microsoft.PowerShell.Core\Registry) ne
parvient pas à ouvrir un fichier.
```

5.6 Get-PsProvider

Ce cmdlet affiche la liste des providers chargés, sachez que certains providers tels que *Certificate* et *WSMan* seront chargés seulement lors de leur premier accès. Par exemple un appel à **Get-PsDrive** chargera ces deux providers, car il déclare un lecteur par défaut.

5.7 La localisation courante

Selon que l'on utilise un cmdlet ou une API dotNet, la localisation courante diffère. Bien que chaque lecteur Powershell mémorise sa localisation, le chemin courant pour les APIs est mémorisé dans le membre statique suivant :

```
[environment]::CurrentDirectory
```

L'article suivant détaille ce point :

The Difference between your Current Directory and your Current Location

Sachez également que l'usage la syntaxe du chemin suivant :

```
C:\Temp> cd G:\
G:\ > Dir 'FileSystem::\Temp'
```

```
   Répertoire : C:\Temp
…
```

référence *[environment]::CurrentDirectory* et pas le répertoire courant du lecteur G.

Dans une fonction avancée une propriété de la variable automatique *$PSCmdlet* renvoie la localisation courante d'un provider :

```
$PSCmdlet.CurrentProviderLocation("FileSystem")
```

```
Path
----
G:\
```

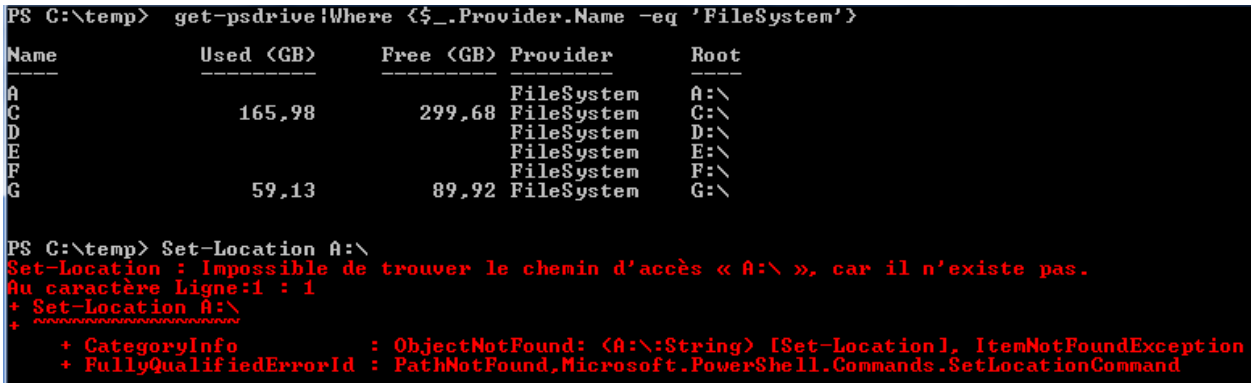
La localisation du script courant est une autre information portée par la variable ***$PSScriptRoot.***

Voir aussi : http://www.leeholmes.com/blog/2006/07/18/set-location-and-environmentcurrentdirectory/

Notez que la modification de SetCurrentDirectory dans un thread affecte les autres threads du processus.

5.8 FileSystem et Bios

Powershell peut mapper un lecteur amovible déclaré dans le BIOS bien que physiquement absent, le lecteur A:\ ci-dessous concerne un lecteur de disquette :



L'accès à un lecteur de CD ne contenant pas de disque, déclenchera la même erreur. Si dans le BIOS on désactive le lecteur A:, l'exception déclenchée sera **DriveNotFoundException.**

Notez que les fonctions nommées A: à Z: sont créées indépendamment de l'existence du lecteur.

5.9 Chemin d'accès trop long

A ce jour, le Framework dotNet ne supporte pas les noms de chemin ayant plus de 248 caractères, dans ce cas Powershell déclenchera une exception *PathTooLongException.*

Vous pouvez créer une arborescence de test à l'aide du script *New-LongPathTest.ps1*.

```
$path='C:\Temp\001\002\003\004\005\006\007\008\009\010\011\012\013\014\015
\016\017\018\019\020\021\022\023\024\025\026\027\028\029\030\031\032\033\0
34\035\036\037\038\039\040\041\042\043\044\045\046\047\048\049\050\051\052
\053\054\055\056\057\058\059\060\061\062\063'

$pwd.Path.Length
```

```
259
dir "$path"
```

```
dir : Le chemin d'accès spécifié, le nom de fichier ou les deux sont trop longs. Le nom de fichier qualifié
complet doit comprendre moins de 260 caractères et le nom du répertoire moins de 248 caractères.
    + CategoryInfo  : ReadError: (C:\Temp\001\002..\062\063:String) [Get-ChildItem], PathTooLongException
    + FullyQualifiedErrorId : DirIOError,Microsoft.PowerShell.Commands.GetChildItemCommand
```

Bien que le répertoire existe, ce comportement est correct :

