Utiliser Powershell pour administrer SQL Server 2005 et 2008

par Rudi Bruchez (contact)

Date de publication : 15/12/2008

Utiliser Powershell comme interpréteur de commandes et langage de script pour l'administration de SQL Server.
Article paru dans l'édition de novembre 2008 de IT Pro Magazine, édition française.

I - Qu'est Powershell ?

De longue date, les environnements UNIX ont disposé d'interpréteurs de commandes puissants, qui permettaient aux administrateurs do contrôler le système en ligne de commande. Il s'agissait d'une nécessité pour un système d'exploitation essentiellement en mode console ou texte, les interfaces graphiques comme X-Windows étant inutiles dans un contexte de serveur. Microsoft semble d'ailleurs se diriger dans cette voie, en permettant à son nouveau système d'exploitation serveur, Windows Server 2008, de s'exécuter dans un mode non graphique, le mode core. Ces interpréteurs de commandes, aussi appelés shells, permettent de lancer les commandes et exécutables du système interactivement, en interprétant les instructions saisies ligne par ligne, et offrent également un mode d'exécution d'un fichier de script, contenant une suite de commandes. Il manquait au monde Windows un shell avancé, permettant de piloter tous les éléments du système et les principales applications serveur en ligne de commande, sans devoir se déplacer à travers d'innombrables fenêtres. Cet outil existe maintenant, il s'appelle Powershell. Powershell est un shell évolué, qui reprend les fonctionnalités des shells UNIX traditionnels, en les étendant. Basé sur le framework .NET, il est complètement orienté objet, et permet d'accéder à toutes les assemblies .NET présentes sur le système. On peut donc le considérer comme un langage à part entière du monde .NET. Traditionnellement, les shells utilisent le concept de redirection ou de tube (pipe, ou pipeline). Le tube permet de transmettre le résultat (l'output) d'une commande vers une autre commande, qui le récupère en input. L'opérateur de tube est le signe |. Voici un exemple de tube utilisable sous Windows :

```
type fichier | find "mot cle" | sort
```

Ce qui signifie : affiche le contenu du fichier, envoie-le vers la commande find, qui cherchera ligne par ligne, et envoie les lignes trouvées vers la commande sort, qui les triera. Ces redirections passent des chaînes de caractères d'une commande à l'autre. Powershell implémente un type de pipe avancé, qui ne transmet pas des chaînes de caractères, mais des objets .NET, et cela confère une grande puissance de manipulation des informations. Pour permettre le traitement de ces objets en entrée et en sortie, les commandes disponibles en Powershell sont des commandes internes spécifiques, compilées en assemblies .NET, qui sont appelées des cmdlets (Command-Let). Ces cmdlets ont une syntaxe de type Verbe-Nom, comme par exemple Get-Help, ou Set-Location. On peut y faire référence plus pratiquement en leur attribuant des alias. Par exemple, en Powershell, la commande dir est un alias du cmdlet Get-ChildItem. Vous pouvez créer vos propres alias pour personnaliser votre système. Pourquoi un dir correspond-il à un cmdlet nommé Get-ChildItem, et pas simplement Get-Files, par exemple ? Un dir n'affiche-t-il pas une liste de fichiers dans un répertoire ? En Powershell, la représentation d'éléments dans des répertoires n'est pas limitée aux fichiers dans l'arborescence du disque. Toute structure hiérarchique peut être manipulée comme des répertoires, dès l'instant où un fournisseur spécifique est fourni. Ainsi, Powershell propose le concept de PSDrive, un lecteur qui n'est pas limité à une partition de disque. Le cmdlet Get-PSDrive affiche la liste des PSDrives disponibles dans l'instance de Powershell. Par défaut, à part le système de fichier, quelques PSDrives sont disponibles pour gérer la base de registre, les variables d'environnement, les alias Powershell. Voici un exemple de code qui parcourt la clé HKEY_LOCAL_MACHINE de la base de registre :

```
cd HKLM:
cd '\SOFTWARE\Microsoft\Microsoft SQL Server\90'
dir
```

