EBook Gratis

APRENDIZAJE PowerShell

Free unaffiliated eBook created from
Stack Overflow contributors.

#powershell

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: powershell

It is an unofficial and free PowerShell ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official PowerShell.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con PowerShell

Observaciones

Windows PowerShell es un componente de shell y scripts de Windows Management Framework, un marco de administración de automatización / configuración de Microsoft creado en .NET Framework. PowerShell se instala de forma predeterminada en todas las versiones compatibles de los sistemas operativos de cliente y servidor de Windows desde Windows 7 / Windows Server 2008 R2. Powershell se puede actualizar en cualquier momento mediante la descarga de una versión posterior de Windows Management Framework (WMF). La versión "Alpha" de PowerShell 6 es multiplataforma (Windows, Linux y OS X) y debe descargarse e instalarse desde esta página de versión .

Recursos adicionales:

- Documentación de MSDN: https://msdn.microsoft.com/en-us/powershell/scripting/powershell-scripting
- TechNet: https://technet.microsoft.com/en-us/scriptcenter/dd742419.aspx
    - Acerca de las páginas
- Galería de PowerShell: https://www.powershellgallery.com/
- Blog de MSDN: https://blogs.msdn.microsoft.com/powershell/
- Github: https://github.com/powershell
- Sitio de la comunidad: http://powershell.com/cs/

Versiones

Versión	Incluido con Windows	Notas	Fecha de lanzamiento
1.0	XP / Server 2008		2006-11-01
2.0	7 / Server 2008 R2		2009-11-01
3.0	8 / Servidor 2012		2012-08-01
4.0	8.1 / Server 2012 R2		2013-11-01
5.0	10 / Server 2016 Tech Preview		2015-12-16
5.1	Edición de 10 Aniversario / Servidor 2016		2017-01-27

Examples

Instalación o configuración

Windows

PowerShell se incluye con Windows Management Framework. La instalación y configuración no son necesarias en las versiones modernas de Windows.

Las actualizaciones de PowerShell se pueden realizar instalando una versión más reciente de Windows Management Framework.

Otras plataformas

La versión "Beta" de PowerShell 6 se puede instalar en otras plataformas. Los paquetes de instalación están disponibles aquí .

Por ejemplo, PowerShell 6, para Ubuntu 16.04, se publica en los repositorios de paquetes para una fácil instalación (y actualizaciones).

Para instalarlo ejecuta lo siguiente:

```
# Import the public repository GPG keys
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -

# Register the Microsoft Ubuntu repository
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee
/etc/apt/sources.list.d/microsoft.list

# Update apt-get
sudo apt-get update

# Install PowerShell
sudo apt-get install -y powershell

# Start PowerShell
powershell
```

Después de registrar el repositorio de Microsoft una vez como superusuario, a partir de ese momento, solo debe usar `sudo apt-get upgrade powershell` para actualizarlo. Entonces solo ejecuta `powershell`

Permitir que los scripts almacenados en su máquina se ejecuten sin firmar

Por razones de seguridad, PowerShell está configurado de forma predeterminada para permitir solo la ejecución de scripts firmados. La ejecución del siguiente comando le permitirá ejecutar scripts sin firmar (debe ejecutar PowerShell como administrador para hacer esto).

```
Set-ExecutionPolicy RemoteSigned
```

Otra forma de ejecutar los scripts de PowerShell es usar `Bypass` como `ExecutionPolicy` :

```
powershell.exe -ExecutionPolicy Bypass -File "c:\MyScript.ps1"
```

O desde dentro de su consola PowerShell existente o sesión ISE ejecutando:

```
Set-ExecutionPolicy Bypass Process
```

También se puede lograr una solución temporal para la política de ejecución ejecutando el ejecutable de Powershell y pasando cualquier política válida como parámetro `-ExecutionPolicy` . La política está vigente solo durante la vida útil del proceso, por lo que no se necesita acceso administrativo al registro.

```
C:\>powershell -ExecutionPolicy RemoteSigned
```

Existen muchas otras políticas disponibles, y los sitios en línea a menudo lo alientan a usar `Set-ExecutionPolicy Unrestricted` . Esta política permanece en su lugar hasta que se modifica, y reduce la postura de seguridad del sistema. Esto no es aconsejable. Se recomienda el uso de `RemoteSigned` porque permite el código almacenado y escrito localmente, y requiere que el código adquirido de forma remota se firme con un certificado de una raíz confiable.

Además, tenga en cuenta que la Política de grupo puede imponer la Política de ejecución, de modo que incluso si la política se cambia a `Unrestricted` sistema `Unrestricted` , la Política de grupo puede revertir esa configuración en su próximo intervalo de aplicación (generalmente 15 minutos). Puede ver el conjunto de políticas de ejecución en los diversos ámbitos utilizando `Get-ExecutionPolicy -List`

Documentación TechNet:
Set-ExecutionPolicy
about_Execution_Policies

Alias y funciones similares

En PowerShell, hay muchas maneras de lograr el mismo resultado. Esto se puede ilustrar muy bien con el sencillo y familiar ejemplo de `Hello World` :

Utilizando `Write-Host` :

```
Write-Host "Hello World"
```

Usando `Write-Output` :

```
Write-Output 'Hello world'
```

Vale la pena señalar que aunque `Write-Output` y `Write-Host` escriben en la pantalla, hay una sutil diferencia. `Write-Host` escribe *solo* en stdout (es decir, la pantalla de la consola), mientras que `Write-Output` escribe en stdout *AND* en el flujo de salida [éxito] permitiendo la redirección . La redirección (y las secuencias en general) permiten que la salida de un comando se dirija como entrada a otro, incluida la asignación a una variable.

```
> $message = Write-Output "Hello World"
> $message
```

```
"Hello World"
```

Estas funciones similares no son alias, pero pueden producir los mismos resultados si se quiere evitar "contaminar" el flujo de éxito.

`Write-Output` tiene un alias para `Echo` o `Write`

```
Echo 'Hello world'
Write 'Hello world'
```

O, simplemente escribiendo 'Hola mundo'!

```
'Hello world'
```

Todo lo cual resultará con la salida de consola esperada.

```
Hello world
```

Otro ejemplo de alias en PowerShell es la asignación común de los comandos antiguos del símbolo del sistema y los comandos BASH a los cmdlets de PowerShell. Todo lo siguiente produce una lista de directorios del directorio actual.

```
C:\Windows> dir
C:\Windows> ls
C:\Windows> Get-ChildItem
```

Finalmente, ¡puede crear su propio alias con el cmdlet Set-Alias! Como ejemplo, vamos a alisas `Test-NetConnection` , que es esencialmente el equivalente de PowerShell al comando ping del símbolo del sistema, a "ping".

```
Set-Alias -Name ping -Value Test-NetConnection
```

Ahora puede usar `ping` lugar de `Test-NetConnection` ! Tenga en cuenta que si el alias ya está en uso, sobrescribirá la asociación.

El alias estará vivo, hasta que la sesión esté activa. Una vez que cierre la sesión e intente ejecutar el alias que ha creado en su última sesión, no funcionará. Para superar este problema, puede importar todos sus alias de un Excel a su sesión una vez, antes de comenzar su trabajo.

The Pipeline: uso de la salida de un cmdlet de PowerShell

Una de las primeras preguntas que tienen las personas cuando comienzan a usar PowerShell para las secuencias de comandos es cómo manipular la salida de un cmdlet para realizar otra acción.

El símbolo de la tubería `|` se usa al final de un cmdlet para tomar los datos que exporta y enviarlos al siguiente cmdlet. Un ejemplo simple es usar Select-Object para mostrar solo la propiedad Name de un archivo que se muestra desde Get-ChildItem:

```
Get-ChildItem | Select-Object Name
#This may be shortened to:
gci | Select Name
```

El uso más avanzado de la canalización nos permite canalizar la salida de un cmdlet en un bucle foreach:

```
Get-ChildItem | ForEach-Object {
    Copy-Item -Path $_.FullName -destination C:\NewDirectory\
}

#This may be shortened to:
gci | % { Copy $_.FullName C:\NewDirectory\ }
```

Tenga en cuenta que el ejemplo anterior utiliza la variable automática $ _ $ _ es el alias corto de $ PSItem, que es una variable automática que contiene el elemento actual en la tubería.

Comentando

Para comentar sobre los scripts de energía al anteponer la línea con el símbolo `#` (hash)

```
# This is a comment in powershell
Get-ChildItem
```

También puede hacer comentarios de varias líneas usando `<#` y `#>` al principio y al final del comentario respectivamente.

```
<#
This is a
multi-line
comment
#>
Get-ChildItem
```

Métodos de la biblioteca.

Los métodos de la biblioteca .Net estática se pueden llamar desde PowerShell encapsulando el nombre completo de la clase en el tercer corchete y luego llamando al método usando `::`

```
#calling Path.GetFileName()
C:\> [System.IO.Path]::GetFileName('C:\Windows\explorer.exe')
explorer.exe
```

Los métodos estáticos se pueden llamar desde la clase en sí, pero llamar a métodos no estáticos requiere una instancia de la clase .Net (un objeto).

Por ejemplo, no se puede llamar al método AddHours desde la propia clase System.DateTime. Requiere una instancia de la clase:

```
C:\> [System.DateTime]::AddHours(15)
```

```
Method invocation failed because [System.DateTime] does not contain a method named 'AddHours'.
At line:1 char:1
+ [System.DateTime]::AddHours(15)
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : MethodNotFound
```

En este caso, primero creamos un objeto , por ejemplo:

```
C:\> $Object = [System.DateTime]::Now
```

Luego, podemos usar métodos de ese objeto, incluso métodos que no pueden llamarse directamente desde la clase System.DateTime, como el método AddHours:

```
C:\> $Object.AddHours(15)

Monday 12 September 2016 01:51:19
```

Creando objetos

El cmdlet `New-Object` se usa para crear un objeto.

```
# Create a DateTime object and stores the object in variable "$var"
$var = New-Object System.DateTime

# calling constructor with parameters
$sr = New-Object System.IO.StreamReader -ArgumentList "file path"
```

En muchos casos, se creará un nuevo objeto para exportar datos o pasarlo a otro commandlet. Esto se puede hacer así:

```
$newObject = New-Object -TypeName PSObject -Property @{
    ComputerName = "SERVER1"
    Role = "Interface"
    Environment = "Production"
}
```

Hay muchas formas de crear un objeto. El siguiente método es probablemente la forma más rápida y rápida de crear un `PSCustomObject` :

```
$newObject = [PSCustomObject]@{
    ComputerName = 'SERVER1'
    Role         = 'Interface'
    Environment  = 'Production'
}
```

En caso de que ya tenga un objeto, pero solo necesite una o dos propiedades adicionales, simplemente puede agregar esa propiedad utilizando `Select-Object` :

```
Get-ChildItem | Select-Object FullName, Name,
    @{Name='DateTime'; Expression={Get-Date}},
```

```
    @{Name='PropertieName'; Expression={'CustomValue'}}
```

Todos los objetos se pueden almacenar en variables o pasar a la tubería. También puede agregar estos objetos a una colección y luego mostrar los resultados al final.

Las colecciones de objetos funcionan bien con Export-CSV (e Import-CSV). Cada línea del CSV es un objeto, cada columna es una propiedad.

Los comandos de formato convierten los objetos en flujo de texto para su visualización. Evite usar los comandos Format- * hasta el último paso de cualquier procesamiento de datos, para mantener la usabilidad de los objetos.

Lea Empezando con PowerShell en línea:
https://riptutorial.com/es/powershell/topic/822/empezando-con-powershell

Capítulo 2: ¿Cómo descargar el último artefacto de Artifactory usando el script de Powershell (v2.0 o inferior)?

Introducción

Esta documentación explica y proporciona los pasos para descargar el último artefacto de un repositorio de JFrog Artifactory utilizando Powershell Script (v2.0 o inferior).

Examples

Powershell Script para descargar el último artifcat

```
$username = 'user'
$password= 'password'
$DESTINATION = "D:\test\latest.tar.gz"
$client = New-Object System.Net.WebClient
$client.Credentials = new-object System.Net.NetworkCredential($username, $password)
$lastModifiedResponse =
$client.DownloadString('https://domain.org.com/artifactory/api/storage/FOLDER/repo/?lastModified')

[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestModifiedResponse = $serializer.DeserializeObject($lastModifiedResponse)
$downloaUriResponse = $getLatestModifiedResponse.uri
Write-Host $json.uri
$latestArtifcatUrlResponse=$client.DownloadString($downloaUriResponse)
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestArtifact = $serializer.DeserializeObject($latestArtifcatUrlResponse)
Write-Host $getLatestArtifact.downloadUri
$SOURCE=$getLatestArtifact.downloadUri
$client.DownloadFile($SOURCE,$DESTINATION)
```

Lea ¿Cómo descargar el último artefacto de Artifactory usando el script de Powershell (v2.0 o inferior)? en línea: https://riptutorial.com/es/powershell/topic/8883/-como-descargar-el-ultimo-artefacto-de-artifactory-usando-el-script-de-powershell--v2-0-o-inferior--

Capítulo 3: Alias

Observaciones

El sistema de nombres de Powershell tiene reglas bastante estrictas para nombrar cmdlets (plantilla Verb-Noun; consulte [el tema aún no se ha creado] para obtener más información). Pero no es realmente conveniente escribir `Get-ChildItems` cada vez que quiera listar archivos en el directorio de manera interactiva.
Por lo tanto, Powershell permite el uso de accesos directos (alias) en lugar de los nombres de los cmdlets.

Puede escribir `ls` , `dir` o `gci` lugar de `Get-ChildItem` y obtener el mismo resultado. Alias es equivalente a su cmdlet.

Algunos de los alias comunes son:

alias	cmdlet
%, para cada	Para cada objeto
?, dónde	Donde-objeto
gato, gc, tipo	Obtener el contenido
cd, chdir, sl	Escoger localización
cls, claro	Clear-Host
cp, copia, cpi	Copiar el artículo
dir / ls / gci	Get-ChildItem
eco, escribe	Escritura-salida
Florida	Lista de formatos
pie	Formato de tabla
fw	Todo el formato
gc, pwd	Get-Location
gm	Get-Member
iex	Invocar-expresión
ii	Invocar objeto

alias	cmdlet
mv mover	Mover elemento
rm, rmdir, del, borrar, rd, ri	Remover el artículo
dormir	Inicio-sueño
comienzo	Proceso de inicio

En la tabla anterior, puede ver cómo los alias habilitaron los comandos de simulación conocidos de otros entornos (cmd, bash), por lo tanto, una mayor capacidad de descubrimiento.

Examples

Get-Alias

Para listar todos los alias y sus funciones:

```
Get-Alias
```

Para obtener todos los alias para un cmdlet específico:

```
PS C:\> get-alias -Definition Get-ChildItem

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           dir -> Get-ChildItem
Alias           gci -> Get-ChildItem
Alias           ls -> Get-ChildItem
```

Para encontrar alias por coincidencia:

```
PS C:\> get-alias -Name p*

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           popd -> Pop-Location
Alias           proc -> Get-Process
Alias           ps -> Get-Process
Alias           pushd -> Push-Location
Alias           pwd -> Get-Location
```

Set-Alias

Este cmdlet le permite crear nuevos nombres alternativos para salir de los cmdlets

```
PS C:\> Set-Alias -Name proc -Value Get-Process
PS C:\> proc

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName
```

```
-------  ------    -----      ----- -----   ------     --  -- -----------
    292      17    13052      20444 ...19     7.94    620   1 ApplicationFrameHost
....
```

Tenga en cuenta que cualquier alias que cree se conservará solo en la sesión actual. Cuando inicie una nueva sesión, tendrá que volver a crear sus alias. Los Perfiles Powershell (ver [tema aún no creado]) son excelentes para estos propósitos.