```
Get-Item $path #ou Dir "$path\.."
```

```
Mode     LastWriteTime       Length  Name
----     -------------       ------  ----
d----    15/10/2014  12:36           <DIR> 063
```

L'ajout du caractère '\' déclenche une exception :

```
Get-Item "$path\" #ou "$path\064"
```

```
Get-Item : Impossible de trouver le chemin d'accès «C:\Temp\001\002\...\062\063\», car il n'existe pas.
    + CategoryInfo  :  ObjectNotFound: (C:\Temp\001..\063\:String) [Get-Item], ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetItemCommand
```

Le problème est que l'exception déclenchée diffère, ici le cmdlet ***Get-Item*** renvoie l'exception *ItemNotFoundException.*

Le constructeur de la classe suivante, qui peut être la correspondance de Get-Item, renvoie l'exception *PathTooLongException :*

```
$d=new-object System.IO.FileInfo "$path\064"
```

```
new-object : Exception lors de l'appel de «.ctor» avec «1» argument(s): «Le chemin d'accès spécifié, le nom de fichier ou les deux sont trop longs. Le nom de fichier qualifié complet doit comprendre moins de 260 caractères et le nom du répertoire moins de 248 caractères.»
```

```
$Error[0].Exception.InnerException.GetType().Fullname
```

```
System.IO.PathTooLongException
```

5.9.1 Une possible résolution

Les APIs win32 proposent des méthodes de manipulation de nom de chemin d'une longueur maximum de 32767 caractères. Ces noms de chemin doivent débuter par la séquence '\\?\'.

Il existe différentes librairie permettant d'utiliser ces APIs sous dotNet, par exemple https://alphafs.codeplex.com/ .Vous trouverez dans le répertoire *Source* un exemple d'utilisation de celle-ci.

Enfin sachez que les cmdlets Powershell ne pourront être utilisés avec les objets fichiers manipulés par ces librairies, toutes les opérations de manipulation de fichier devront se faire via les classes et méthodes de ces librairies. Par exemple on ne peut récupérer un objet fichier ayant un nom long et le passer au cmdlet Get-ACL, cela ne fonctionnera pas.

Voir aussi :

http://blogs.msdn.com/b/bclteam/archive/2007/02/13/long-paths-in-net-part-1-of-3-kim-hamilton.aspx

5.9.2 Nom court

Certains cmdlets ou APIs Powershell ne transforme pas un nom court en un nom long :

```
$sp='C:\PROGRA~1\MIA713~1'
Resolve-Path $sp\*
```

```
Path
----
C:\PROGRA~1\MIA713~1\Windows Azure
```

```
(Get-Item C:\PROGRA~1\MIA713~1).FullName
```

```
C:\Program Files\Microsoft SDKs
```

En dehors du fait qu'à la lecture du contenu de ***$SP*** on ne sache pas de quel répertoire il s'agit, les noms courts ne posent pas de problème.

5.10 Quelques bugs et comportement particuliers

Les chemins suivants sont gérés sous PS, mais provoqueront des erreurs si on les passe à une API nécessitant un nom de chemin :

```
"C:\Windows\$NotExist\explorer.exe",
"C:\Windows\\\\\\\\\\\\\\\\\\\\\explorer.exe",

"C:\Windows\///\\\\explorer.exe" | Dir
```

Le chemin UNC suivant, pointant vers un répertoire inexistant ne provoque pas d'erreur :

```
dir '\\localhost\c$\temp\inconnu\*'
```

Là où l'accès via un lecteur déclenche une exception :

```
dir 'c:\temp\inconnu\*'
```

Le comportement de l'analyse du chemin suivant diffère selon la version de Powershell :

```
dir '\\\localhost\c$\temp\' #Path OK avec la V2, en erreur avec la V3
```

Le chemin '…' est reconnu par les APIs de Powershell, mais le cmdlet **Set-Location** l'interdit :

```
C:\temp> dir '...'