Ici, la commande dir, c'est-à-dire Get-ChildItem, retrouve une liste d'éléments présents dans ce niveau de hiérarchie. Ces éléments sont des objets, et sont affichés par défaut dans une représentation tabulaire, qui montre quelques unes de leurs propriétés. Des cmdlets permettent de personnaliser cet affichage. Par expemple, Format-List affiche les propriétés ligne par ligne, dans une représentation Nom : valeur. Comme cette commande accepte en entrée une collection d'objets, vous pouvez l'utiliser dans un tube, comme ceci :

```
dir | Format-List
```

Voici l'exemple du retour, pour l'un des objets présent dans le répertoire HKLM:\SOFTWARE\Microsoft\Microsoft SQL Server\90 :

```
Name         : ProductID
ValueCount   : 2
Property     : {ProductID77558, DigitalProductID77558}
SubKeyCount : 0
```

Nous le verrons, SQL Server 2008 est livré avec un "minishell" Powershell qui ajoute un PSDrive SQLSERVER: permettant de naviguer dans l'arborescence des objets du serveur. Certains cmdlets acceptent en entrée une collection d'objets, mais pour ceux qui ne peuvent recevoir qu'un seul objet, ou pour les commandes externe du système, ou les méthodes .NET qui demandent des paramètres scalaires, il nous faut un moyen de traiter le résultat d'un cmdlet objet par objet. Le cmdlet Foreach-Object offre cette possibilité en bouclant sur chaque objet retourné dans le tube pour effectuer un traitement. Comme tout langage, Powershell accepte la création de variables. Ces variables sont préfixées par le signe $, et il existe quelques variables "automatiques", dont la plus utilisée et $_, qui correspond à l'objet par défaut. Le concept et la syntaxe sont familiers aux connaisseurs du langage Perl. Cet objet par défaut permet de référencer l'objet et ses propriétés dans la boucle de Foreach-Object (qui possède deux alias : foreach, ou %). Voici un exemple de traitement en boucle, qui retourne seulement la propriété Name des objets listés :

```
dir | Foreach-Object { Write-Output $_.Name }
```

II - SMO

SMO, ou SQL Server Management Objects, est une collection d'assemblies .NET qui permettent d'administrer par programmation tous les éléments d'une serveur SQL. Cette bibliothèque est disponible depuis SQL Server 2005, et peut également se connecter à SQL Server 2000. Elle publie des objets, méthodes et propriétés pour obtenir des informations et exécuter des actions sur le serveur. Il s'agit d'une alternative au code T-SQL pour l'administration de SQL Server, sous forme d'une couche intermédiaire : au final, les commandes SMO génèrent du code T-SQL. Comme il s'agit d'assemblies, rien n'est plus simple que de les utiliser en Powershell. Voici un premier exemple de déclaration et d'utilisation de la bibliothèque SMO :

```
[System.Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo") | out-null
$srv = New-Object "Microsoft.SqlServer.Management.Smo.Server" ".\SQL2008"
Get-Member -InputObject $srv
```

Le cmdlet Get-Member liste les méthodes et propriétés d'un objet. Vous pouvez le passer au cmdlet Where-Object pour filtrer le résultat, et donc recherche des fonctionnalités de SMO. Par exemple, cherchons les membres nous permettant de gérer les connexions (logins) :

```
Get-Member -InputObject $srv | Where-Object { $_.Name -ilike "*login*"}
```

-ilike représente un opérateur de recherche like non sensible à la casse.

Lorsque vous aurez pris l'habitude de ces cmdlets, vous pourrez raccourcir votre code en utilisant des alias et des abréviations de paramètres. La commande ci-dessus peut aussi s'écrire :

```
gm -i $srv | ? { $_.Name -ilike "*login*"}
```

Le résultat retourné est (tronqué pour améliorer la lisibilité) :

```
TypeName: Microsoft.SqlServer.Management.Smo.Server

Name          MemberType Definition
----          ---------- ----------
get_LoginMode Method     [...].ServerLoginMode get_LoginMode()
get_Logins    Method     [...].LoginCollection get_Logins()
set_LoginMode Method     System.Void set_LoginMode(ServerLoginMode value)
LoginMode     Property   [...].ServerLoginMode LoginMode {get;set;}
Logins        Property   [...].LoginCollection Logins {get;}
```