Lea Alias en línea: https://riptutorial.com/es/powershell/topic/5287/alias

Capítulo 4: Análisis CSV

Examples

Uso básico de Import-Csv

Dado el siguiente archivo CSV

```
String,DateTime,Integer
First,2016-12-01T12:00:00,30
Second,2015-12-01T12:00:00,20
Third,2015-12-01T12:00:00,20
```

Uno puede importar las filas CSV en objetos de PowerShell usando el comando `Import-Csv`

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows

String DateTime            Integer
------ --------            -------
First  2016-12-01T12:00:00 30
Second 2015-11-03T13:00:00 20
Third  2015-12-05T14:00:00 20

> Write-Host $row[0].String1
Third
```

Importar desde CSV y propiedades de conversión al tipo correcto

De forma predeterminada, `Import-CSV` importa todos los valores como cadenas, por lo que para obtener objetos DateTime e integer, debemos convertirlos o analizarlos.

Usando `Foreach-Object` :

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | ForEach-Object {
    #Cast properties
    $_.DateTime = [datetime]$_.DateTime
    $_.Integer = [int]$_.Integer

    #Output object
    $_
}
```

Usando propiedades calculadas:

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | Select-Object String,
    @{name="DateTime";expression={ [datetime]$_.DateTime }},
    @{name="Integer";expression={ [int]$_.Integer }}
```

Salida:

```
String DateTime            Integer
------ --------            -------
First  01.12.2016 12:00:00      30
Second 03.11.2015 13:00:00      20
Third  05.12.2015 14:00:00      20
```

Lea Análisis CSV en línea: https://riptutorial.com/es/powershell/topic/5691/analisis-csv

Capítulo 5: Anonimizar IP (v4 y v6) en un archivo de texto con Powershell

Introducción

Manipulación de Regex para IPv4 e IPv6 y reemplazo por una dirección IP falsa en un archivo de registro leído

Examples

Anonimizar la dirección IP en el archivo de texto

```
# Read a text file and replace the IPv4 and IPv6 by fake IP Address


# Describe all variables
$SourceFile = "C:\sourcefile.txt"
$IPv4File = "C:\IPV4.txt"
$DestFile = "C:\ANONYM.txt"
$Regex_v4 = "(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})"
$Anonym_v4 = "XXX.XXX.XXX.XXX"
$Regex_v6 = "((([0-9A-Fa-f]{1,4}:){7}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){6}:[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-f]{1,4}:){5}:([0-9A-Fa-f]{1,4}:)?[0-9A-Fa-f]{1,4})|(([0-9A-Fa-
f]{1,4}:){4}:([0-9A-Fa-f]{1,4}:){0,2}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){3}:([0-9A-Fa-
f]{1,4}:){0,3}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){2}:([0-9A-Fa-f]{1,4}:){0,4}[0-9A-Fa-
f]{1,4})|(([0-9A-Fa-f]{1,4}:){6}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-
5])|(1d{2})|(2[0-4]d)|(d{1,2}))b))|(([0-9A-Fa-f]{1,4}:){0,5}:((b((25[0-5])|(1d{2})|(2[0-
4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b))|(::([0-9A-Fa-
f]{1,4}:){0,5}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-
4]d)|(d{1,2}))b))|([0-9A-Fa-f]{1,4}::([0-9A-Fa-f]{1,4}:){0,5}[0-9A-Fa-f]{1,4})|(::([0-9A-Fa-
f]{1,4}:){0,6}[0-9A-Fa-f]{1,4})|(([0-9A-Fa-f]{1,4}:){1,7}:))"
$Anonym_v6 = "YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY"
$SuffixName = "-ANONYM."
$AnonymFile = ($Parts[0] + $SuffixName + $Parts[1])

# Replace matching IPv4 from sourcefile and creating a temp file IPV4.txt
Get-Content $SourceFile | Foreach-Object {$_ -replace $Regex_v4, $Anonym_v4} | Set-Content
$IPv4File

# Replace matching IPv6 from IPV4.txt and creating a temp file ANONYM.txt
Get-Content $IPv4File | Foreach-Object {$_ -replace $Regex_v6, $Anonym_v6} | Set-Content
$DestFile

# Delete temp IPV4.txt file
Remove-Item $IPv4File

# Rename ANONYM.txt in sourcefile-ANONYM.txt
$Parts = $SourceFile.Split(".")
If (Test-Path $AnonymFile)
{
    Remove-Item $AnonymFile
    Rename-Item $DestFile -NewName $AnonymFile
    }
```

```
    Else
    {
    Rename-Item $DestFile -NewName $AnonymFile
}
```

Lea Anonimizar IP (v4 y v6) en un archivo de texto con Powershell en línea: https://riptutorial.com/es/powershell/topic/9171/anonimizar-ip--v4-y-v6--en-un-archivo-de-texto-con-powershell

Capítulo 6: Automatización de infraestructura

Introducción

La automatización de los servicios de gestión de la infraestructura permite reducir el FTE y mejorar el ROI acumulativamente mediante el uso de múltiples herramientas, orquestadores, motor de orquestación, scripts y una interfaz de usuario sencilla.

Examples

Script simple para prueba de integración de caja negra de aplicaciones de consola

Este es un ejemplo simple de cómo puede automatizar las pruebas para una aplicación de consola que interactúa con la entrada estándar y la salida estándar.

La aplicación probada lee y suma cada nueva línea y proporcionará el resultado después de que se proporcione una sola línea blanca. El script del shell de poder escribe "pass" cuando la salida coincide.

```
$process = New-Object System.Diagnostics.Process
$process.StartInfo.FileName = ".\ConsoleApp1.exe"
$process.StartInfo.UseShellExecute = $false
$process.StartInfo.RedirectStandardOutput = $true
$process.StartInfo.RedirectStandardInput = $true
if ( $process.Start() ) {
    # input
    $process.StandardInput.WriteLine("1");
    $process.StandardInput.WriteLine("2");
    $process.StandardInput.WriteLine("3");
    $process.StandardInput.WriteLine();
    $process.StandardInput.WriteLine();
    # output check
    $output = $process.StandardOutput.ReadToEnd()
    if ( $output ) {
        if ( $output.Contains("sum 6") ) {
            Write "pass"
        }
        else {
            Write-Error $output
        }
    }
    $process.WaitForExit()
}
```

Lea Automatización de infraestructura en línea:
https://riptutorial.com/es/powershell/topic/10909/automatizacion-de-infraestructura

Capítulo 7: Ayuda basada en comentarios

Introducción

PowerShell presenta un mecanismo de documentación llamado ayuda basada en comentarios. Permite documentar scripts y funciones con comentarios de código. La ayuda basada en comentarios es la mayor parte del tiempo escrita en bloques de comentarios que contienen varias palabras clave de ayuda. Las palabras clave de ayuda comienzan con puntos e identifican las secciones de ayuda que se mostrarán al ejecutar el cmdlet `Get-Help` .

Examples

Función basada en ayuda de comentarios

```
<#

.SYNOPSIS
    Gets the content of an INI file.

.DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.

.INPUTS
    System.String

.OUTPUTS
    System.Collections.Hashtable

.PARAMETER FilePath
    Specifies the path to the input INI file.

.EXAMPLE
    C:\PS>$IniContent = Get-IniContent -FilePath file.ini
    C:\PS>$IniContent['Section1'].Key1
    Gets the content of file.ini and access Key1 from Section1.

.LINK
    Out-IniFile

#>
function Get-IniContent
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]
        [ValidateNotNullOrEmpty()]
        [ValidateScript({(Test-Path $_) -and ((Get-Item $_).Extension -eq ".ini")})]
        [System.String]$FilePath
    )

    # Initialize output hash table.
    $ini = @{}
```

```
    switch -regex -file $FilePath
    {
        "^\[(.+)\]$" # Section
        {
            $section = $matches[1]
            $ini[$section] = @{}
            $CommentCount = 0
        }
        "^(;.*)$" # Comment
        {
            if( !($section) )
            {
                $section = "No-Section"
                $ini[$section] = @{}
            }
            $value = $matches[1]
            $CommentCount = $CommentCount + 1
            $name = "Comment" + $CommentCount
            $ini[$section][$name] = $value
        }
        "(.+?)\s*=\s*(.*)" # Key
        {
            if( !($section) )
            {
                $section = "No-Section"
                $ini[$section] = @{}
            }
            $name,$value = $matches[1..2]
            $ini[$section][$name] = $value
        }
    }

    return $ini
}
```

La documentación de la función anterior se puede mostrar ejecutando `Get-Help -Name Get-IniContent -Full` :

```
PS C:\Scripts> Get-Help -Name Get-IniContent -Full

NAME
    Get-IniContent

SYNOPSIS
    Gets the content of an INI file.


SYNTAX
    Get-IniContent [-FilePath] <String> [<CommonParameters>]


DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.


PARAMETERS
    -FilePath <String>
        Specifies the path to the input INI file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       true (ByValue)
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    System.String


OUTPUTS
    System.Collections.Hashtable


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>$IniContent = Get-IniContent -FilePath file.ini

    C:\PS>$IniContent['Section1'].Key1
    Gets the content of file.ini and access Key1 from Section1.




RELATED LINKS
    Out-IniFile



PS C:\Scripts>
```

Observe que las palabras clave basadas en comentarios que comienzan con a `.` coincide con las secciones de resultados de `Get-Help` .

Ayuda basada en comentarios del script

```
<#

.SYNOPSIS
    Reads a CSV file and filters it.
```

```
.DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.

.PARAMETER Path
    Specifies the path of the CSV input file.

.INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.

.OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.

.EXAMPLE
    C:\PS> .\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe

#>
Param
(
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $Path,
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $UserName
)

Import-Csv -Path $Path | Where-Object -FilterScript {$_.UserName -eq $UserName}
```

La documentación del script anterior se puede mostrar ejecutando `Get-Help -Name ReadUsersCsv.ps1 -Full` :

```
PS C:\Scripts> Get-Help -Name .\ReadUsersCsv.ps1 -Full

NAME
    C:\Scripts\ReadUsersCsv.ps1

SYNOPSIS
    Reads a CSV file and filters it.


SYNTAX
    C:\Scripts\ReadUsersCsv.ps1 [-Path] <String> [-UserName] <String> [<CommonParameters>]


DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.


PARAMETERS
    -Path <String>
        Specifies the path of the CSV input file.

        Required?                    true
        Position?                    1
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    -UserName <String>
        Specifies the user name that will be used to filter the CSV file.

        Required?                    true
        Position?                    2
        Default value
        Accept pipeline input?       false
        Accept wildcard characters?  false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.


OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.


    -------------------------- EXAMPLE 1 --------------------------

    C:\PS>.\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe




RELATED LINKS



PS C:\Scripts>
```

Lea Ayuda basada en comentarios en línea:
https://riptutorial.com/es/powershell/topic/9530/ayuda-basada-en-comentarios

Capítulo 8: Bucles

Introducción

Un bucle es una secuencia de instrucción (es) que se repite continuamente hasta que se alcanza una determinada condición. Ser capaz de hacer que su programa ejecute repetidamente un bloque de código es una de las tareas más básicas pero útiles en la programación. Un bucle le permite escribir una declaración muy simple para producir un resultado significativamente mayor simplemente por repetición. Si se ha alcanzado la condición, la siguiente instrucción "cae" a la siguiente instrucción secuencial o se ramifica fuera del bucle.

Sintaxis

- para (<Initialization>; <Condition>; <Repetition>) {<Script_Block>}
- <Coleccion> | Foreach-Object {<Script_Block_with _ $ __ as_current_item>}
- foreach (<Item> en <Collection>) {<Script_Block>}
- while (<Condición>) {<Script_Block>}
- do {<Script_Block>} while (<Condition>)
- do {<Script_Block>} hasta (<Condition>)
- <Collection> .foreach ({<Script_Block_with _ $ __ as_current_item>})

Observaciones

Para cada

Hay varias formas de ejecutar un bucle foreach en PowerShell y todas aportan sus propias ventajas y desventajas:

Solución	Ventajas	Desventajas
Declaración de foreach	Lo más rápido. Funciona mejor con colecciones estáticas (almacenadas en una variable).	Sin entrada o salida de tubería
Método ForEach ()	La misma sintaxis de scriptblock que `Foreach-Object` , pero más rápida. Funciona mejor con colecciones estáticas (almacenadas en una variable). Soporta salida de tubería.	No hay soporte para entrada de tubería. Requiere PowerShell 4.0 o superior

Solución	Ventajas	Desventajas
Foreach-Object (cmdlet)	Soporta entrada y salida de ductos. Admite bloques de script de inicio y fin para la inicialización y cierre de conexiones, etc. La solución más flexible.	El más lento

Actuación

```
$foreach = Measure-Command { foreach ($i in (1..1000000)) { $i * $i } }
$foreachmethod = Measure-Command { (1..1000000).ForEach{ $_ * $_ } }
$foreachobject = Measure-Command { (1..1000000) | ForEach-Object { $_ * $_ } }

"Foreach: $($foreach.TotalSeconds)"
"Foreach method: $($foreachmethod.TotalSeconds)"
"ForEach-Object: $($foreachobject.TotalSeconds)"

Example output:

Foreach: 1.9039875
Foreach method: 4.7559563
ForEach-Object: 10.7543821
```

Si bien `Foreach-Object` es el más lento, su compatibilidad con tuberías puede ser útil, ya que le permite procesar los elementos a medida que llegan (al leer un archivo, recibir datos, etc.). Esto puede ser muy útil cuando se trabaja con big data y poca memoria, ya que no es necesario cargar todos los datos en la memoria antes de procesar.

Examples

por

```
for($i = 0; $i -le 5; $i++){
    "$i"
}
```

Un uso típico del bucle for es operar en un subconjunto de los valores en una matriz. En la mayoría de los casos, si desea iterar todos los valores de una matriz, considere usar una instrucción foreach.

Para cada

`ForEach` tiene dos significados diferentes en PowerShell. Una es una palabra clave y la otra es un alias para el cmdlet ForEach-Object . El primero se describe aquí.

Este ejemplo muestra cómo imprimir todos los elementos de una matriz en el host de la consola:

```
$Names = @('Amy', 'Bob', 'Celine', 'David')
```

```
ForEach ($Name in $Names)
{
    Write-Host "Hi, my name is $Name!"
}
```

Este ejemplo muestra cómo capturar la salida de un bucle ForEach:

```
$Numbers = ForEach ($Number in 1..20) {
    $Number # Alternatively, Write-Output $Number
}
```

Al igual que el último ejemplo, este ejemplo, en su lugar, demuestra la creación de una matriz antes de almacenar el bucle:

```
$Numbers = @()
ForEach ($Number in 1..20)
{
    $Numbers += $Number
}
```

Mientras

Un bucle while evaluará una condición y si es verdadero realizará una acción. Mientras la condición se evalúe como verdadera, la acción continuará realizándose.

```
while(condition){
  code_block
}
```

El siguiente ejemplo crea un bucle que contará de 10 a 0

```
$i = 10
while($i -ge 0){
    $i
    $i--
}
```

A diferencia del bucle `Do` While, la condición se evalúa antes de la primera ejecución de la acción. La acción no se realizará si la condición inicial se evalúa como falsa.

Nota: Al evaluar la condición, PowerShell tratará la existencia de un objeto devuelto como verdadero. Esto se puede usar de varias maneras, pero a continuación se muestra un ejemplo para monitorear un proceso. Este ejemplo generará un proceso de bloc de notas y luego dormirá el shell actual mientras ese proceso se esté ejecutando. Cuando cierras manualmente la instancia del bloc de notas, la condición while fallará y el bucle se interrumpirá.