    Répertoire : C:\


Mode                LastWriteTime     Length Name
d----        08/10/2014     22:36            <DIR> Temp
C:\temp> set-location '...'
set-location : Un objet n'existe pas à l'emplacement spécifié C:\temp\....
```

5.11 Interopérabilité

On peut aussi se préoccuper d'interopérabilité et de compatibilité de noms de fichier entre Windows / Linux / Unix / Mac :

https://support.apple.com/kb/HT5923?viewlocale=fr_FR

https://en.wikipedia.org/wiki/Filename

6 API et path helper

La variable automatique ***$ExecutionContext*** facilite l'accès aux APIs de manipulation de chemin par la propriété suivante : ***$ExecutionContext.SessionState.Path*** qui propose une instance de la classe PathIntrinsics.

Voici des exemples d'usage de certaines de ses méthodes.

6.1 Méthodes de gestion de path Powershell

Pour faciliter l'écriture et accélérer l'accès à l'instance, nous utiliserons la variable suivante :

```
$PathHelper=$ExecutionContext.SessionState.Path
```

Ces APIs sont implémentées dans les différents cmdlets de manipulation de chemin **-Path* .

6.1.1 GetResolvedProviderPathFromProviderPath

Renvoie un ou plusieurs noms de chemins résolus à partir d'un chemin interne de provider :

```
$Pathhelper.GetResolvedProviderPathFromProviderPath('c:\*','FileSystem')
```

Cet appel renvoie une liste des noms de fichiers et de noms répertoire présent dans le chemin indiqué du Provider FileSystem.

Le chemin suivant n'est pas un chemin interne de provider :

```
$Pathhelper.GetResolvedProviderPathFromProviderPath('FileSystem::c:\temp\*
','FileSystem')
```

```
#Ne renvoie rien
$pathhelper.GetResolvedProviderPathFromProviderPath('.\*','FileSystem')
```

```
#Renvoie les entrées du répertoire [environment]::CurrentDirectory
```

6.1.2 GetResolvedProviderPathFromPSPath

Renvoie un ou plusieurs noms de chemins à partir d'un chemin absolu ou relatif de type drive-qualified ou provider-qualified :

```
[ref]$provider=$null
$pathHelper.GetResolvedProviderPathFromPSPath('Psionic:\Tools',$provider)
```

```
G:\PS\PsIonic\Tools\
```

```
$provider
```

```
Value
-----
Microsoft.PowerShell.Core\FileSystem
```

Cette méthode retrouve le provider à partir d'un nom de chemin Powershell existant et résout également le drive Powershell, c'est-à-dire qu'il remplace le nom du lecteur par le root du PSdrive.

```
$pathHelper.GetResolvedProviderPathFromPSPath('FileSystem::c:\Windows\*.ex
e', $provider)
```

```
Path
----
c:\Windows\bfsvc.exe  …
```

Notez le cas suivant :

```
C:\temp> $p='Registry::HKEY_LOCAL_MACHINE\SOFTWARE\A*'
C:\temp> $pathHelper.GetResolvedProviderPathFromPSPath($p,$provider)
```

```
HKEY_LOCAL_MACHINE\SOFTWARE\Activision
```

Ici on perd l'information de la localisation, on a bien un chemin, mais les APIs ne pourront retrouver le provider qui lui est associé. Hormis pour le FileSystem, les noms de chemin de type Provider-Qualified ne peuvent donc être scindés.

6.1.2.1 Retrouver les fichiers cachés

Malheureusement cette méthode ne renvoie pas les fichiers cachés :

```
$pathHelper.GetResolvedProviderPathFromPSPath('c:\*',$provider)
```

En revanche la méthode suivante les renvoie :

```
$ExecutionContext.InvokeProvider.ChildItem.Get("C:\",$false,$true,$false)
```

Powerhell ayant un petit côté espiègle, un peu d'anthropomorphisme ça ne fait pas de mal, cette méthode peut poser problème avec des jokers :

```
$ExecutionContext.InvokeProvider.ChildItem.Get("C:\*",$false,$true,$false)
```

```
Exception lors de l'appel de « Get » avec « 4 » argument(s) : « Élément C:\hiberfil.sys introuvable. »
```

Donc, pour résoudre le problème de la résolution, on utilise une classe cité dans un lien précédent :

```
$p=New-Object System.Management.Automation.WildcardPattern -ArgumentList
'h*','IgnoreCase,CultureInvariant'


$ExecutionContext.InvokeProvider.ChildItem.Get("C:\",$false,$true,$false)|
 Where { $p.ismatch($_.name)}
```

```
   Répertoire : C:\

Mode   LastWriteTime         Length          Name
----   -------------         ------          ----
-a-hs  23/10/2014     10:38  2515886080      hiberfil.sys
```

Roulez tambours sonnez trompettes, c'est jour de fête !

Mais de courte durée ☺, car le provider Registry la gâche :

```
$ExecutionContext.InvokeProvider.ChildItem.Get("hklm:\",$false,$true,$fals
e)
```

```
Exception lors de l'appel de « Get » avec « 4 » argument(s) : « Accès au registre demandé non autorisé. »
```

Si on souhaite réaliser un traitement générique *Get-ChildItem* reste la solution, si on cible uniquement le provider FileSystem les classes dotNet de gestion de chemin suffiront.

6.1.3 GetResolvedPSPathFromPSPath

Renvoie un ou plusieurs noms de chemins résolus à partir d'un chemin d'accès Windows PowerShell spécifié :