Nous voyons donc que nous disposons d'une collection d'objets nommée Logins. Retrouvons la liste des membres disponibles dans un objet Login, en cherchant particulièrement ce qui pourrait toucher à la base de données par défaut du login :

```
gm -i $srv.Logins[0] | where { $_.Name -ilike "*default*" }
```

(where, comme ?, est un alias de Where-Object). Nous trouvons ceci :

```
Name               MemberType Definition
----               ---------- ----------
get_DefaultDatabase Method     System.String get_DefaultDatabase()
set_DefaultDatabase Method     System.Void set_DefaultDatabase(String value)
DefaultDatabase     Property   System.String DefaultDatabase {get;set;}
```

Il existe donc une propriété DefaultDatabase, qui est accessible en lecture et en écriture.
Grace à l'expressivité du modèle cmdlets + tubes, il est possible de réaliser des opérations complexes parfois en une seule ligne de commande. Utilisons notre propriété DefaultDatabase :

```
$srv.Logins | where {$_.DefaultDatabase -eq "master"} | foreach {$_.DefaultDatabase = "AdventureWorks"}
```

qui change la base de données par défaut des connexions (logins) en AdventureWorks, seulement si cette base par défaut est Master.

III - Administrer votre parc de serveurs

En vous aidant des structures de contrôle de Powershell, et du cmdlet Get-Content, qui lit le contenu d'un fichier et retourne chaque ligne comme un objet de type chaîne de caractères (en .NET, tout type de données est un objet), vous pouvez aussi automatiser l'administration de plusieurs serveurs SQL. En imaginant que vous avez listé l'adresse de vos serveurs SQL dans un fichier nommé sqlservers.txt, voici du code qui retourne tous vos serveurs qui comportent la connexion liée au groupe administrateurs de la machine, créée à l'installation de SQL Server 2005, avec des privilèges sysadmin :

```
foreach ($svr in get-content "sqlservers.txt") {
    $srv=New-Object "Microsoft.SqlServer.Management.Smo.Server" "$svr"
    trap {"Erreur! $_"; continue } $srv.Logins | where {$_.Name -eq "BUILTIN\Administrators" -and
 $_.IsMember("sysadmin") } | select Parent, Name
}
```

Ici, trap est une commande Powershell peu documentée, qui gère les erreurs éventuelles.

IV - Créer des scripts

Outre saisir des commandes ligne par ligne, vous pouvez bien entendu créer des scripts, enregistrés dans des fichiers à l'extension .ps1, qui contiennent une suite de commandes Powershell. Ces scripts peuvent recevoir des paramètres, que vous déclarez en début de script, de cette façon :

```
param (
  [TypeDeDonnées] $variable = "ValeurParDéfaut"
)
```

Les scripts ne peuvent être appelés par un double-clic sur le fichier .ps1 lui-même, pour des raisons de sécurité. De même, en ligne de commande, vous devez préfixer le script par son chemin relatif ou absolu. Par exemple, pour lancer un script appelé monscript.ps1 dans le répertoire courant, saisissez :

```
./monscript.ps1
```

(en Powershell, le forward slash fait aussi office de séparateur de répertoire).

Voici un exemple complet de script. Il permet de détacher toutes les bases du serveur, et de les réattacher. Vous pouvez l'utiliser pour scripter la migration rapide de bases d'un serveur vers un autre. La partie d'attachement illustre la puissance de SMO allié à Powershell, car nous y utilisons des cmdlets Powershell, des objets SMO et des types de données du framework .NET. : nous parcourons le répertoire courant à la recherche de fichiers .mdf, pour chacun nous testons s'il s'agit d'un fichier primaire de base de données détachée. Si c'est le cas, nous collectons la liste des fichiers composant la base de données, et nous attachons la base en spécifiant son nom logique, et les fichiers qui la compose :

```
param (
  [string] $Server = ".\SQL2008",
  [string] $Action = "ATTACH",
  [string] $Path = "d:\sqldata\MSSQL10.SQL2008\MSSQL\DATA\"
)
[System.Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo") | out-null

$srv = New-Object "Microsoft.SqlServer.Management.Smo.Server" $Server

#------------------- DETACH -------------------
if ($Action -eq "DETACH") {
    $dbs = @($srv.Databases)
    foreach ($db in $dbs) {
        if (!$db.IsSystemObject) {
            Write-Host "Détachage de $db..."
            $srv.DetachDatabase($db, $FALSE)
        }
    }
}
#------------------- ATTACH -------------------
elseif ($Action -eq "ATTACH") {
    Set-Location $path
    foreach ($mdf in (gci *.mdf)) {
        if ($srv.IsDetachedPrimaryFile($mdf)) {
            $d = $srv.DetachedDatabaseInfo($mdf)
            $dbname = ($d | Where-Object {$_.Property -eq "database Name"} | Select-Object Value).Value
            $fichiers = New-Object System.Collections.Specialized.StringCollection
            $fichiers.Add($srv.EnumDetachedDatabaseFiles($mdf))
            $fichiers.Add($srv.EnumDetachedLogFiles($mdf))
            $srv.AttachDatabase($dbname, $fichiers)
        }
    }
}
```