```
Start-Process notepad.exe
while(Get-Process notepad -ErrorAction SilentlyContinue){
  Start-Sleep -Milliseconds 500
}
```

ForEach-Object

El cmdlet `ForEach-Object` funciona de manera similar a la instrucción `foreach` , pero toma su entrada de la canalización.

Uso básico

```
$object | ForEach-Object {
    code_block
}
```

Ejemplo:

```
$names = @("Any","Bob","Celine","David")
$names | ForEach-Object {
    "Hi, my name is $_!"
}
```

`Foreach-Object` tiene dos alias predeterminados, `foreach` y `%` (sintaxis abreviada). El más común es `%` porque `foreach` puede confundirse con la instrucción foreach . Ejemplos:

```
$names | % {
    "Hi, my name is $_!"
}

$names | foreach {
    "Hi, my name is $_!"
}
```

Uso avanzado

`Foreach-Object` destaca de las soluciones alternativas de `foreach` porque es un cmdlet, lo que significa que está diseñado para usar la canalización. Debido a esto, admite tres bloques de secuencias de comandos, como un cmdlet o una función avanzada:

- **Inicio** : se ejecuta una vez antes de recorrer en bucle los elementos que llegan desde la tubería. Generalmente se utiliza para crear funciones para su uso en el bucle, crear variables, abrir conexiones (base de datos, web +), etc.
- **Proceso** : Ejecutado una vez por artículo llegó desde la tubería. "Normal" para cada bloque de código. Este es el valor predeterminado que se utiliza en los ejemplos anteriores cuando no se especifica el parámetro.
- **Fin** : Se ejecuta una vez después de procesar todos los elementos. Usualmente se usa para cerrar conexiones, generar un informe, etc.

Ejemplo:

```
"Any","Bob","Celine","David" | ForEach-Object -Begin {
    $results = @()
} -Process {
```

```
    #Create and store message
    $results += "Hi, my name is $_!"
} -End {
    #Count messages and output
    Write-Host "Total messages: $($results.Count)"
    $results
}
```

Hacer

Los do-loops son útiles cuando siempre desea ejecutar un bloque de código al menos una vez. Un Do-loop evaluará la condición después de ejecutar el bloque de código, a diferencia de un ciclo while que lo hace antes de ejecutar el bloque de código.

Puedes usar do-loops de dos maneras:

- Bucle *mientras* la condición es verdadera:

```
Do {
    code_block
} while (condition)
```

- Bucle *hasta que* la condición sea verdadera, en otras palabras, haga un ciclo mientras la condición sea falsa:

```
Do {
    code_block
} until (condition)
```

Ejemplos reales:

```
$i = 0

Do {
    $i++
    "Number $i"
} while ($i -ne 3)

Do {
    $i++
    "Number $i"
} until ($i -eq 3)
```

Do-While y Do-Until son bucles antónimos. Si el código dentro del mismo, la condición se invertirá. El ejemplo anterior ilustra este comportamiento.

Método ForEach ()

4.0

En lugar del cmdlet `ForEach-Object` , aquí también existe la posibilidad de usar un método `ForEach` directamente en matrices de objetos, de este modo

```
(1..10).ForEach({$_ * $_})
```

o, si lo desea, se pueden omitir los paréntesis alrededor del bloque de script

```
(1..10).ForEach{$_ * $_}
```

Ambos resultarán en la salida de abajo

```
1
4
9
16
25
36
49
64
81
100
```

Continuar

El operador `Continue` trabaja en los `ForEach For` , `ForEach` , `While` y `Do` Omite la iteración actual del bucle, saltando a la parte superior del bucle más interno.

```
$i =0
while ($i -lt 20) {
    $i++
    if ($i -eq 7) { continue }
    Write-Host $I
}
```

Lo anterior dará salida de 1 a 20 a la consola, pero se perderá el número 7.

Nota : Al usar un bucle de tubería, debe usar `return` lugar de `Continue` .

Descanso

El operador de `break` saldrá de un ciclo de programa inmediatamente. Se puede usar en los `ForEach For` , `ForEach` , `While` y `Do` o en una `ForEach Switch` .

```
$i = 0
while ($i -lt 15) {
    $i++
    if ($i -eq 7) {break}
    Write-Host $i
}
```

Lo anterior contará hasta 15 pero se detendrá tan pronto como se alcance 7.

Nota : cuando se usa un bucle de tubería, la `break` se comportará como `continue` . Para simular una `break` en el bucle de canalización, debe incorporar lógica adicional, cmdlet, etc. Es más fácil

seguir con bucles no de tubería si necesita usar `break` .

Etiquetas de rotura

Break también puede llamar a una etiqueta que se colocó delante de la creación de instancias de un bucle:

```
$i = 0
:mainLoop While ($i -lt 15) {
    Write-Host $i -ForegroundColor 'Cyan'
    $j = 0
    While ($j -lt 15) {
        Write-Host $j -ForegroundColor 'Magenta'
        $k = $i*$j
        Write-Host $k -ForegroundColor 'Green'
        if ($k -gt 100) {
            break mainLoop
        }
        $j++
    }
    $i++
}
```

Nota: este código incrementará `$i` a `8` y `$j` a `13` que hará que `$k` igual a `104` . Como `$k` excede de `100` , el código se separará de ambos bucles.

Lea Bucles en línea: https://riptutorial.com/es/powershell/topic/1067/bucles

Capítulo 9: Cambiar la declaración

Introducción

Una declaración de cambio permite que una variable se pruebe para determinar su igualdad frente a una lista de valores. Cada valor se llama un *caso* , y la variable que se está *activando* se comprueba para cada caso de interruptor. Le permite escribir un script que puede elegir entre una serie de opciones, pero sin requerir que escriba una larga serie de sentencias if.

Observaciones

Este tema documenta la ***instrucción de conmutación*** utilizada para bifurcar el flujo del script. No lo confunda con los ***parámetros del interruptor*** que se utilizan en las funciones como indicadores booleanos.

Examples

Interruptor simple

Las declaraciones de cambio comparan un solo valor de prueba con múltiples condiciones y realizan las acciones asociadas para realizar comparaciones exitosas. Puede dar lugar a múltiples coincidencias / acciones.

Dado el siguiente interruptor ...

```
switch($myValue)
{
    'First Condition'    { 'First Action' }
    'Second Condition'   { 'Second Action' }
}
```

`'First Action'` emitirá `'First Action'` si `$myValue` se establece como `'First Condition'` .

`'Section Action'` se emitirá si `$myValue` se establece como `'Second Condition'` .

No se emitirá nada si `$myValue` no coincide con ninguna de las condiciones.

Declaración de cambio con el parámetro Regex

El parámetro `-Regex` permite que las instrucciones de conmutación realicen una comparación de expresiones regulares con las condiciones.

Ejemplo:

```
switch -Regex ('Condition')
{
```

```
  'Con\D+ion'    {'One or more non-digits'}
  'Conditio*$'   {'Zero or more "o"'}
  'C.ndition'    {'Any single char.'}
  '^C\w+ition$'  {'Anchors and one or more word chars.'}
  'Test'         {'No match'}
}
```

Salida:

```
One or more non-digits
Any single char.
Anchors and one or more word chars.
```

Interruptor simple con rotura

La palabra clave `break` se puede usar en las instrucciones de cambio para salir de la declaración antes de evaluar todas las condiciones.

Ejemplo:

```
switch('Condition')
{
  'Condition'
  {
    'First Action'
  }
  'Condition'
  {
    'Second Action'
    break
  }
  'Condition'
  {
    'Third Action'
  }
}
```

Salida:

```
First Action
Second Action
```

Debido a la palabra clave `break` en la segunda acción, la tercera condición no se evalúa.

Cambiar la instrucción con el parámetro comodín

El parámetro `-Wildcard` permite que las instrucciones de conmutación realicen una coincidencia de comodín con las condiciones.

Ejemplo:

```
switch -Wildcard ('Condition')
```

```
{
    'Condition'           {'Normal match'}
    'Condit*'             {'Zero or more wildcard chars.'}
    'C[aoc]ndit[f-l]on'   {'Range and set of chars.'}
    'C?ndition'           {'Single char. wildcard'}
    'Test*'               {'No match'}
}
```

Salida:

```
Normal match
Zero or more wildcard chars.
Range and set of chars.
Single char. wildcard
```

Declaración de cambio con el parámetro exacto

El parámetro `-Exact` que las instrucciones de conmutación realicen una coincidencia exacta, que no `-Exact` entre mayúsculas y minúsculas, con condiciones de cadena.

Ejemplo:

```
switch -Exact ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'   {'Third Action'}
  '^*ondition$' {'Fourth Action'}
  'Conditio*'   {'Fifth Action'}
}
```

Salida:

```
First Action
Second Action
Third Action
```

Las acciones primera a tercera se ejecutan porque sus condiciones asociadas coinciden con la entrada. Las cadenas de expresiones regulares y de comodín en las condiciones cuarta y quinta no coinciden.

Tenga en cuenta que la cuarta condición también coincidiría con la cadena de entrada si se realizara una comparación de expresiones regulares, pero en este caso se ignoró porque no lo es.

Declaración de cambio con el parámetro CaseSensitive

El parámetro `-CaseSensitive` impone instrucciones de conmutación para realizar una comparación exacta y sensible a mayúsculas y minúsculas con las condiciones.

Ejemplo:

```
switch -CaseSensitive ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'   {'Third Action'}
}
```

Salida:

```
Second Action
```

La segunda acción es la única acción ejecutada porque es la única condición que coincide exactamente con la cadena `'Condition'` cuando se tiene en cuenta la distinción entre mayúsculas y minúsculas.

Cambiar instrucción con parámetro de archivo

El parámetro `-file` permite que la instrucción de cambio reciba entrada de un archivo. Cada línea del archivo es evaluada por la instrucción switch.

Archivo de ejemplo `input.txt` :

```
condition
test
```

Ejemplo de instrucción de cambio:

```
switch -file input.txt
{
  'condition' {'First Action'}
  'test'      {'Second Action'}
  'fail'      {'Third Action'}
}
```

Salida:

```
First Action
Second Action
```

Interruptor simple con condición predeterminada

La palabra clave `Default` se usa para ejecutar una acción cuando ninguna otra condición coincide con el valor de entrada.

Ejemplo:

```
switch('Condition')
{
  'Skip Condition'
  {
    'First Action'
```

```
  }
  'Skip This Condition Too'
  {
    'Second Action'
  }
  Default
  {
    'Default Action'
  }
}
```

Salida:

```
Default Action
```

Cambiar declaración con expresiones

Las condiciones también pueden ser expresiones:

```
$myInput = 0

switch($myInput) {
    # because the result of the expression, 4,
    # does not equal our input this block should not be run.
    (2+2)  { 'True. 2 +2 = 4' }

    # because the result of the expression, 0,
    # does equal our input this block should be run.
    (2-2) { 'True. 2-2 = 0' }

    # because our input is greater than -1 and is less than 1
    # the expression evaluates to true and the block should be run.
    { $_ -gt -1 -and $_ -lt 1 } { 'True. Value is 0' }
}

#Output
True. 2-2 = 0
True. Value is 0
```

Lea Cambiar la declaración en línea: https://riptutorial.com/es/powershell/topic/1174/cambiar-la-declaracion

Capítulo 10: Clases de PowerShell

Introducción

Una clase es una plantilla de código de programa extensible para crear objetos, que proporciona valores iniciales para el estado (variables miembro) e implementaciones de comportamiento (funciones o métodos miembro). Una clase es un plano para un objeto. Se utiliza como modelo para definir la estructura de los objetos. Un objeto contiene datos a los que accedemos a través de propiedades y en los que podemos trabajar utilizando métodos. PowerShell 5.0 agregó la capacidad de crear sus propias clases.

Examples

Métodos y propiedades.

```
class Person {
    [string] $FirstName
    [string] $LastName
    [string] Greeting() {
        return "Greetings, {0} {1}!" -f $this.FirstName, $this.LastName
    }
}

$x = [Person]::new()
$x.FirstName = "Jane"
$x.LastName = "Doe"
$greeting = $x.Greeting() # "Greetings, Jane Doe!"
```

Listado de constructores disponibles para una clase

5.0

En PowerShell 5.0+, puede enumerar los constructores disponibles llamando al `new` método estático sin paréntesis.

```
PS> [DateTime]::new

OverloadDefinitions
-------------------
datetime new(long ticks)
datetime new(long ticks, System.DateTimeKind kind)
datetime new(int year, int month, int day)
datetime new(int year, int month, int day, System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
```

```
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar, System.DateTimeKind kind)
```

Esta es la misma técnica que puede usar para enumerar las definiciones de sobrecarga para cualquier método.

```
> 'abc'.CompareTo

OverloadDefinitions
-------------------
int CompareTo(System.Object value)
int CompareTo(string strB)
int IComparable.CompareTo(System.Object obj)
int IComparable[string].CompareTo(string other)
```

Para versiones anteriores puede crear su propia función para listar los constructores disponibles:

```
function Get-Constructor {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline=$true)]
        [type]$type
    )

    Process {
        $type.GetConstructors() |
        Format-Table -Wrap @{
            n="$($type.Name) Constructors"
            e={ ($_.GetParameters() | % { $_.ToString() }) -Join ", " }
        }
    }
}
```

Uso:

```
Get-Constructor System.DateTime
#Or [datetime] | Get-Constructor

DateTime Constructors
---------------------
Int64 ticks
Int64 ticks, System.DateTimeKind kind
Int32 year, Int32 month, Int32 day
Int32 year, Int32 month, Int32 day, System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
```

```
endar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
endar calendar, System.DateTimeKind kind
```

Sobrecarga del constructor

```
class Person {
    [string] $Name
    [int] $Age

    Person([string] $Name) {
        $this.Name = $Name
    }

    Person([string] $Name, [int]$Age) {
        $this.Name = $Name
        $this.Age = $Age
    }
}
```

Obtener todos los miembros de una instancia

```
PS > Get-Member -InputObject $anObjectInstance
```

Esto devolverá a todos los miembros de la instancia de tipo. Aquí es una parte de una salida de muestra para la instancia de cadena

```
   TypeName: System.String

Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone(), System.Object ICloneable.Clone()
CompareTo        Method                int CompareTo(System.Object value), int
CompareTo(string strB), i...
Contains         Method                bool Contains(string value)
CopyTo           Method                void CopyTo(int sourceIndex, char[] destination, int
destinationI...
EndsWith         Method                bool EndsWith(string value), bool EndsWith(string
value, System.S...
Equals           Method                bool Equals(System.Object obj), bool Equals(string
value), bool E...
GetEnumerator    Method                System.CharEnumerator GetEnumerator(),
System.Collections.Generic...
GetHashCode      Method                int GetHashCode()
GetType          Method                type GetType()
...
```

Plantilla de clase básica

```
# Define a class
class TypeName
{
   # Property with validate set
```

```
    [ValidateSet("val1", "Val2")]
    [string] $P1

    # Static property
    static [hashtable] $P2

    # Hidden property does not show as result of Get-Member
    hidden [int] $P3

    # Constructor
    TypeName ([string] $s)
    {
        $this.P1 = $s
    }

    # Static method
    static [void] MemberMethod1([hashtable] $h)
    {
        [TypeName]::P2 = $h
    }

    # Instance method
    [int] MemberMethod2([int] $i)
    {
        $this.P3 = $i
        return $this.P3
    }
}
```

Herencia de la clase padre a la clase infantil