```
$pathHelper.GetResolvedPSPathFromPSPath('Psionic:\Tools')
```

```
Path
----
PsIonic:\Tools
…
```

Cette méthode ne remplace pas le nom du lecteur par le root du PSdrive, ni ne supprime le nom du provider s'il est précisé :

```
$PathHelper.GetResolvedPSPathFromPSPath('FileSystem::c:\*')
```

```
Path
----
FileSystem::c:\Program Files
…
```

Elle résout les chemins relatifs :

```
C:\temp> $PathHelper.GetResolvedPSPathFromPSPath('.\*')
```

```
Path
----
C:\temp\Logs …
```

Avec une particularité déjà vue :

```
C:\temp> $pathhelper. GetResolvedPSPathFromPSPath ('FileSystem::.\*')
```

```
#Renvoie les entrées du répertoire [environment]::CurrentDirectory
Path
----
FileSystem::.\Documents
…
```

6.1.4 GetUnresolvedProviderPathFromPSPath

Obtient un chemin de provider interne irrésolu à partir d'un chemin de type drive-qualified ou provider-qualified absolu ou relatif. Les variantes de cette méthode peuvent retourner le chemin de provider interne irrésolu avec ou sans information sur le provider ou le lecteur associé au chemin :

```
C:\temp> $pathHelper.GetUnresolvedProviderPathFromPSPath('*')
```

```
C:\temp\*
```

```
$pathHelper.GetUnresolvedProviderPathFromPSPath('FileSystem::.\*')
```

```
.\*
```

Cette méthode référence la localisation courante :

```
HKLM:\> $pathHelper.GetUnresolvedProviderPathFromPSPath('*')
```

```
HKEY_LOCAL_MACHINE\*
```

Comme vu précédemment ici il faut ajouter le nom du provider de la localisation courante pour obtenir un nom de chemin utilisable.

Selon le type du chemin le résultat diffère :

```
C:\temp> New-PSDrive -name System -root C:\Windows\System -psp FileSystem
$pathHelper.GetUnresolvedProviderPathFromPSPath('FileSystem::System:\')
```

```
System:\
$pathHelper.GetUnresolvedProviderPathFromPSPath('System:\')
```

```
C:\Windows\System\
```

L'exemple suivant renvoie un chemin même si l'entrée n'existe pas :

```
$pathHelper.GetUnresolvedProviderPathFromPSPath('.\NotExist.txt')
```

```
C:\temp\NotExist.txt
```

là où **Resolve-Path** échoue :

```
Resolve-Path '.\NotExist.txt'
```

```
Resolve-Path : Impossible de trouver le chemin d'accès « C:\temp\NotExist.txt », car il n'existe pas.
```

6.1.5 IsProviderQualified

Vérifie si le chemin d'accès spécifié est un chemin de type provider-qualified.

```
$PathHelper.IsProviderQualified('Test.ps1')
```

```
False
$PathHelper.IsProviderQualified('WSMan:\localhost\')
```

```
False
$PathHelper.IsProviderQualified('hklm:\NotExist::\*')
```

```
False
$PathHelper.IsProviderQualified('NotExist::\*')
```

```
True
$PathHelper.IsProviderQualified('FileSystem::C:\Temp')
```

```
True
```

Notez que le nom du provider peut ne pas exister ou ne pas être chargé.

Un cas particulier :

```
$PathHelper.IsProviderQualified('ceci est un texte avec ::')
```

```
True
```

6.1.6 IsPSAbsolute

Vérifie si le chemin d'accès spécifié est un chemin de type drive-qualified.

```
[ref]$DriveName=$null
$PathHelper.IsPSAbsolute('Test.ps1',$DriveName)
```

```
False
$PathHelper.IsPSAbsolute('C:\NotExist\Test.ps1',$DriveName)
```

```
True
$DriveName
```

```
Value
-----
C
$PathHelper.IsPSAbsolute('A Z:\Temp',$DriveName)
```

```
True
```

```
$DriveName
```

```
Value
-----
A Z
```

Notez que la création d'un nom de lecteur contenant des espaces est possible.

Si la fonction renvoie *true*, le chemin n'est donc pas un chemin relatif.

Notez le cas suivant :

```
$Filename='FileSystem::C:\notexist\Test.ps1'
$PathHelper.IsPSAbsolute($Filename,$DriveName)
```

```
True
```

```
$DriveName
```

```
Value
-----
FileSystem
```

Remarquez que le nom du drive contient le nom du provider. On doit combiner les appels :

```
$Path=$pathHelper.GetunResolvedProviderPathFromPSPath($Filename)
$PathHelper.IsPSAbsolute($Path,$DriveName)
```