V - SQL Server 2008 Powershell

SQL Server 2008 est livré avec une extension Powershell (un snap-in) qui ajoute un PSDrive spécifique à SQL Server, et quelques cmdlets. Le tout est livré sous forme de minishell, c'est-à-dire un shell préconfiguré. Ce PSDrive vous offre une manière bien plus intégrée à Powershell pour gérer les objets SMO, que par l'appel de l'assembly SMO que nous avons vu plus tôt.

Vous pouvez le lancer par un clic droit sur un niveau de l'explorateur d'objets de SQL Server Management Studio (SSMS) 2008, ou directement en invoquant l'exécutable sqlps.exe. Vous pouvez ainsi vous déplacer dans les objets de SQL Server comme dans un système de fichiers :

```
cd \SQL\MON_SERVEUR\MON_INSTANCE\Databases\AdventureWorks2008\tables
dir | where { $_.Schema -eq "Sales" -and $_.Name -eq "Currency" }
dir | where { $_.HasInsteadOfTrigger } | Format-Table Schema, Name -autosize
```

Un dir (Get-ChildItem) dans la hiérarchie du PSDrive SQL Server n'affiche pas les objets système, vous pouvez utiliser l'option -force pour les faire apparaître.

La dernière commande retourne les tables qui ont un déclencheur de type Instead Of, en affichant le schéma et le nom de la table. Vous pouvez aller plus loin, et par exemple utiliser la méthode SMO Script() pour afficher le code source des déclencheurs :

```
dir | where { $_.HasInsteadOfTrigger } |
    foreach { $_.Triggers | where { $_.InsteadOf } |
        foreach { $_.Script() } }
```

la commande ci-dessus peut-être saisie telle quelle. Lorsqu'une commande Powershell n'est pas terminée au retour chariot, l'interpréteur ouvre une ligne supplémentaire présentée par un >>, et attend la suite de la commande. Il lance l'exécution après un double appui sur la touche ENTREE. Pssql ajoute parmi d'autres un cmdlet nommé Invoke-sqlcmd, avec lequel vous pouvez passer une commande T-SQL au contexte du serveur sur lequel vous êtes connecté. Voici un exemple qui retourne le COUNT(*) de toutes les tables dans le schéma Sales (vous devez être dans le répertoire Tables pour que cette commande fonctionne) :

```
dir | where { $_.Schema -eq "Sales" } | foreach { $_.Name; Invoke-sqlcmd -query "SELECT count(*) as cnt
 FROM $_" -SuppressProviderContextWarning } | format-List
```

Vous pouvez également planifier l'exécution de votre code Powershell dans l'agent SQL, qui inclut en 2008 une tâche de type Powershell.

VI - Conclusion

L'alliance de Powershell et de SMO, ou l'utilisation directe du minishell sqlps, permettent d'automatiser l'administration de SQL Server d'une autre manière que par le code T-SQL, et de l'inclure dans une stratégie d'automatisation globale des serveurs Windows, en ayant accès à toutes les ressources des éléments Windows, et des bibliothèques .NET.

VI-A - Références

Références

- **Télécharger Powershell 1**
- **aide sur les objets SMO**
- Bruce Payette, Windows PowerShell in Action, Manning 2007