```
class ParentClass
{
    [string] $Message = "Its under the Parent Class"

    [string] GetMessage()
    {
        return ("Message: {0}" -f $this.Message)
    }
}

# Bar extends Foo and inherits its members
class ChildClass : ParentClass
{

}
$Inherit = [ChildClass]::new()
```

SO, **$ Heredado** . El **mensaje** te dará el

> "Está bajo la clase principal"

Lea Clases de PowerShell en línea: https://riptutorial.com/es/powershell/topic/1146/clases-de-powershell

Capítulo 11: Cmdlet Naming

Introducción

Los CmdLets deben nombrarse utilizando un esquema de denominación `<verb>-<noun>` para mejorar el descubrimiento.

Examples

Verbos

Los verbos que se usan para nombrar CmdLets deben nombrarse de los verbos de la lista que se proporciona con `Get-Verb`

Se pueden encontrar más detalles sobre cómo usar verbos en Verbos aprobados para Windows PowerShell

Sustantivos

Los sustantivos siempre deben ser singulares.

Sea consistente con los sustantivos. Por ejemplo, `Find-Package` necesita un proveedor `PackageProvider` nombre sea `PackageProvider` no `ProviderPackage` .

Lea Cmdlet Naming en línea: https://riptutorial.com/es/powershell/topic/8703/cmdlet-naming

Capítulo 12: Codificar / Decodificar URL

Observaciones

La expresión regular utilizada en los ejemplos de *URL de decodificación* se tomó del RFC 2396, Apéndice B: Análisis de una referencia URI con una expresión regular ; Para la posteridad, aquí hay una cita:

> La siguiente línea es la expresión regular para desglosar una referencia URI en sus componentes.
>
> ```
> ^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?
>  12            3  4          5       6  7        8 9
> ```
>
> Los números en la segunda línea de arriba son solo para ayudar a la legibilidad; indican los puntos de referencia para cada subexpresión (es decir, cada paréntesis emparejado). Nos referimos al valor que coincide con la subexpresión como $. Por ejemplo, haciendo coincidir la expresión anterior con
>
> ```
> http://www.ics.uci.edu/pub/ietf/uri/#Related
> ```
>
> los resultados en las siguientes subexpresiones coinciden:
>
> ```
> $1 = http:
> $2 = http
> $3 = //www.ics.uci.edu
> $4 = www.ics.uci.edu
> $5 = /pub/ietf/uri/
> $6 = <undefined>
> $7 = <undefined>
> $8 = #Related
> $9 = Related
> ```

Examples

Inicio rápido: codificación

```
$url1 = [uri]::EscapeDataString("http://test.com?test=my value")
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value

$url2 = [uri]::EscapeUriString("http://test.com?test=my value")
# url2: http://test.com?test=my%20value

# HttpUtility requires at least .NET 1.1 to be installed.
$url3 = [System.Web.HttpUtility]::UrlEncode("http://test.com?test=my value")
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
```

Nota: Más información sobre HTTPUtility .

Inicio rápido: decodificación

Nota: estos ejemplos utilizan las variables creadas en la sección de *Inicio rápido: Codificación* anterior.

```
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[uri]::UnescapeDataString($url1)
# Returns: http://test.com?test=my value

# url2: http://test.com?test=my%20value
[uri]::UnescapeDataString($url2)
# Returns: http://test.com?test=my value

# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[uri]::UnescapeDataString($url3)
# Returns: http://test.com?test=my+value

# Note: There is no `[uri]::UnescapeUriString()`;
#       which makes sense since the `[uri]::UnescapeDataString()`
#       function handles everything it would handle plus more.

# HttpUtility requires at least .NET 1.1 to be installed.
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[System.Web.HttpUtility]::UrlDecode($url1)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url2: http://test.com?test=my%20value
[System.Web.HttpUtility]::UrlDecode($url2)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[System.Web.HttpUtility]::UrlDecode($url3)
# Returns: http://test.com?test=my value
```

Nota: Más información sobre HTTPUtility .

Codificar cadena de consulta con `[uri] :: EscapeDataString ()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [uri]::EscapeDataString($qs.Name)
    $qs_value = [uri]::EscapeDataString($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

Con `[uri]::EscapeDataString()` , notará que el apóstrofe (`'`) no estaba codificado:

> https://example.vertigion.com/foos ? foo2 = complejo% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20bar '% 22 y complejo% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20foo '% 22 = bar2 & foo1 = bar1

Codificar cadena de consulta con `[System.Web.HttpUtility] :: UrlEncode ()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&=+$, bar''"';
    'complex;/?:@&=+$, foo''"'='bar2';
}

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qs_data.GetEnumerator()) {
    $qs_key = [System.Web.HttpUtility]::UrlEncode($qs.Name)
    $qs_value = [System.Web.HttpUtility]::UrlEncode($qs.Value)
    $qs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

Con `[System.Web.HttpUtility]::UrlEncode()` , notará que los espacios se convierten en signos más (`+`) en lugar de `%20` :

> https://example.vertigion.com/foos ? foo2 = complejo% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + barra% 27% 22 y complejo% 3b% 2f% 3f% 3a% 40% 26% 26% 3d% 2b% 24% 2c + foo% 27% 22 = bar2 & foo1 = bar1

Decodificar URL con `[uri] :: UnescapeDataString ()`

Codificado con `[uri]::EscapeDataString()`

Primero, descodificaremos la URL y la cadena de consulta codificada con `[uri]::EscapeDataString()` en el ejemplo anterior:

> https://example.vertigion.com/foos ? foo2 = complejo% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20bar '% 22 y complejo% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20foo '% 22 = bar2 & foo1 = bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3B%2F

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
```

```
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Esto te devuelve `[hashtable]$url_parts` ; que es igual (**Nota:** los *espacios* en las partes complejas son *espacios*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Codificado con `[System.Web.HttpUtility]::UrlEncode()`

Ahora, descodificaremos la URL y la cadena de consulta codificada con `[System.Web.HttpUtility]::UrlEncode()` en el ejemplo anterior:

> https://example.vertigion.com/foos ? foo2 = complejo% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + barra% 27% 22 y complejo% 3b% 2f% 3f% 3a% 40% 26% 26% 3d% 2b% 24% 2c + foo% 27% 22 = bar2 & foo1 = bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
```

```
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Esto le devuelve `[hashtable]$url_parts` , que es igual a (**Nota:** los *espacios* en las partes complejas son *signos más* (`+`) en la primera parte y *espacios* en la segunda parte):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Decodificar URL con `[System.Web.HttpUtility] :: UrlDecode ()`

Codificado con `[uri]::EscapeDataString()`

Primero, descodificaremos la URL y la cadena de consulta codificada con `[uri]::EscapeDataString()` en el ejemplo anterior:

> https://example.vertigion.com/foos ? foo2 = complejo% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20bar '% 22 y complejo% 3B% 2F% 3F% 3A% 40% 26% 3D% 2B% 24% 2C% 20foo '% 22 = bar2 & foo1 = bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar''%22&complex%3B%2

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks
```

```
if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Esto te devuelve `[hashtable]$url_parts` ; que es igual (**Nota:** los *espacios* en las partes complejas son *espacios*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Codificado con `[System.Web.HttpUtility]::UrlEncode()`

Ahora, descodificaremos la URL y la cadena de consulta codificada con `[System.Web.HttpUtility]::UrlEncode()` en el ejemplo anterior:

> https://example.vertigion.com/foos ? foo2 = complejo% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + barra% 27% 22 y complejo% 3b% 2f% 3f% 3a% 40% 26% 26% 3d% 2b% 24% 2c + foo% 27% 22 = bar2 & foo1 = bar1

```
$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks
```

```
if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}
```

Esto te devuelve `[hashtable]$url_parts` ; que es igual (**Nota:** los *espacios* en las partes complejas son *espacios*):

```
PS > $url_parts

Name                           Value
----                           -----
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=ba

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}


PS > $url_parts.QueryStringParts

Name                           Value
----                           -----
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1
```

Lea Codificar / Decodificar URL en línea: https://riptutorial.com/es/powershell/topic/7352/codificar--decodificar-url

Capítulo 13: Comportamiento de retorno en PowerShell

Introducción

Se puede utilizar para salir del ámbito actual, que puede ser una función, un script o un bloque de script. En PowerShell, el resultado de cada declaración se devuelve como resultado, incluso sin una palabra clave de retorno explícita o para indicar que se ha llegado al final del alcance.

Observaciones

Puede leer más sobre la semántica de retorno en la página about_Return en TechNet, o invocando la `get-help return` desde un indicador de PowerShell.

Pregunta (s) de preguntas y respuestas notables con más ejemplos / explicación:

- Valor de retorno de la función en PowerShell
- PowerShell: la función no tiene un valor de retorno adecuado

---

about_return en MSDN lo explica de manera sucinta:

> La palabra clave Return sale de una función, script o bloque de script. Se puede usar para salir de un alcance en un punto específico, para devolver un valor o para indicar que se ha llegado al final del alcance.
>
> Los usuarios que están familiarizados con lenguajes como C o C # pueden querer usar la palabra clave Return para hacer que la lógica de dejar un alcance explícito.
>
> En Windows PowerShell, los resultados de cada declaración se devuelven como resultados, incluso sin una declaración que contenga la palabra clave Return. Los idiomas como C o C # devuelven solo el valor o los valores especificados por la palabra clave Return.

Examples

Salida temprana

```
function earlyexit {
    "Hello"
    return
    "World"
}
```

"Hola" se colocará en el canal de salida, "Mundo" no

Gotcha! Retorno en la tubería

```
get-childitem | foreach-object { if ($_.IsReadOnly) { return } }
```

Los cmdlets de `ForEach-Object` (por ejemplo, `ForEach-Object` , `Where-Object` , etc.) operan en los cierres. La devolución aquí solo se moverá al siguiente elemento en la canalización, no al proceso de salida. Puede usar **break en** lugar de **return** si desea salir del proceso.

```
get-childitem | foreach-object { if ($_.IsReadOnly) { break } }
```

Gotcha! Ignorar la salida no deseada

Inspirado por

- PowerShell: la función no tiene un valor de retorno adecuado

```
function bar {
 [System.Collections.ArrayList]$MyVariable = @()
 $MyVariable.Add("a") | Out-Null
 $MyVariable.Add("b") | Out-Null
 $MyVariable
}
```

`Out-Null` es necesario porque el método .NET `ArrayList.Add` devuelve el número de elementos en la colección después de agregarlos. Si se omite, la tubería habría contenido `1, 2, "a", "b"`

Hay varias formas de omitir resultados no deseados:

```
function bar
{
    # New-Item cmdlet returns information about newly created file/folder
    New-Item "test1.txt" | out-null
    New-Item "test2.txt" > $null
    [void](New-Item "test3.txt")
    $tmp = New-Item "test4.txt"
}
```

Nota: para obtener más información sobre por qué preferir `> $null` , consulte [tema aún no creado].

Vuelve con un valor

(Parafraseado de about_return)

Los siguientes métodos tendrán los mismos valores en la tubería.

```
function foo {
    $a = "Hello"
```

```
    return $a
}

function bar {
    $a = "Hello"
    $a
    return
}

function quux {
    $a = "Hello"
    $a
}
```

Cómo trabajar con funciones devoluciones.

Una función devuelve todo lo que no es capturado por otra cosa.
Si u utilizar la palabra clave **de retorno,** no se ejecutará cada declaración después de la línea de retorno!

Me gusta esto:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    "Start"
    if($ExceptionalReturn){Return "Damn, it didn't work!"}
    New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
    Return "Yes, it worked!"
}
```

Función de prueba
Volverá

- comienzo
- La clave de registro recién creada (esto se debe a que hay algunas declaraciones que crean resultados que puede no estar esperando)
- Sí, funcionó!

Test-Function -ExceptionalReturn devolverá:

- comienzo
- Maldita sea, no funcionó!

Si lo haces así:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
```

```
    )
    . {
       "Start"
        if($ExceptionalReturn)
        {
            $Return = "Damn, it didn't work!"
            Return
        }
        New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
        $Return = "Yes, it worked!"
        Return
    } | Out-Null
    Return $Return
}
```

Función de prueba
Volverá

- Sí, funcionó!

Test-Function -ExceptionalReturn devolverá:

- Maldita sea, no funcionó!

Con este truco, puede controlar el resultado devuelto incluso si no está seguro de lo que escupirá cada declaración.

Funciona así

```
.{<Statements>} | Out-Null
```

la . realiza el siguiente bloque de script incluido en el código
el {} marca el bloque de script
el | Out-Null canaliza cualquier salida inesperada a Out-Null (¡así que se ha ido!)
Debido a que el bloque de secuencias de comandos está incluido, obtiene el mismo ámbito que el resto de la función.
Así que puedes acceder a las variables que fueron creadas dentro del bloque de script.

Lea Comportamiento de retorno en PowerShell en línea:
https://riptutorial.com/es/powershell/topic/4781/comportamiento-de-retorno-en-powershell

Capítulo 14: Comunicación TCP con PowerShell

Examples

Oyente TCP

```
Function Receive-TCPMessage {
    Param (
        [Parameter(Mandatory=$true, Position=0)]
        [ValidateNotNullOrEmpty()]
        [int] $Port
    )
    Process {
        Try {
            # Set up endpoint and start listening
            $endpoint = new-object System.Net.IPEndPoint([ipaddress]::any,$port)
            $listener = new-object System.Net.Sockets.TcpListener $EndPoint
            $listener.start()

            # Wait for an incoming connection
            $data = $listener.AcceptTcpClient()

            # Stream setup
            $stream = $data.GetStream()
            $bytes = New-Object System.Byte[] 1024

            # Read data from stream and write it to host
            while (($i = $stream.Read($bytes,0,$bytes.Length)) -ne 0){
                $EncodedText = New-Object System.Text.ASCIIEncoding
                $data = $EncodedText.GetString($bytes,0, $i)
                Write-Output $data
            }

            # Close TCP connection and stop listening
            $stream.close()
            $listener.stop()
        }
        Catch {
            "Receive Message failed with: `n" + $Error[0]
        }
    }
}
```

Comience a escuchar con lo siguiente y capture cualquier mensaje en la variable `$msg` :

```
$msg = Receive-TCPMessage -Port 29800
```

TCP Sender

```
Function Send-TCPMessage {
    Param (
```

```
            [Parameter(Mandatory=$true, Position=0)]
            [ValidateNotNullOrEmpty()]
            [string]
            $EndPoint
        ,
            [Parameter(Mandatory=$true, Position=1)]
            [int]
            $Port
        ,
            [Parameter(Mandatory=$true, Position=2)]
            [string]
            $Message
    )
    Process {
        # Setup connection
        $IP = [System.Net.Dns]::GetHostAddresses($EndPoint)
        $Address = [System.Net.IPAddress]::Parse($IP)
        $Socket = New-Object System.Net.Sockets.TCPClient($Address,$Port)

        # Setup stream wrtier
        $Stream = $Socket.GetStream()
        $Writer = New-Object System.IO.StreamWriter($Stream)

        # Write message to stream
        $Message | % {
            $Writer.WriteLine($_)
            $Writer.Flush()
        }

        # Close connection and stream
        $Stream.Close()
        $Socket.Close()
    }
}
```

Enviar un mensaje con:

```
Send-TCPMessage -Port 29800 -Endpoint 192.168.0.1 -message "My first TCP message !"
```

Nota : los mensajes TCP pueden estar bloqueados por el firewall de su software o por cualquier firewall externo que esté tratando de atravesar. Asegúrese de que el puerto TCP que configuró en el comando anterior esté abierto y que haya configurado el servicio de escucha en el mismo puerto.

Lea Comunicación TCP con PowerShell en línea:
https://riptutorial.com/es/powershell/topic/5125/comunicacion-tcp-con-powershell

Capítulo 15: Comunicarse con APIs RESTful

Introducción

REST significa "Representational State Transfer" (Transferencia estatal representativa) (a veces deletreado "ReST"). Se basa en un protocolo de comunicaciones sin caché, cliente-servidor, que se puede almacenar en caché y, en su mayoría, se utiliza el protocolo HTTP. Se utiliza principalmente para crear servicios web que sean livianos, mantenibles y escalables. Un servicio basado en REST se denomina servicio RESTful y las API que se están utilizando son API RESTful. En PowerShell, *`Invoke-RestMethod`* se utiliza para tratar con ellos.

Examples

Utilice Slack.com entrantes Webhooks

Defina su carga útil para enviar posibles datos más complejos.