```
True
```

```
$DriveName
```

```
Value
-----
C
```

6.1.7 IsValid

Vérifie si le chemin d'accès spécifié est syntaxiquement et sémantiquement valide pour le fournisseur.

Un exemple de chemin ayant une syntaxe et une sémantique correcte pour le provider FileSystem :

```
$PathHelper.IsValid('C:\Temp')
```

```
True
```

Si le chemin n'existe pas la méthode renvoie une exception :

```
$PathHelper.IsValid('Z:\Temp')
```

```
Exception lors de l'appel de «IsValid» avec «1» argument(s): «Lecteur introuvable. Il n'existe aucun lecteur nommé «Z».
```

En ajoutant le nom du provider cela fonctionne :

```
$PathHelper.IsValid('FileSystem::Z:\Temp')
```

```
True
```

```
$PathHelper.IsValid('FileSystem::1:\Temp')
```

```
False
```

Un exemple de chemin ayant une syntaxe correcte, mais une sémantique incorrecte pour le provider FileSystem :

```
$PathHelper.IsValid('C:\Temp\Test*.ps1')
```

False

```
$PathHelper.IsValid('C:\Temp\Test[1].ps1')
```

True

L'exemple précédent est correct pour le provider Registry :

```
$PathHelper.IsValid('HKLM:\Test*.ps1')
```

True

Le résultat pour un même chemin peut donc différé selon la localisation courante :

```
Cd c:\
$PathHelper.isValid('t*')
```

False

```
Cd Hklm:\
$PathHelper.isValid('t*')
```

True

Il reste quelques cas qui posent problème avec cette méthode :

```
$PathHelper.IsValid(':Temp')
```

True

```
$PathHelper.IsValid("FileSystem::C:\AUX")
```

Le nom ***AUX*** est un nom réservé, la méthode ne teste pas ce cas-là.

True

```
C:\temp> $PathHelper.IsValid('C:\temp?\')
```

False

```
C:\temp> $PathHelper.IsValid('C:\temp?\CON')
```

True

Quant à celui-ci il s'agit d'un bug je pense :

```
$PathHelper.isValid('Registry::..\temp')
```

Exception lors de l'appel de «IsValid» avec «1» argument(s): «La tentative d'exécution de l'opération IsValidPath sur le fournisseur «Registry» a échoué pour le chemin d'accès «..\temp». La référence d'objet n'est pas définie à une instance d'un objet.»

6.1.8 NormalizeRelativePath

Normalise un nom de chemin relatif d'après un nom de chemin :

```
C:\temp> $PathHelper.NormalizeRelativePath('.','C:\Windows\System32')
```

..\..\temp

Cet exemple construit le chemin relatif permettant d'aller du répertoire 'C:\Windows\System32' vers le répertoire 'C:\temp'.

L'API se base donc sur le chemin courant, mais ne gère pas un drive différent existant :

```
C:\temp> $PathHelper.NormalizeRelativePath('G:.','C:\Windows\System32')
```

..\..\

6.2 Méthodes d'accès au provider

La variable automatique ***$ExecutionContext*** permet également l'accès à certaines APIs du provider par la propriété suivante : ***$ExecutionContext.InvokeProvider*** .

J'utilise les mots 'du provider' et pas 'des providers', car cette propriété n'est qu'un sélecteur qui dépend de la localisation courante, à moins que le chemin précisé soit univoque (*Provider-Qualified Path*).

Sachez que cette méthode renvoie *$True* si l'élément ne contient que des sous-conteneurs :

```
$ExecutionContext.InvokeProvider.Item.IsContainer('c:\temp\test\t*')
```

Et que celle-ci permet de savoir si le chemin contient au moins un élément :

```
$ExecutionContext.InvokeProvider.Item.Exists('*')
```

Je vous laisse étudier les différentes propriétés et méthodes que cette instance propose.

Note : la variable ***$PScmdlet*** d'une fonction avancée permet également d'appeler certaines de ces APIs.

6.3 La classe PathInfo

Le Framework dotNet propose, entre autres, les classes *System.IO.Path* et *System.IO.Fileinfo*, celle-ci gèrent uniquement des noms de chemin du FileSystem, Powershell propose la classe *PathInfo* :

```
C:\temp> $pwd|fl *

Drive      : C
Provider   : Microsoft.PowerShell.Core\FileSystem
ProviderPath : C:\temp
Path       : C:\temp
```

Elle décompose un nom de chemin Powershell, mais il n'existe pas de constructeur pour cette classe :

```
$F=New-object System.Management.Automation.PathInfo('FileSystem::C:\temp')

New-object : Constructeur introuvable. Impossible de trouver un constructeur approprié pour le type
System.Management.Automation.PathInfo.
```

Certains cmdlets l'utilisent :