```
$Payload = @{ text="test string"; username="testuser" }
```

Use el cmdlet `ConvertTo-Json` y `Invoke-RestMethod` para ejecutar la llamada

```
Invoke-RestMethod -Uri "https://hooks.slack.com/services/yourwebhookstring" -Method Post -Body
(ConvertTo-Json $Payload)
```

Publicar mensaje en hipChat

```
$params = @{
    Uri = "https://your.hipchat.com/v2/room/934419/notification?auth_token=???"
    Method = "POST"
    Body = @{
        color = 'yellow'
        message = "This is a test message!"
        notify = $false
        message_format = "text"
    } | ConvertTo-Json
    ContentType = 'application/json'
}

Invoke-RestMethod @params
```

Uso de REST con objetos de PowerShell para obtener y colocar datos individuales

OBTENGA sus datos REST y almacénelos en un objeto de PowerShell:

```
$Post = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

Modifique sus datos:

```
$Post.title = "New Title"
```

PONER los datos REST de vuelta

```
$Json = $Post | ConvertTo-Json
Invoke-RestMethod -Method Put -Uri "http://jsonplaceholder.typicode.com/posts/1" -Body $Json -
ContentType 'application/json'
```

Usando REST con objetos de PowerShell para GET y POST muchos artículos

OBTENGA sus datos REST y almacénelos en un objeto de PowerShell:

```
$Users = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/users"
```

Modificar muchos elementos en sus datos:

```
$Users[0].name = "John Smith"
$Users[0].email = "John.Smith@example.com"
$Users[1].name = "Jane Smith"
$Users[1].email = "Jane.Smith@example.com"
```

POST todos los datos REST de vuelta:

```
$Json = $Users | ConvertTo-Json
Invoke-RestMethod -Method Post -Uri "http://jsonplaceholder.typicode.com/users" -Body $Json -
ContentType 'application/json'
```

Uso de REST con PowerShell para eliminar elementos

Identifique el elemento que se va a eliminar y elimínelo:

```
Invoke-RestMethod -Method Delete -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

Lea Comunicarse con APIs RESTful en línea:
https://riptutorial.com/es/powershell/topic/3869/comunicarse-con-apis-restful

Capítulo 16: Configuración del estado deseado

Examples

Ejemplo simple - Habilitar WindowsFeature

```
configuration EnableIISFeature
{
    node localhost
    {
        WindowsFeature IIS
        {
            Ensure = “Present”
            Name = “Web-Server”
        }
    }
}
```

Si ejecuta esta configuración en Powershell (EnableIISFeature), producirá un archivo localhost.mof. Esta es la configuración "compilada" que puede ejecutar en una máquina.

Para probar la configuración de DSC en su host local, simplemente puede invocar lo siguiente:

```
Start-DscConfiguration -ComputerName localhost -Wait
```

Iniciando DSC (mof) en una máquina remota

Iniciar un DSC en una máquina remota es casi igual de simple. Suponiendo que ya haya configurado la comunicación remota de Powershell (o que haya habilitado WSMAN).

```
$remoteComputer = "myserver.somedomain.com"
$cred = (Get-Credential)
Start-DSCConfiguration -ServerName $remoteComputer -Credential $cred -Verbose
```

Nb: asumiendo que ha compilado una configuración para su nodo en su máquina local (y que el archivo myserver.somedomain.com.mof está presente antes de comenzar la configuración)

Importando psd1 (archivo de datos) en una variable local

A veces puede ser útil probar sus archivos de datos de Powershell e iterar a través de los nodos y servidores.

Powershell 5 (WMF5) agregó esta pequeña característica para hacer esto llamado Import-PowerShellDataFile.

Ejemplo:

```
$data = Import-PowerShellDataFile -path .\MydataFile.psd1
$data.AllNodes
```

Lista de recursos DSC disponibles

Para listar los recursos DSC disponibles en su nodo de autoría:

```
 Get-DscResource
```

Esto mostrará una lista de todos los recursos para todos los módulos instalados (que están en su PSModulePath) en su nodo de creación.

Para enumerar todos los recursos DSC disponibles que se pueden encontrar en las fuentes en línea (PSGallery ++) en WMF 5:

```
Find-DSCResource
```

Importando recursos para usar en DSC

Antes de poder utilizar un recurso en una configuración, debe importarlo explícitamente. El solo hecho de tenerlo instalado en su computadora, no le permitirá usar el recurso de manera implícita.

Importe un recurso utilizando Import-DscResource.

Ejemplo que muestra cómo importar el recurso PSDesiredStateConfiguration y el recurso File.

```
Configuration InstallPreReqs
{
   param(); # params to DSC goes here.

   Import-DscResource PSDesiredStateConfiguration

   File CheckForTmpFolder {
        Type = 'Directory'
        DestinationPath = 'C:\Tmp'
        Ensure = "Present"
    }
 }
```

Nota : para que los recursos de DSC funcionen, debe tener los módulos instalados en las máquinas de destino al ejecutar la configuración. Si no los tiene instalados, la configuración fallará.

Lea Configuración del estado deseado en línea:
https://riptutorial.com/es/powershell/topic/5662/configuracion-del-estado-deseado

Capítulo 17: Conjuntos de parámetros

Introducción

Los conjuntos de parámetros se utilizan para limitar la posible combinación de parámetros, o para imponer el uso de parámetros cuando se seleccionan 1 o más parámetros.

Los ejemplos explicarán el uso y la razón de un conjunto de parámetros.

Examples

Conjuntos de parámetros simples

```
function myFunction
{
    param(
        # If parameter 'a' is used, then 'c' is mandatory
        # If parameter 'b' is used, then 'c' is optional, but allowed
        # You can use parameter 'c' in combination with either 'a' or 'b'
        # 'a' and 'b' cannot be used together

        [parameter(ParameterSetName="AandC", mandatory=$true)]
        [switch]$a,
        [parameter(ParameterSetName="BandC", mandatory=$true)]
        [switch]$b,
        [parameter(ParameterSetName="AandC", mandatory=$true)]
        [parameter(ParameterSetName="BandC", mandatory=$false)]
        [switch]$c
    )
    # $PSCmdlet.ParameterSetName can be used to check which parameter set was used
    Write-Host $PSCmdlet.ParameterSetName
}

# Valid syntaxes
myFunction -a -c
# => "Parameter set : AandC"
myFunction -b -c
# => "Parameter set : BandC"
myFunction -b
# => "Parameter set : BandC"

# Invalid syntaxes
myFunction -a -b
# => "Parameter set cannot be resolved using the specified named parameters."
myFunction -a
# => "Supply values for the following parameters:
#    c:"
```

Conjunto de parámetros para imponer el uso de un parámetro cuando se selecciona otro.

Cuando desee, por ejemplo, imponer el uso del parámetro Contraseña si se proporciona el

parámetro Usuario. (y viceversa)

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Credentials", mandatory=$false)]
        [String]$Computername = "LocalHost",
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [String]$User,
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [SecureString]$Password
    )

    #Do something
}

# This will not work he will ask for user and password
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -ComputerName

# This will not work he will ask for password
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -User
```

Conjunto de parámetros para limitar la combinación de parámetros.

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomeThingToDo,
        [Parameter(ParameterSetName="Silently", mandatory=$false)]
        [Switch]$Silently,
        [Parameter(ParameterSetName="Loudly", mandatory=$false)]
        [Switch]$Loudly
    )

    #Do something
}

# This will not work because you can not use the combination Silently and Loudly
Do-Something -SomeThingToDo 'get-help about_Functions_Advanced' -Silently -Loudly
```

Lea Conjuntos de parámetros en línea: https://riptutorial.com/es/powershell/topic/6598/conjuntos-de-parametros

Capítulo 18: consultas de powershell sql

Introducción

Al revisar este documento, puede aprender a utilizar consultas SQL con powershell.

Parámetros

ít	Descripción
$ ServerInstance	Aquí tenemos que mencionar la instancia en la que está presente la base de datos.
$ Base de datos	Aquí tenemos que mencionar la base de datos en la que está presente la tabla.
$ Consulta	Aquí tenemos a la consulta que queremos ejecutar en SQ.
$ Nombre de usuario y $ Contraseña	Nombre de usuario y contraseña que tienen acceso en la base de datos

Observaciones

Puede usar la siguiente función si en caso de no poder importar el módulo SQLPS

```
function Import-Xls
{

    [CmdletBinding(SupportsShouldProcess=$true)]

    Param(
        [parameter(
            mandatory=$true,
            position=1,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true)]
        [String[]]
        $Path,

        [parameter(mandatory=$false)]
        $Worksheet = 1,

        [parameter(mandatory=$false)]
        [switch]
        $Force
    )

    Begin
    {
        function GetTempFileName($extension)
```

```
        {
            $temp = [io.path]::GetTempFileName();
            $params = @{
                Path = $temp;
                Destination = $temp + $extension;
                Confirm = $false;
                Verbose = $VerbosePreference;
            }
            Move-Item @params;
            $temp += $extension;
            return $temp;
        }

        # since an extension like .xls can have multiple formats, this
        # will need to be changed
        #
        $xlFileFormats = @{
            # single worksheet formats
            '.csv'  = 6;        # 6, 22, 23, 24
            '.dbf'  = 11;       # 7, 8, 11
            '.dif'  = 9;        #
            '.prn'  = 36;       #
            '.slk'  = 2;        # 2, 10
            '.wk1'  = 31;       # 5, 30, 31
            '.wk3'  = 32;       # 15, 32
            '.wk4'  = 38;       #
            '.wks'  = 4;        #
            '.xlw'  = 35;       #

            # multiple worksheet formats
            '.xls'  = -4143;    # -4143, 1, 16, 18, 29, 33, 39, 43
            '.xlsb' = 50;       #
            '.xlsm' = 52;       #
            '.xlsx' = 51;       #
            '.xml'  = 46;       #
            '.ods'  = 60;       #
        }

        $xl = New-Object -ComObject Excel.Application;
        $xl.DisplayAlerts = $false;
        $xl.Visible = $false;
    }

    Process
    {
        $Path | ForEach-Object {

            if ($Force -or $psCmdlet.ShouldProcess($_)) {

                $fileExist = Test-Path $_

                if (-not $fileExist) {
                    Write-Error "Error: $_ does not exist" -Category ResourceUnavailable;

                } else {
                    # create temporary .csv file from excel file and import .csv
                    #
                    $_ = (Resolve-Path $_).toString();
                    $wb = $xl.Workbooks.Add($_);
                    if ($?) {
                        $csvTemp = GetTempFileName(".csv");
```

```
                        $ws = $wb.Worksheets.Item($Worksheet);
                        $ws.SaveAs($csvTemp, $xlFileFormats[".csv"]);
                        $wb.Close($false);
                        Remove-Variable -Name ('ws', 'wb') -Confirm:$false;
                        Import-Csv $csvTemp;
                        Remove-Item $csvTemp -Confirm:$false -Verbose:$VerbosePreference;
                    }
                }
            }
        }
    }

    End
    {
        $xl.Quit();
        Remove-Variable -name xl -Confirm:$false;
        [gc]::Collect();
    }
}
```

Examples

Ejemplo de ejemplo

Para consultar todos los datos de la tabla *MachineName* podemos usar el comando como el siguiente.

$ Query = "Select * from MachineName"

$ Inst = "ServerInstance"

$ DbName = "DatabaseName

$ UID = "ID de usuario"

$ Contraseña = "Contraseña"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

SQLQuery

Para consultar todos los datos de la tabla *MachineName* podemos usar el comando como el siguiente.

$ Query = "Select * from MachineName"

$ Inst = "ServerInstance"

$ DbName = "DatabaseName

$ UID = "ID de usuario"

$ Contraseña = "Contraseña"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password
$Password
```

Lea consultas de powershell sql en línea:
https://riptutorial.com/es/powershell/topic/8217/consultas-de-powershell-sql

Capítulo 19: Convenciones de nombres

Examples

Funciones

```
Get-User()
```

- Use el patrón *Verb-Sustantivo* al nombrar una función.
- Verbo implica una acción, por ejemplo, `Get` , `Set` , `New` , `Read` , `Write` y muchos más. Ver verbos aprobados .
- El nombre debe ser singular, incluso si actúa sobre varios elementos. `Get-User()` puede devolver uno o varios usuarios.
- Utilice el caso de Pascal tanto para el verbo como para el sustantivo. Por ejemplo, `Get-UserLogin()`

Lea Convenciones de nombres en línea:
https://riptutorial.com/es/powershell/topic/9714/convenciones-de-nombres

Capítulo 20: Creación de recursos basados en clases DSC

Introducción

A partir de la versión 5.0 de PowerShell, puede usar las definiciones de clase de PowerShell para crear recursos de configuración de estado deseado (DSC).

Para ayudar en la creación del recurso DSC, hay un `[DscResource()]` que se aplica a la definición de la clase y un `[DscProperty()]` para designar propiedades configurables por el usuario del recurso DSC.

Observaciones

Un recurso DSC basado en clase debe:

- Ser decorado con el `[DscResource()]`
- Define un método `Test()` que devuelve `[bool]`
- Defina un método `Get()` que devuelva su propio tipo de objeto (por ejemplo, `[Ticket]`)
- Define un método `Set()` que devuelve `[void]`
- Al menos una propiedad `Key` DSC

Después de crear un recurso DSC de PowerShell basado en clase, debe ser "exportado" desde un módulo, usando un archivo de manifiesto de módulo (.psd1). Dentro del manifiesto del módulo, la clave de tabla hash `DscResourcesToExport` se utiliza para declarar una matriz de recursos DSC (nombres de clase) para "exportar" desde el módulo. Esto permite a los consumidores del módulo DSC "ver" los recursos basados en clase dentro del módulo.

Examples

Crear una clase de esqueleto de recursos DSC

```
[DscResource()]
class File {
}
```

Este ejemplo muestra cómo construir la sección externa de una clase de PowerShell, que declara un recurso DSC. Aún debe completar los contenidos de la definición de clase.

DSC Resource Skeleton con propiedad clave

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
```

```
  [string] $TicketId
}
```

Un recurso DSC debe declarar al menos una propiedad clave. La propiedad clave es lo que identifica de forma única el recurso de otros recursos. Por ejemplo, digamos que está creando un recurso DSC que representa un ticket en un sistema de tickets. Cada boleto estaría representado de manera única con un ID de boleto.

Cada propiedad que se expondrá al *usuario* del recurso DSC debe estar decorada con el `[DscProperty()]` . Este atributo acepta un parámetro `key` , para indicar que la propiedad es un atributo clave para el recurso DSC.

Recurso DSC con propiedad obligatoria

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId

  [DscProperty(Mandatory)]
  [string] $Subject
}
```

Al crear un recurso de DSC, a menudo encontrará que no todas las propiedades deben ser obligatorias. Sin embargo, hay algunas propiedades centrales que querrá asegurarse de que estén configuradas por el usuario del recurso DSC. Utiliza el parámetro `Mandatory` del `[DscResource()]` para declarar una propiedad según lo requiera el usuario del recurso DSC.

En el ejemplo anterior, hemos agregado una propiedad del `Subject` a un recurso de `Ticket` , que representa un boleto único en un sistema de boleto, y lo hemos designado como una propiedad `Mandatory` .

Recurso DSC con métodos requeridos

```
[DscResource()]
class Ticket {
  [DscProperty(Key)]
  [string] $TicketId

  # The subject line of the ticket
  [DscProperty(Mandatory)]
  [string] $Subject

  # Get / Set if ticket should be open or closed
  [DscProperty(Mandatory)]
  [string] $TicketState

  [void] Set() {
    # Create or update the resource
  }

  [Ticket] Get() {
    # Return the resource's current state as an object
```

```
    $TicketState = [Ticket]::new()
    return $TicketState
  }

  [bool] Test() {
    # Return $true if desired state is met
    # Return $false if desired state is not met
    return $false
  }
}
```

Este es un recurso de DSC completo que muestra todos los requisitos básicos para construir un recurso válido. Las implementaciones del método no están completas, pero se proporcionan con la intención de mostrar la estructura básica.

Lea Creación de recursos basados en clases DSC en línea:
https://riptutorial.com/es/powershell/topic/8733/creacion-de-recursos-basados---en-clases-dsc

Capítulo 21: Cumplimiento de requisitos previos de script

Sintaxis

- #Requiere -Version <N> [. <n>]
- #Requiere –PSSnapin <PSSnapin-Name> [-Version <N> [. <n>]]
- #Requiere -Módulos {<Módulo-Nombre> | <Hashtable>}
- #Requires –ShellId <ShellId>
- #Requiere -RunAsAdministrator

Observaciones

`#requires` declaración `#requires` se puede colocar en cualquier línea de la secuencia de comandos (no tiene que ser la primera línea), pero debe ser la primera declaración en esa línea.

Se pueden usar múltiples declaraciones `#requires` en un script.

Para obtener más información, consulte la documentación oficial en Technet - about_about_Requires .

Examples

Exigir la versión mínima del servidor de PowerShell

```
#requires –version 4
```

Después de intentar ejecutar esta secuencia de comandos en una versión inferior, verá este mensaje de error

> . \ script.ps1: el script 'script.ps1' no se puede ejecutar porque contenía una declaración "#requires" en la línea 1 para Windows PowerShell versión 5.0. La versión requerida por el script no coincide con la versión actualmente en ejecución de Windows PowerShell versión 2.0.

Exigir la ejecución de la secuencia de comandos como administrador

4.0

```
#requires –RunAsAdministrator
```

Después de intentar ejecutar este script sin privilegios de administrador, verá este mensaje de error

. \ script.ps1: El script 'script.ps1' no se puede ejecutar porque contiene una declaración "#requires" para ejecutarse como Administrador. La sesión actual de Windows PowerShell no se está ejecutando como administrador. Inicie Windows PowerShell con la opción Ejecutar como administrador y luego intente ejecutar el script nuevamente.

Lea Cumplimiento de requisitos previos de script en línea:
https://riptutorial.com/es/powershell/topic/5637/cumplimiento-de-requisitos-previos-de-script

Capítulo 22: Ejecutando ejecutables

Examples

Aplicaciones de consola

```
PS> console_app.exe
PS> & console_app.exe
PS> Start-Process console_app.exe
```

Aplicaciones GUI

```
PS> gui_app.exe (1)
PS> & gui_app.exe (2)
PS> & gui_app.exe | Out-Null (3)
PS> Start-Process gui_app.exe (4)
PS> Start-Process gui_app.exe -Wait (5)
```

Las aplicaciones GUI se inician en un proceso diferente e inmediatamente devolverán el control al host de PowerShell. A veces, necesita que la aplicación termine de procesarse antes de que se ejecute la siguiente instrucción de PowerShell. Esto se puede lograr canalizando la salida de la aplicación a $ null (3) o utilizando el proceso de inicio con el interruptor -Wait (5).

Transmisiones de consola

```
PS> $ErrorActionPreference = "Continue" (1)
PS> & console_app.exe *>&1 | % { $_ } (2)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.ErrorRecord] } (3)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.WarningRecord] } (4)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.VerboseRecord] } (5)
PS> & console_app.exe *>&1 (6)
PS> & console_app.exe 2>&1 (7)
```

La secuencia 2 contiene objetos System.Management.Automation.ErrorRecord. Tenga en cuenta que algunas aplicaciones como git.exe utilizan la "secuencia de error" con fines informativos, que no son necesariamente errores en absoluto. En este caso, es mejor mirar el código de salida para determinar si la secuencia de error debe interpretarse como errores.

PowerShell entiende estas secuencias: Salida, Error, Advertencia, Verbose, Depurar, Progreso. Las aplicaciones nativas comúnmente usan solo estas secuencias: Salida, Error, Advertencia.

En PowerShell 5, todos los flujos pueden redirigirse al flujo de salida / éxito estándar (6).

En versiones anteriores de PowerShell, solo las secuencias específicas se pueden redirigir a la secuencia de salida / éxito estándar (7). En este ejemplo, el "flujo de error" se redireccionará al flujo de salida.

Códigos de salida

```
PS> $LastExitCode
PS> $?
PS> $Error[0]
```

Estas son variables incorporadas de PowerShell que proporcionan información adicional sobre el error más reciente. `$LastExitCode` es el código de salida final de la última aplicación nativa que se ejecutó. `$?` y `$Error[0]` es el último registro de error generado por PowerShell.

Lea Ejecutando ejecutables en línea: https://riptutorial.com/es/powershell/topic/7707/ejecutando-ejecutables

Capítulo 23: Enviando email

Introducción

Una técnica útil para los administradores de Exchange Server es poder enviar mensajes de correo electrónico a través de SMTP desde PowerShell. Dependiendo de la versión de PowerShell instalada en su computadora o servidor, hay varias formas de enviar correos electrónicos a través de powershell. Existe una opción de cmdlet nativa que es simple y fácil de usar. Utiliza el cmdlet ***Send-MailMessage*** .

Parámetros

Parámetro	Detalles
Adjuntos <String []>	Ruta y nombres de archivos de los archivos que se adjuntarán al mensaje. Las rutas y los nombres de archivo se pueden canalizar a Send-MailMessage.
Bcc <String []>	Direcciones de correo electrónico que reciben una copia de un mensaje de correo electrónico pero no aparecen como destinatarios en el mensaje. Ingrese los nombres (opcional) y la dirección de correo electrónico (requerido), como Nombre a alguien@ejemplo.com o alguien@ejemplo.com.
Cuerpo <String_>	Contenido del mensaje de correo electrónico.
BodyAsHtml	Indica que el contenido está en formato HTML.
Cc <String []>	Direcciones de correo electrónico que reciben una copia de un mensaje de correo electrónico. Ingrese los nombres (opcional) y la dirección de correo electrónico (requerido), como Nombre a alguien@ejemplo.com o alguien@ejemplo.com.
Credencial	Especifica una cuenta de usuario que tiene permiso para enviar mensajes desde la dirección de correo electrónico especificada. El valor predeterminado es el usuario actual. Ingrese un nombre como Usuario o Dominio \ Usuario, o ingrese un objeto PSCredential.
EntregaNotificaciónOpción	Especifica las opciones de notificación de entrega para el mensaje de correo electrónico. Se pueden especificar múltiples valores. Las notificaciones de entrega se envían en el mensaje a la dirección especificada en el parámetro Para. Valores aceptables: ninguno, OnSuccess, OnFailure, Delay, Never.
Codificación	Codificación para el cuerpo y sujeto. Valores aceptables: ASCII,

Parámetro	Detalles
	UTF8, UTF7, UTF32, Unicode, BigEndianUnicode, Predeterminado, OEM.
Desde	Direcciones de correo electrónico desde las que se envía el correo. Ingrese los nombres (opcional) y la dirección de correo electrónico (requiera), como Nombre a alguien@ejemplo.com o alguien@ejemplo.com.
Puerto	Puerto alternativo en el servidor SMTP. El valor predeterminado es 25. Disponible desde Windows PowerShell 3.0.
Prioridad	Prioridad del mensaje de correo electrónico. Valores aceptables: Normal, Alto, Bajo.
Servidor SMTP	Nombre del servidor SMTP que envía el mensaje de correo electrónico. El valor predeterminado es el valor de la variable $ PSEmailServer.
Tema	Asunto del mensaje de correo electrónico.
A	Direcciones de correo electrónico a las que se envía el correo. Ingrese los nombres (opcional) y la dirección de correo electrónico (requerido), como Nombre alguien@ejemplo.com o alguien@ejemplo.com
UseSsl	Utiliza el protocolo Secure Sockets Layer (SSL) para establecer una conexión con la computadora remota para enviar correo

Examples

Mensaje simple de envío de correo

```
Send-MailMessage -From sender@bar.com -Subject "Email Subject" -To receiver@bar.com -
SmtpServer smtp.com
```

Send-MailMessage con parámetros predefinidos

```
$parameters = @{
    From = 'from@bar.com'
    To = 'to@bar.com'
    Subject = 'Email Subject'
    Attachments =  @('C:\files\samplefile1.txt','C:\files\samplefile2.txt')
    BCC = 'bcc@bar.com'
    Body = 'Email body'
    BodyAsHTML = $False
    CC = 'cc@bar.com'
    Credential = Get-Credential
```

```
    DeliveryNotificationOption = 'onSuccess'
    Encoding = 'UTF8'
    Port = '25'
    Priority = 'High'
    SmtpServer = 'smtp.com'
    UseSSL = $True
}

# Notice: Splatting requires @ instead of $ in front of variable name
Send-MailMessage @parameters
```

SMTPClient - Correo con archivo .txt en el mensaje del cuerpo

```
# Define the txt which will be in the email body
$Txt_File = "c:\file.txt"

function Send_mail {
    #Define Email settings
    $EmailFrom = "source@domain.com"
    $EmailTo = "destination@domain.com"
    $Txt_Body = Get-Content $Txt_File -RAW
    $Body = $Body_Custom + $Txt_Body
    $Subject = "Email Subject"
    $SMTPServer = "smtpserver.domain.com"
    $SMTPClient = New-Object Net.Mail.SmtpClient($SmtpServer, 25)
    $SMTPClient.EnableSsl = $false
    $SMTPClient.Send($EmailFrom, $EmailTo, $Subject, $Body)

}

$Body_Custom = "This is what contain file.txt : "

Send_mail
```

Lea Enviando email en línea: https://riptutorial.com/es/powershell/topic/2040/enviando-email

Capítulo 24: Expresiones regulares

Sintaxis

- 'text' -match 'RegExPattern'
- 'text' -replace 'RegExPattern', 'newvalue'
- [regex] :: Match ("text", "pattern") #Single match
- [regex] :: Coincidencias ("texto", "patrón") #Múltiples coincidencias
- [regex] :: Reemplazar ("texto", "patrón", "nuevo valor")
- [regex] :: Replace ("text", "pattern", {param ($ m)}) #MatchEvaluator
- [regex] :: Escape ("input") #Escape caracteres especiales

Examples

Partido individual

Puede determinar rápidamente si un texto incluye un patrón específico utilizando Regex. Hay varias formas de trabajar con Regex en PowerShell.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Usando el operador -Match

Para determinar si una cadena coincide con un patrón utilizando el operador `-matches` de `-matches` , use la sintaxis `'input' -match 'pattern'` . Esto devolverá `true` o `false` dependiendo del resultado de la búsqueda. Si hubo coincidencia, puede ver la coincidencia y los grupos (si están definidos en el patrón) accediendo a la variable `$Matches` .

```
> $text -match $pattern
True

> $Matches

Name Value
---- -----
0    (a)
```

También puede usar `-match` para filtrar a través de una matriz de cadenas y solo devolver las

cadenas que contienen una coincidencia.

```
> $textarray = @"
This is (a) sample
text, this is
a (sample text)
"@ -split "`n"

> $textarray -match $pattern
This is (a) sample
a (sample text)
```

2.0

Usando Select-String

PowerShell 2.0 introdujo un nuevo cmdlet para buscar en texto usando expresiones regulares. Devuelve un objeto `MatchInfo` por entrada de texto que contiene una coincidencia. Puedes acceder a sus propiedades para encontrar grupos coincidentes, etc.

```
> $m = Select-String -InputObject $text -Pattern $pattern

> $m

This is (a) sample
text, this is
a (sample text)

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
             text, this is
             a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a)}
```

Como `-match` , `Select-String` también se puede usar para filtrar a través de una matriz de cadenas al canalizar una matriz a la misma. Crea un objeto `MatchInfo` por cadena que incluye una coincidencia.

```
> $textarray | Select-String -Pattern $pattern

This is (a) sample
a (sample text)

#You can also access the matches, groups etc.
> $textarray | Select-String -Pattern $pattern | fl *
```

```
IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a)}

IgnoreCase : True
LineNumber : 3
Line       : a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(sample text)}
```

`Select-String` también puede buscar usando un patrón de texto normal (sin `-SimpleMatch` regulares) agregando el interruptor `-SimpleMatch` .

Usando [RegEx] :: Match ()

También puede usar el método estático `Match()` disponible en la clase .NET `[RegEx]` .

```
> [regex]::Match($text,$pattern)

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

> [regex]::Match($text,$pattern) | Select-Object -ExpandProperty Value
(a)
```

Reemplazar

Una tarea común para las expresiones regulares es reemplazar el texto que coincida con un patrón con un nuevo valor.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Text wrapped in ()
$pattern = '\(.*?\)'

#Replace matches with:
$newvalue = 'test'
```

Usando el operador de reemplazo

El operador `-replace` en PowerShell se puede usar para reemplazar el texto que coincida con un patrón con un nuevo valor usando la sintaxis `'input' -replace 'pattern', 'newvalue'`.

```
> $text -replace $pattern, $newvalue
This is test sample
text, this is
a test
```

Usando el método [RegEx] :: Replace ()

El reemplazo de coincidencias también se puede hacer usando el método `Replace()` en la clase `[RegEx]` .NET.

```
[regex]::Replace($text, $pattern, 'test')
This is test sample
text, this is
a test
```

Reemplace el texto con un valor dinámico utilizando un MatchEvalutor

A veces es necesario reemplazar un valor que coincida con un patrón con un nuevo valor que se base en esa coincidencia específica, lo que hace imposible predecir el nuevo valor. Para estos tipos de escenarios, un `MatchEvaluator` puede ser muy útil.

En PowerShell, un `MatchEvaluator` es tan simple como un bloque de script con un solo parámetro que contiene un objeto `Match` para la coincidencia actual. La salida de la acción será el nuevo valor para esa coincidencia específica. `MatchEvalutor` se puede usar con el método estático `[Regex]::Replace()`.

Ejemplo : reemplazar el texto dentro de `()` con su longitud

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '(?<=\().*?(?=\))'

$MatchEvalutor = {
    param($match)

    #Replace content with length of content
    $match.Value.Length
```

```
}
```

Salida:

```
> [regex]::Replace($text, $pattern, $MatchEvalutor)

This is 1 sample
text, this is
a 11
```

Ejemplo: hacer una `sample` en mayúsculas

```
#Sample pattern: "Sample"
$pattern = 'sample'

$MatchEvalutor = {
    param($match)

    #Return match in upper-case
    $match.Value.ToUpper()

}
```

Salida:

```
> [regex]::Replace($text, $pattern, $MatchEvalutor)

This is (a) SAMPLE
text, this is
a (SAMPLE text)
```

Escapar de personajes especiales.