```
HKLM:\> 'C:\temp'|Resolve-Path|fl *

Drive      : C
Provider   : Microsoft.PowerShell.Core\FileSystem
ProviderPath : C:\temp
Path       : C:\temp
```

Pour un chemin de type Provider-Qualified, la notion de drive n'existe pas :

```
C:\temp> 'Registry::HKEY_CURRENT_USER\'|Resolve-Path|fl *

Drive      :
Provider   : Microsoft.PowerShell.Core\Registry
ProviderPath : HKEY_CURRENT_USER\
Path       : Registry::HKEY_CURRENT_USER\
```

```
C:\temp> Set-Location Registry::
Dir
```

```
   Hive:

Name                           Property
----                           --------
HKEY_LOCAL_MACHINE
HKEY_CURRENT_USER              DisplayStatistics : 0
                               LastBuild         :
                               Rebuild           : 0
                               (default)         : 0
                               profDataStr       :   20
HKEY_CLASSES_ROOT              (default) :
                               EditFlags : {0, 0, 0, 0}
HKEY_CURRENT_CONFIG
HKEY_USERS
HKEY_PERFORMANCE_DATA          Global : {80, 0, 69, 0, 82, 0, 70, 0, 1, 0, 0, 0, 1, 0, 0...}
                               Costly : {80, 0, 69, 0, 82, 0, 70, 0, 1, 0, 0, 0, 1, 0, 0...}
```

Le chemin précédent accède directement au magasin de données géré par le provider, on affiche donc toutes les ruches.

```
Microsoft.PowerShell.Core\Registry::> Get-Location |Fl *
```

```
Drive        :
Provider     : Microsoft.PowerShell.Core\Registry
ProviderPath :
Path         : Microsoft.PowerShell.Core\Registry::
```

Tous les providers ne supportent pas ce chemin de localisation sur la racine.

Par exemple celui du FileSystem ne manipule pas la liste des disques physiques.

7 Quelques exemples

L'usage de ces différents type de chemin peut impacter vos scripts, principalement ceux utilisant des paramètres ciblant implicitement un provider particulier.

On peut utiliser simplement les cmdlets Powershell, vous trouverez dans le répertoire 'Source' une fonction *Get-Path* utilisant **Test-Path**, **Resolve-Path** et **Convert-Path.**

Vous trouverez également des données de test constituées de noms de chemin de différentes constructions valides ou invalides. Ceux-ci permettront de vérifier la gestion des chemins d'un script ou d'une fonction voir d'un cmdlet.

Voici une partie de la gestion d'un chemin issu du code de la fonction *Get-Path :*

```
if (!(Test-Path $Path)) {
 Write-host "Cannot find path '$Path' because it does not exist."
}
$resolvedPaths = $Path | Resolve-Path | Convert-Path
```

Le premier test vérifie si le chemin existe, puis résout le chemin (les jokers) et le dernier transforme le chemin en un chemin interne de provider.

Une fois chargé en dot source la fonction et les données, on peut effectuer un premier test :

```
Cd c:\temp
#cd hkcu:\Test
$Error.clear()
foreach ($p in $Datas) {
 Write-warning "path=$p"
 Get-path -path $p
  #  Write-warning "Literalpath=$p"
  #  Get-path -literal $p
}
 #Regroupe les erreurs selon leurs type
$g=$Error|Group-Object -Property {$_.Exception.GetType()}
$g|Select Count,Name
```

```
Count Name
----- ----
  98 System.Management.Automation.ItemNotFoundException
  30 System.ArgumentException
   7 System.Management.Automation.ProviderNotFoundException
   2 System.ArgumentOutOfRangeException
   9 System.Management.Automation.WildcardPatternException
   1 System.Management.Automation.ParameterBindingException
   7 System.Management.Automation.DriveNotFoundException
```

Sans rentrer dans les détails, on constate que la moitié des chemins posent problème et sont à priori gérés pour le paramètre *-Path*, pour s'en assurer l'usage de Pester est nécessaire, mais ce n'est pas le sujet. Il reste également à valider ces données avec le paramètre *–LiteralPath*, puis exécuter de nouveau ces tests en modifiant la localisation courante pour la gestion des chemins relatifs. Et enfin, si nécessaire, recommencer ces opérations avec la version 2 de Powershell.

On peut donc rapidement se dire que cette approche convient pour la plupart des noms de chemin. Presque, car on ne peut valider les chemins inexistant, par exemple un nouveau fichier dans un répertoire existant, et la construction comporte un piège.

Pour gérer le chemin suivant *C:\Temp\frm[az]** on doit utiliser le paramètre *–LiteralPath*, mais la présence du joker ne le permet pas, on utilise donc le paramètre *–Path* en échappant une partie du globbing :

```
Get-Path -path 'C:\Temp\frm`[az`]\*'
```

```
Cannot find path because it does not exist.
```

Mais **Resolve-Path** supprime l'échappement et *via* le pipeline transmet les noms de chemins trouvés au cmdlet **Convert-Path.** Celui-ci traite le chemin en utilisant le paramètre *–Path* qui ne trouve pas de correspondance pour le nom de répertoire (frma, frmz).

Powershell, un chemin de croix ?

7.1 Chemin et API dotNet

L'usage du cmdlet **Test-Path** ne nous assure en rien que le chemin testé référence un chemin du FileSystem, si on lui passe en paramètre un chemin de la registry, le cmdlet renverra $true.

Pour s'assurer que le chemin reçu est bien un chemin pointant sur le FileSystem et ainsi l'utiliser sans risque d'erreur avec une méthode d'une classe dotNet et si seul le provider FileSystem est concerné, la fonction *Get-NormalizedFileSystemPath*, présente dans le répertoire 'Source', est une solution. Elle gère les chemins inexistants, les chemins relatifs pointent sur la localisation courante, mais pas les chemins avec un PSdrive pointant sur le FileSystem :

```
New-PSDrive -name 'Test' -root C:\Temp -psp FileSystem > $null
cd c:\temp
cd hklm:\
'~','FileSystem::c:\Notexist','c:\Notexist','Test:\Notexist','hklm:\','.'|
 Get-NormalizedFileSystemPath
```

Vous pouvez également vérifier son comportement avec les données de tests.

7.2 PSCX

Ce projet open source contient des classes prenant en charge cette de gestion des noms de chemin. La classe C# PscxPath implémente un attribut de transformation facilitant la manipulation des noms de chemin, son usage implique de livrer ce module sur chaque serveur/poste.

Voir aussi :

v3 suggestion - providers and paths: the PathInfo class is not enough - by Oisín Grehan
https://connect.microsoft.com/PowerShell/Feedback/Details/525584

7.3 Une autre approche : New-PSPathInfo

De mon côté j'ai décidé d'étudier une solution orientée 'fonctionnelle' plutôt que technique, car les traitements et vérifications autour des chemins de fichier sont très souvent les mêmes.

Le démarche se concentre sur que l'on veut faire et pas sur comment le faire. Ce qui importe ici c'est la réponse à une question et pas comment résoudre la question. A l'origine je comptais gérer tous les types de chemin et pas seulement des fichiers.

Par exemple pour extraire des fichiers d'une archive on souhaite savoir si le nom de chemin reçu est un répertoire valide : c'est-à-dire ne pas contenir de joker, ni de caractères interdits, il doit exister et pointer sur le provider du système de fichier.

Pour écrire des données dans un fichier on souhaite savoir si le nom de chemin reçu est valide : c'est-à-dire ne pas contenir de joker, ni de caractères interdits, s'il existe ou pas et pointer sur le provider du système de fichier.

Etc.

Pour répondre aux questions qu'on peut se poser, l'ajout de membres synthétiques permet de spécialiser l'objet créé selon des règles, une méthode répondra donc à une question en particulier.

Une fois chargée la fonction New-PSPathInfo, son usage est aisé :

```
C:\temp> $info=New-PSPathInfo -Path $PSHome
C:\temp> $info
```

```
isProviderQualified          : False
Provider                     : FileSystem
isWildcard                   : False
Win32PathName                : C:\Windows\System32\WindowsPowerShell\v1.0
isItemExist                  : True
isPSValid                    : True
LastError                    :
isCurrentLocationFileSystem  : True
ResolvedPSPath               : C:\Windows\System32\WindowsPowerShell\v1.0
isUNC                        : False
isParentItemExist            : True
isAbsolute                   : True
ResolvedPSFiles              : {C:\Windows\System32\WindowsPowerShell\v1.0}
isProviderExist              : True
isDriveExist                 : True
Name                         : C:\Windows\System32\WindowsPowerShell\v1.0
CurrentDriveName             : C
isFileSystemProvider         : True
DriveName                    : C
asLiteral                    : False
```

Cela permet d'analyser et de contrôler le contenu d'un paramètre avant l'exécution du traitement qui lui ne gère que les erreurs d'IO.

La propriété *Win32PathName* contient le chemin du fichier qui peut être décomposé ainsi :

```
$info.Win32PathName.GetasFileInfo()
```

Les méthodes de base :

```
$info|gm -MemberType ScriptMethod
```

```
   TypeName : PSPathInfo

Name                           MemberType   Definition
----                           ----------   ----------
GetFileName                    ScriptMethod   System.Object GetFileName();
IsaValidNameForTheFileSystem   ScriptMethod   System.Object IsaValidNameForTheFileSystem();
```

Vous constatez que la version actuelle de cette fonction reste tout de même bien attachée à la notion de fichier…

Ajoutons des ‘réponses’ supplémentaires :