Un patrón de expresiones regulares usa muchos caracteres especiales para describir un patrón. Ej., `.` significa "cualquier carácter", `+` es "uno o más", etc.

Para utilizar estos caracteres, como a `.` , `+` etc., en un patrón, debes escapar de ellos para eliminar su significado especial. Esto se hace usando el carácter de escape que es una barra invertida `\` en regex. Ejemplo: Para buscar `+` , usarías el patrón `\+` .

Puede ser difícil recordar todos los caracteres especiales en expresiones regulares, por lo que para escapar de cada carácter especial en una cadena que desea buscar, puede usar el método `[RegEx]::Escape("input")` .

```
> [regex]::Escape("(foo)")
\(foo\)

> [regex]::Escape("1+1.2=2.2")
1\+1\.2=2\.2
```

Múltiples partidos

Hay varias formas de encontrar todas las coincidencias para un patrón en un texto.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Usando Select-String

Puede encontrar todas las coincidencias (coincidencia global) agregando el interruptor `-AllMatches` a `Select-String` .

```
> $m = Select-String -InputObject $text -Pattern $pattern -AllMatches

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
             text, this is
             a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a), (sample text)}

#List all matches
> $m.Matches

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

Groups   : {(sample text)}
Success  : True
Captures : {(sample text)}
Index    : 37
Length   : 13
Value    : (sample text)

#Get matched text
> $m.Matches | Select-Object -ExpandProperty Value
(a)
(sample text)
```

Usando [RegEx] :: Coincidencias ()

El método `Matches()` en .NET `[regex] -class también se puede usar para hacer una búsqueda global de múltiples coincidencias.

```
> [regex]::Matches($text,$pattern)

Groups   : {(a)}
Success  : True
Captures : {(a)}
Index    : 8
Length   : 3
Value    : (a)

Groups   : {(sample text)}
Success  : True
Captures : {(sample text)}
Index    : 37
Length   : 13
Value    : (sample text)

> [regex]::Matches($text,$pattern) | Select-Object -ExpandProperty Value

(a)
(sample text)
```

Lea Expresiones regulares en línea: https://riptutorial.com/es/powershell/topic/6674/expresiones-regulares

Capítulo 25: Firma de Scripts

Observaciones

Firmar un script hará que sus scripts cumplan con todas las políticas de ejecución en PowerShell y garantizará la integridad de un script. Los scripts firmados no se ejecutarán si se han modificado después de firmarlos.

La firma de scripts requiere un certificado de firma de código. Recomendaciones:

- Pruebas / scripts personales (no compartidos): certificado de una autoridad certificada de confianza (interna o de terceros) **O** un certificado autofirmado.
- Organización interna compartida: Certificado de la autoridad certificada de confianza (interna o de terceros)
- Organización externa compartida: Certificado de la autoridad certificadora de terceros de confianza

Lea más en about_Signing @ TechNet

Políticas de ejecución

PowerShell tiene políticas de ejecución configurables que controlan qué condiciones se requieren para que se ejecute un script o una configuración. Se puede establecer una política de ejecución para múltiples ámbitos; Computadora, usuario actual y proceso actual. **Las políticas de ejecución se pueden omitir fácilmente y no están diseñadas para restringir a los usuarios, sino protegerlos de violar las políticas de firma involuntariamente.**

Las políticas disponibles son:

Ajuste	Descripción
Restringido	No se permiten scripts
AllSigned	Todos los scripts necesitan ser firmados
RemoteSigned	Todos los scripts locales permitidos; solo scripts remotos firmados
Irrestricto	No hay requisitos. Todos los scripts están permitidos, pero avisarán antes de ejecutar los scripts descargados de Internet.
Derivación	Se permiten todos los scripts y no se muestran advertencias
Indefinido	Eliminar la política de ejecución actual para el ámbito actual. Utiliza la política de los padres. Si todas las políticas no están definidas, se utilizará la restricción.

Puede modificar las políticas de ejecución actuales utilizando `Set-ExecutionPolicy` -cmdlet, Group Policy o el parámetro `-ExecutionPolicy` cuando `-ExecutionPolicy` un proceso `powershell.exe` .

Lea más en about_Execution_Policies @ TechNet

Examples

Firmando un guion

La firma de un script se realiza mediante el uso del `Set-AuthenticodeSignature` -cmdlet y un certificado de firma de código.

```
#Get the first available personal code-signing certificate for the logged on user
$cert = @(Get-ChildItem -Path Cert:\CurrentUser\My -CodeSigningCert)[0]

#Sign script using certificate
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1
```

También puede leer un certificado de un archivo `.pfx` usando:

```
$cert = Get-PfxCertificate -FilePath "C:\MyCodeSigningCert.pfx"
```

El guión será válido hasta que caduque el certificado. Si utiliza un servidor de marca de tiempo durante la firma, la secuencia de comandos seguirá siendo válida después de que caduque el certificado. También es útil agregar la cadena de confianza para el certificado (incluida la autoridad raíz) para ayudar a la mayoría de las computadoras a confiar en el certificado utilizado para firmar el script.

```
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1 -IncludeChain All -
TimeStampServer "http://timestamp.verisign.com/scripts/timstamp.dll"
```

Se recomienda utilizar un servidor de marca de tiempo de un proveedor de certificados de confianza como Verisign, Comodo, Thawte, etc.

Cambiando la política de ejecución usando Set-ExecutionPolicy

Para cambiar la política de ejecución para el ámbito predeterminado (LocalMachine), use:

```
Set-ExecutionPolicy AllSigned
```

Para cambiar la política para un alcance específico, use:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy AllSigned
```

Puede suprimir las indicaciones agregando el interruptor `-Force` .

Omitir la política de ejecución para un solo script

A menudo es posible que necesite ejecutar un script sin firmar que no cumpla con la política de ejecución actual. Una forma fácil de hacer esto es omitir la política de ejecución para ese proceso único. Ejemplo:

```
powershell.exe -ExecutionPolicy Bypass -File C:\MyUnsignedScript.ps1
```

O puedes usar la taquigrafía:

```
powershell -ep Bypass C:\MyUnsignedScript.ps1
```

Otras políticas de ejecución:

Política	Descripción
`AllSigned`	Solo se pueden ejecutar scripts firmados por un editor de confianza.
`Bypass`	Sin restricciones; Se pueden ejecutar todos los scripts de Windows PowerShell.
`Default`	Normalmente `RemoteSigned` , pero se controla a través de ActiveDirectory
`RemoteSigned`	Los scripts descargados deben estar firmados por un editor de confianza antes de que puedan ejecutarse.
`Restricted`	No se pueden ejecutar scripts. Windows PowerShell solo se puede utilizar en modo interactivo.
`Undefined`	N / A
`Unrestricted *`	Similar a la `bypass`

`Unrestricted*` Advertencia: *si ejecuta una secuencia de comandos sin firmar que se descargó de Internet, se le solicitará permiso antes de que se ejecute.*

Más información disponible aquí .

Obtener la política de ejecución actual.

Obtención de la política de ejecución efectiva para la sesión actual:

```
PS> Get-ExecutionPolicy
RemoteSigned
```

Listar todas las políticas de ejecución efectivas para la sesión actual:

```
PS> Get-ExecutionPolicy -List

        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser       Undefined
 LocalMachine    RemoteSigned
```

Listar la política de ejecución para un alcance específico, ej. proceso:

```
PS> Get-ExecutionPolicy -Scope Process
Undefined
```

Obteniendo la firma de un script firmado

Obtenga información sobre la firma Authenticode de un script firmado utilizando el `Get-AuthenticodeSignature` -cmdlet:

```
Get-AuthenticodeSignature .\MyScript.ps1 | Format-List *
```

Creación de un certificado de firma de código autofirmado para pruebas

Al firmar scripts personales o al probar la firma de código, puede ser útil crear un certificado de firma de código autofirmado.

5.0

A partir de PowerShell 5.0, puede generar un certificado de firma de código autofirmado mediante el certificado de `New-SelfSignedCertificate` Self:

```
New-SelfSignedCertificate -FriendlyName "StackOverflow Example Code Signing" -
CertStoreLocation Cert:\CurrentUser\My -Subject "SO User" -Type CodeSigningCert
```

En versiones anteriores, puede crear un certificado autofirmado utilizando la herramienta `makecert.exe` que se encuentra en .NET Framework SDK y Windows SDK.

Solo las computadoras que tengan instalado el certificado confiarán en un certificado autofirmado. Para los scripts que se compartirán, se recomienda un certificado de una autoridad de certificados de confianza (interna o de terceros de confianza).

Lea Firma de Scripts en línea: https://riptutorial.com/es/powershell/topic/5670/firma-de-scripts

Capítulo 26: Flujos de trabajo de PowerShell

Introducción

PowerShell Workflow es una característica que se introdujo a partir de la versión 3.0 de PowerShell. Las definiciones de flujo de trabajo son muy similares a las definiciones de funciones de PowerShell, sin embargo, se ejecutan dentro del entorno de Windows Workflow Foundation, en lugar de hacerlo directamente en el motor de PowerShell.

Con el motor de flujo de trabajo se incluyen varias características exclusivas "fuera de la caja", sobre todo, la persistencia del trabajo.

Observaciones

La función de flujo de trabajo de PowerShell se admite exclusivamente en la plataforma Microsoft Windows, en PowerShell Desktop Edition. PowerShell Core Edition, que es compatible con Linux, Mac y Windows, no es compatible con la función de flujo de trabajo de PowerShell.

Al crear un flujo de trabajo de PowerShell, tenga en cuenta que los flujos de trabajo llaman a las actividades, no a los cmdlets. Aún puede llamar a los cmdlets desde un flujo de trabajo de PowerShell, pero el motor de flujo de trabajo envolverá implícitamente la invocación del cmdlet en una actividad de `InlineScript` . También puede ajustar explícitamente el código dentro de la actividad `InlineScript` , que ejecuta el código de PowerShell; de forma predeterminada, la actividad `InlineScript` se ejecuta en un proceso independiente y devuelve el resultado al flujo de trabajo de la llamada.

Examples

Ejemplo de flujo de trabajo simple

```
workflow DoSomeWork {
  Get-Process -Name notepad | Stop-Process
}
```

Este es un ejemplo básico de una definición de flujo de trabajo de PowerShell.

Flujo de trabajo con parámetros de entrada

Al igual que las funciones de PowerShell, los flujos de trabajo pueden aceptar parámetros de entrada. Los parámetros de entrada pueden vincularse opcionalmente a un tipo de datos específico, como una cadena, entero, etc. Use la palabra clave `param` estándar para definir un bloque de parámetros de entrada, directamente después de la declaración del flujo de trabajo.

```
workflow DoSomeWork {
  param (
```

```
    [string[]] $ComputerName
  )
  Get-Process -ComputerName $ComputerName
}

DoSomeWork -ComputerName server01, server02, server03
```

Ejecutar flujo de trabajo como un trabajo en segundo plano

Los flujos de trabajo de PowerShell están equipados de forma inherente con la capacidad de ejecutarse como un trabajo en segundo plano. Para llamar a un flujo de trabajo como un trabajo en segundo plano de PowerShell, use el parámetro `-AsJob` cuando invoque el flujo de trabajo.

```
workflow DoSomeWork {
  Get-Process -ComputerName server01
  Get-Process -ComputerName server02
  Get-Process -ComputerName server03
}

DoSomeWork -AsJob
```

Agregar un bloque paralelo a un flujo de trabajo

```
workflow DoSomeWork {
  parallel {
    Get-Process -ComputerName server01
    Get-Process -ComputerName server02
    Get-Process -ComputerName server03
  }
}
```

Una de las características únicas de PowerShell Workflow es la capacidad de definir un bloque de actividades como paralelo. Para usar esta función, use la palabra clave `parallel` dentro de su flujo de trabajo.

Llamar en paralelo a las actividades del flujo de trabajo puede ayudar a mejorar el rendimiento de su flujo de trabajo.

Lea Flujos de trabajo de PowerShell en línea:
https://riptutorial.com/es/powershell/topic/8745/flujos-de-trabajo-de-powershell

Capítulo 27: Funciones de PowerShell

Introducción

Una función es básicamente un bloque de código con nombre. Cuando llama al nombre de la función, se ejecuta el bloque de script dentro de esa función. Es una lista de declaraciones de PowerShell que tiene un nombre que usted asigna. Cuando ejecuta una función, escribe el nombre de la función. Es un método para ahorrar tiempo al abordar tareas repetitivas. Los formatos de PowerShell en tres partes: la palabra clave 'Función', seguida de un Nombre, finalmente, la carga útil que contiene el bloque de script, que está encerrada entre corchetes de estilo de paréntesis.

Examples

Función simple sin parámetros

Este es un ejemplo de una función que devuelve una cadena. En el ejemplo, la función se llama en una declaración asignando un valor a una variable. El valor en este caso es el valor de retorno de la función.

```
function Get-Greeting{
    "Hello World"
}

# Invoking the function
$greeting = Get-Greeting

# demonstrate output
$greeting
Get-Greeting
```

`function` declara que el siguiente código es una función.

`Get-Greeting` es el nombre de la función. En cualquier momento en que esa función deba utilizarse en el script, se puede llamar a la función invocándola por su nombre.

`{ ... }` es el bloque de script que ejecuta la función.

Si el código anterior se ejecuta en el ISE, los resultados serían algo así como:

```
Hello World
Hello World
```

Parametros basicos

Una función se puede definir con parámetros usando el bloque param:

```
function Write-Greeting {
    param(
        [Parameter(Mandatory,Position=0)]
        [String]$name,
        [Parameter(Mandatory,Position=1)]
        [Int]$age
    )
    "Hello $name, you are $age years old."
}
```

O usando la sintaxis de la función simple:

```
function Write-Greeting ($name, $age) {
    "Hello $name, you are $age years old."
}
```

Nota: los parámetros de conversión no son necesarios en ninguno de los tipos de definición de parámetros.

La sintaxis de función simple (SFS) tiene capacidades muy limitadas en comparación con el bloque param.
Aunque puede definir los parámetros que se expondrán dentro de la función, no puede especificar atributos de parámetros , utilizar la validación de parámetros , incluir `[CmdletBinding()]` , con SFS (y esta es una lista no exhaustiva).

Las funciones pueden ser invocadas con parámetros ordenados o nombrados.

El orden de los parámetros en la invocación coincide con el orden de la declaración en el encabezado de la función (por defecto), o se puede especificar usando el atributo de parámetro de `Position` (como se muestra en el ejemplo de la función avanzada, más arriba).

```
$greeting = Write-Greeting "Jim" 82
```

Alternativamente, esta función puede ser invocada con parámetros nombrados.

```
$greeting = Write-Greeting -name "Bob" -age 82
```

Parámetros obligatorios

Los parámetros de una función se pueden marcar como obligatorios.