```
C:\temp> $info=New-PSPathInfo -Path $PSHome |
            Add-FileSystemValidationMember
$info|gm -MemberType ScriptMethod
```

```
   TypeName : PSPathInfo

Name                           MemberType   Definition
----                           ----------   ----------
GetFileName                    ScriptMethod   System.Object GetFileName();
IsaValidNameForTheFileSystem   ScriptMethod   System.Object IsaValidNameForTheFileSystem();

IsCandidateForCreation         ScriptMethod System.Object IsCandidateForCreation();
IsDirectoryExist               ScriptMethod System.Object IsDirectoryExist();
isFileSystemItemContainsResolvedFiles ScriptMethod System.Object ….
isFileSystemItemFound          ScriptMethod System.Object isFileSystemItemFound();
```

Voici quelque exemple de code répondant à une question.

Le nom de chemin est-il un nom valide pouvant être crée sur le FileSystem ?

```
Add-Member -Membertype Scriptmethod -Name IsCandidateForCreation {
  $this.IsaValidNameForTheFileSystem() -and ($this.isItemExist -eq $false)
}
```

Le nom de chemin valide renvoie-t-il un et un seul élément ?

```
Add-Member -Membertype Scriptmethod -Name isFileSystemItemFound {
 $this.isaValidFileSystemPath() -and $this.isItemExist -and
$this.isWildcard -eq $false }
```

Le nom de chemin valide existant et comportant des jokers, renvoie-t-il au moins une entrée ?

```
Add-Member -Membertype Scriptmethod -Name isFileSystemItemContainsResolvedFiles {
 $this.isaValidFileSystemPath() -and $this.isItemExist -and $this.isWildcard -and
$this.ResolvedPSFiles.Count -gt 0 }
```

Un exemple de gestion d’un paramètre attendant un nom de chemin (projet ConvertForm) :

```
 #Le PSPath doit exister, ne pas ?tre un r?pertoire, ne pas contenir de globbing et ?tre sur le FileSystem
 #On doit lire un fichier.
 #On pr?cise la raison de l'erreur
if (!$SourcePathInfo.isFileSystemItemFound())
{
  if (!$SourcePathInfo.isDriveExist)
  {Throw (New-Object System.ArgumentException(($ConvertFormMsgs.DriveNotFound  -F $FileName),'Source')) }

     #C'est un chemin relatif, le drive courant appartient-il au provider FileSystem ?
  if (!$SourcePathInfo.isAbsolute -and !$SourcePathInfo.isCurrentLocationFileSystem)
  {Throw (New-Object System.ArgumentException(($ConvertFormMsgs.FileSystemPathRequiredForCurrentLocation -F $FileName),

  if (!$SourcePathInfo.isFileSystemProvider)
  {Throw (New-Object System.ArgumentException(($ConvertFormMsgs.FileSystemPathRequired -F $FileName),'Source')) }

  if ($SourcePathInfo.isWildcard)
  {Throw (New-Object System.ArgumentException(($ConvertFormMsgs.GlobbingUnsupported -F $FileName),'Source'))}
  else
  {Throw (New-Object System.ArgumentException(($ConvertFormMsgs.ItemNotFound -F $FileName),'Source')) }
}
$SourceFI=$FileName.GetasFileInfo()
if ($SourceFI.Attributes -eq 'Directory')
{ Throw (New-Object System.ArgumentException(($ConvertFormMsgs.ParameterMustBeAfile -F $FileName),'Source')) }
```

Ici, même si cela reste du code, il n'existe plus de trace d'appel d'APIs et il est selon moi bien plus lisible.

Il reste que cette fonction a elle aussi ses limites, la documentation n'existe pas bien que le code soit commenté, certaines propriétés nécessitent une interprétation contextuelle, dans certains cas les exceptions sont trappées puis redéclencher dans l'appelant, les noms long et court ne sont pas encore gérés.

Le code est stable et répond aux besoins, toutefois je pense qu'il est possible de mieux faire. Par exemple l'usage de classes C# faciliterais le polymorphisme nécessaire à la gestion de règles pour différents providers.

8 Conclusion

Gérer des chemins semble trivial, comme vous avez pu le constater lors de votre lecture ce n'est pas vrai sous Powershell. On peut pratiquer Powershell sans connaitre ces APIs, mais de comprendre ce que sous-entend le terme ***Path*** dans son contexte me semble important.

Je trouve qu'il y a un véritable manque pour gérer le contenu d'un simple paramètre et si dans le scripting et le développement on ne peut pas supprimer tous les problèmes, on peut au moins tenter d'en réduire le nombre ☺

Une fois encore n'hésitez pas à consulter le site MSConnect, car la gestion des paths, que ce soit des cdmlets ou des APIs contient de nombreux bugs :
https://connect.microsoft.com/PowerShell/Feedback