```
function Get-Greeting{
    param
    (
        [Parameter(Mandatory=$true)]$name
    )
    "Hello World $name"
}
```

Si la función se invoca sin un valor, la línea de comando solicitará el valor:

```
$greeting = Get-Greeting

cmdlet Get-Greeting at command pipeline position 1
Supply values for the following parameters:
name:
```

Función avanzada

Esta es una copia del fragmento de función avanzada del ISE de Powershell. Básicamente, esta es una plantilla para muchas de las cosas que puede usar con funciones avanzadas en Powershell. Puntos clave a tener en cuenta:

- Integración get-help: el principio de la función contiene un bloque de comentarios que está configurado para ser leído por el cmdlet get-help. El bloque de funciones se puede ubicar al final, si se desea.
- cmdletbinding - la función se comportará como un cmdlet
- parámetros
- conjuntos de parámetros

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
.INPUTS
   Inputs to this cmdlet (if any)
.OUTPUTS
   Output from this cmdlet (if any)
.NOTES
   General notes
.COMPONENT
   The component this cmdlet belongs to
.ROLE
   The role this cmdlet belongs to
.FUNCTIONALITY
   The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
                   SupportsShouldProcess=$true,
                   PositionalBinding=$false,
                   HelpUri = 'http://www.microsoft.com/',
                   ConfirmImpact='Medium')]
    [Alias()]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                    ValueFromPipeline=$true,
                    ValueFromPipelineByPropertyName=$true,
```

```
                    ValueFromRemainingArguments=$false,
                    Position=0,
                    ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,

        # Param2 help description
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,

        # Param3 help description
        [Parameter(ParameterSetName='Another Parameter Set')]
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    Begin
    {
    }
    Process
    {
        if ($pscmdlet.ShouldProcess("Target", "Operation"))
        {
        }
    }
    End
    {
    }
}
```

Validación de parámetros

Hay varias formas de validar la entrada de parámetros, en PowerShell.

En lugar de escribir código dentro de funciones o scripts para validar los valores de los parámetros, estos atributos de parámetro se lanzarán si se pasan valores no válidos.

ValidateSet

A veces necesitamos restringir los valores posibles que un parámetro puede aceptar. Digamos que queremos permitir solo rojo, verde y azul para el parámetro `$Color` en un script o función.

Podemos usar el atributo de parámetro `ValidateSet` para restringir esto. Tiene la ventaja adicional de permitir que se complete la pestaña al configurar este argumento (en algunos entornos).

```
param(
    [ValidateSet('red','green','blue',IgnoreCase)]
    [string]$Color
)
```

También puede especificar `IgnoreCase` para desactivar la sensibilidad a las mayúsculas.

Validar Rango

Este método de validación de parámetros toma un valor Int32 mínimo y máximo, y requiere que el parámetro esté dentro de ese rango.

```
param(
    [ValidateRange(0,120)]
    [Int]$Age
)
```

ValidatePattern

Este método de validación de parámetros acepta parámetros que coinciden con el patrón de expresiones regulares especificado.

```
param(
    [ValidatePattern("\w{4-6}\d{2}")]
    [string]$UserName
)
```

ValidateLength

Este método de validación de parámetros prueba la longitud de la cadena pasada.

```
param(
    [ValidateLength(0,15)]
    [String]$PhoneNumber
)
```

ValidateCount

Este método de validación de parámetros prueba la cantidad de argumentos pasados, por ejemplo, una matriz de cadenas.

```
param(
    [ValidateCount(1,5)]
    [String[]]$ComputerName
)
```

ValidateScript

Finalmente, el método ValidateScript es extraordinariamente flexible, tomando un bloque de script y evaluándolo usando $ _ para representar el argumento pasado. Luego pasa el argumento si el resultado es $ verdadero (incluyendo cualquier resultado como válido).

Esto puede ser usado para probar que un archivo existe:

```
param(
    [ValidateScript({Test-Path $_})]
    [IO.FileInfo]$Path
)
```

Para comprobar que un usuario existe en AD:

```
param(
    [ValidateScript({Get-ADUser $_})]
    [String]$UserName
)
```

Y casi cualquier otra cosa que puedas escribir (ya que no está restringida a oneliners):

```
param(
    [ValidateScript({
        $AnHourAgo = (Get-Date).AddHours(-1)
        if ($_ -lt $AnHourAgo.AddMinutes(5) -and $_ -gt $AnHourAgo.AddMinutes(-5)) {
            $true
        } else {
            throw "That's not within five minutes. Try again."
        }
    })]
    [String]$TimeAboutAnHourAgo
)
```

Lea Funciones de PowerShell en línea: https://riptutorial.com/es/powershell/topic/1673/funciones-de-powershell

Capítulo 28: Gestión de paquetes

Introducción

PowerShell Package Management le permite encontrar, instalar, actualizar y desinstalar PowerShell Modules y otros paquetes.

PowerShellGallery.com es la fuente predeterminada de los módulos de PowerShell. También puede navegar por el sitio en busca de paquetes disponibles, ordenar y obtener una vista previa del código.

Examples

Encuentra un módulo PowerShell usando un patrón

Para encontrar un módulo que termine con `DSC`

```
Find-Module -Name *DSC
```

Crear la estructura predeterminada del módulo de PowerShell

Si, por algún motivo, se elimina el repositorio de PowerShell predeterminado, `PSGallery` . Necesitarás crearlo. Este es el comando.

```
Register-PSRepository -Default
```

Encuentra un módulo por nombre

```
Find-Module -Name <Name>
```

Instala un módulo por nombre

```
Install-Module -Name <name>
```

Desinstalar un módulo mi nombre y versión.

```
Uninstall-Module -Name <Name> -RequiredVersion <Version>
```

Actualizar un módulo por nombre

```
Update-Module -Name <Name>
```

Lea Gestión de paquetes en línea: https://riptutorial.com/es/powershell/topic/8698/gestion-de-paquetes

Capítulo 29: GUI en Powershell

Examples

GUI de WPF para cmdlet Get-Service

```
Add-Type -AssemblyName PresentationFramework

[xml]$XAMLWindow = '
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Height="Auto"
    SizeToContent="WidthAndHeight"
    Title="Get-Service">
    <ScrollViewer Padding="10,10,10,0" ScrollViewer.VerticalScrollBarVisibility="Disabled">
        <StackPanel>
            <StackPanel Orientation="Horizontal">
                <Label Margin="10,10,0,10">ComputerName:</Label>
                <TextBox Name="Input" Margin="10" Width="250px"></TextBox>
            </StackPanel>
            <DockPanel>
                <Button Name="ButtonGetService" Content="Get-Service" Margin="10"
Width="150px" IsEnabled="false"/>
                <Button Name="ButtonClose" Content="Close" HorizontalAlignment="Right"
Margin="10" Width="50px"/>
            </DockPanel>
        </StackPanel>
    </ScrollViewer >
</Window>
'

# Create the Window Object
$Reader=(New-Object System.Xml.XmlNodeReader $XAMLWindow)
$Window=[Windows.Markup.XamlReader]::Load( $Reader )

# TextChanged Event Handler for Input
$TextboxInput = $Window.FindName("Input")
$TextboxInput.add_TextChanged.Invoke({
    $ComputerName = $TextboxInput.Text
    $ButtonGetService.IsEnabled = $ComputerName -ne ''
})

# Click Event Handler for ButtonClose
$ButtonClose = $Window.FindName("ButtonClose")
$ButtonClose.add_Click.Invoke({
    $Window.Close();
})

# Click Event Handler for ButtonGetService
$ButtonGetService = $Window.FindName("ButtonGetService")
$ButtonGetService.add_Click.Invoke({
    $ComputerName = $TextboxInput.text.Trim()
    try{
        Get-Service -ComputerName $computerName | Out-GridView -Title "Get-Service on
$ComputerName"
    }catch{
```

```
[System.Windows.MessageBox]::Show($_.exception.message,"Error",[System.Windows.MessageBoxButton]::OK,[S

    }
})

# Open the Window
$Window.ShowDialog() | Out-Null
```

Esto crea una ventana de diálogo que le permite al usuario seleccionar un nombre de computadora, luego mostrará una tabla de servicios y sus estados en esa computadora. Este ejemplo usa WPF en lugar de Windows Forms.

Lea GUI en Powershell en línea: https://riptutorial.com/es/powershell/topic/7141/gui-en-powershell

Capítulo 30: HashTables

Introducción

Una tabla hash es una estructura que asigna claves a valores. Ver tabla de hash para más detalles.

Observaciones

Un concepto importante que se basa en tablas hash es Splatting . Es muy útil para hacer un gran número de llamadas con parámetros repetitivos.

Examples

Creación de una tabla hash

Ejemplo de crear una HashTable vacía:

```
$hashTable = @{}
```

Ejemplo de creación de una tabla hash con datos:

```
$hashTable = @{
    Name1 = 'Value'
    Name2 = 'Value'
    Name3 = 'Value3'
}
```

Acceda a un valor de tabla hash por clave.

Un ejemplo de cómo definir una tabla hash y acceder a un valor mediante la clave

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$hashTable.Key1
#output
Value1
```

Un ejemplo de acceso a una clave con caracteres no válidos para un nombre de propiedad:

```
$hashTable = @{
    'Key 1' = 'Value3'
    Key2 = 'Value4'
}
$hashTable.'Key 1'
```

```
#Output
Value3
```

Buceando sobre una mesa de hash

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}

foreach($key in $hashTable.Keys)
{
    $value = $hashTable.$key
    Write-Output "$key : $value"
}
#Output
Key1 : Value1
Key2 : Value2
```

Agregar un par de valores clave a una tabla hash existente

Un ejemplo, para agregar una clave "Clave2" con un valor de "Valor2" a la tabla hash, usando el operador de suma:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable += @{Key2 = 'Value2'}
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
Key2                           Value2
```

Un ejemplo, para agregar una clave "Clave2" con un valor de "Valor2" a la tabla hash usando el método Agregar:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable.Add("Key2", "Value2")
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
Key2                           Value2
```

Enumeración a través de claves y pares clave-valor

Enumerar a través de claves

```
foreach ($key in $var1.Keys) {
    $value = $var1[$key]
    # or
    $value = $var1.$key
}
```

Enumeración a través de pares clave-valor

```
foreach ($keyvaluepair in $var1.GetEnumerator()) {
    $key1 = $_.Key1
    $val1 = $_.Val1
}
```

Eliminar un par de valores clave de una tabla hash existente

Un ejemplo, para eliminar una clave "Clave2" con un valor de "Valor2" de la tabla hash, utilizando el operador eliminar:

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}
$hashTable.Remove("Key2", "Value2")
$hashTable

#Output

Name                           Value
----                           -----
Key1                           Value1
```

Lea HashTables en línea: https://riptutorial.com/es/powershell/topic/8083/hashtables

Capítulo 31: Incrustar código gestionado (C # | VB)

Introducción

Este tema es para describir brevemente cómo el código administrado de C # o VB .NET puede ser ejecutado en un script de PowerShell. Este tema no está explorando todas las facetas del cmdlet Add-Type.

Para obtener más información sobre el cmdlet Add-Type, consulte la documentación de MSDN (para 5.1) aquí: https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/add- tipo

Parámetros

Parámetro	Detalles
-TypeDefinition <String_>	Acepta el código como una cadena.
-Lenguaje <String_>	Especifica el lenguaje de código administrado. Valores aceptados: CSharp, CSharpVersion3, CSharpVersion2, VisualBasic, JScript

Observaciones

Eliminar tipos agregados

En versiones posteriores de PowerShell, Remove-TypeData se agregó a las bibliotecas de cmdlet de PowerShell que pueden permitir la eliminación de un tipo dentro de una sesión. Para obtener más detalles sobre este cmdlet, vaya aquí: https://msdn.microsoft.com/en-us/powershell/reference/4.0/microsoft.powershell.utility/remove-typedata

Sintaxis CSharp y .NET

Para aquellas experiencias con .NET, no hace falta decir que las diferentes versiones de C # pueden ser radicalmente diferentes en su nivel de compatibilidad con cierta sintaxis.

Si utiliza Powershell 1.0 y / o -Language CSharp, el código administrado utilizará .NET 2.0, que carece de una serie de características que los desarrolladores de C # normalmente usan sin pensarlo dos veces, como Genéricos, Linq y Lambda. Además de esto, se encuentra el polimorfismo formal, que se maneja con parámetros predeterminados en versiones posteriores de C # / .NET.

Examples

Ejemplo de C

Este ejemplo muestra cómo incrustar algunos C # básicos en un script de PowerShell, agregarlo al espacio de ejecución / sesión y utilizar el código dentro de la sintaxis de PowerShell.

```
$code = "
using System;

namespace MyNameSpace
{
    public class Responder
    {
        public static void StaticRespond()
        {
            Console.WriteLine("Static Response");
        }

        public void Respond()
        {
            Console.WriteLine("Instance Respond");
        }
    }
}
"@

# Check the type has not been previously added within the session, otherwise an exception is
raised
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language CSharp;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

Ejemplo de VB.NET

Este ejemplo muestra cómo incrustar algunos C # básicos en un script de PowerShell, agregarlo al espacio de ejecución / sesión y utilizar el código dentro de la sintaxis de PowerShell.

```
$code = @"
Imports System

Namespace MyNameSpace
    Public Class Responder
        Public Shared Sub StaticRespond()
            Console.WriteLine("Static Response")
        End Sub

        Public Sub Respond()
            Console.WriteLine("Instance Respond")
        End Sub
```

```
    End Class
End Namespace
"@

# Check the type has not been previously added within the session, otherwise an exception is
raised
if (-not ([System.Management.Automation.PSTypeName]'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language VisualBasic;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

Lea Incrustar código gestionado (C # | VB) en línea:
https://riptutorial.com/es/powershell/topic/9823/incrustar-codigo-gestionado--c-sharp---vb-

Capítulo 32: Instrumentos de cuerda

Sintaxis

- "(Cita doble)"
- 'Cuerda literal'
- @ "
  Aquí cadena
  "@
- @ '
  Literal aquí-cadena
  '@

Observaciones

Las cadenas son objetos que representan texto.

Examples

Creando una cadena básica

Cuerda

Las cadenas se crean envolviendo el texto con comillas dobles. Las cadenas entre comillas dobles pueden evaluar variables y caracteres especiales.

```
$myString = "Some basic text"
$mySecondString = "String with a $variable"
```

Para usar una comilla doble dentro de una cadena, debe escaparse usando el carácter de escape, comilla hacia atrás (`). Las comillas simples se pueden utilizar dentro de una cadena entre comillas dobles.

```
$myString = "A `"double quoted`" string which also has 'single quotes'."
```

Cuerda literal

Las cadenas literales son cadenas que no evalúan variables y caracteres especiales. Se crea utilizando comillas simples.

```
$myLiteralString = 'Simple text including special characters (`n) and a $variable-reference'
```

Para usar comillas simples dentro de una cadena literal, use comillas simples dobles o una cadena literal aquí. Los qutoes dobles se pueden usar de manera segura dentro de una cadena literal

```
$myLiteralString = 'Simple string with ''single quotes'' and "double quotes".'
```

Cadena de formato

```
$hash = @{ city = 'Berlin' }

$result = 'You should really visit {0}' -f $hash.city
Write-Host $result #prints "You should really visit Berlin"
```

Las cadenas de formato se pueden usar con el operador `-f` o el método estático `[String]::Format(string format, args)` .NET.

Cuerda multilínea

Hay varias formas de crear una cadena multilínea en PowerShell:

- Puede usar los caracteres especiales para el retorno de carro y / o la nueva línea manualmente o usar la variable `NewLine` para insertar el valor de "nueva línea" de los sistemas)

  ```
  "Hello`r`nWorld"
  "Hello{0}World" -f [environment]::NewLine
  ```

- Cree un salto de línea mientras define una cadena (antes de cerrar la cita)

  ```
  "Hello
  World"
  ```

- Usando una cadena de aquí. *Esta es la técnica más común.*

  ```
  @"
  Hello
  World
  "@
  ```

Aquí cadena

Aquí, las cadenas son muy útiles al crear cadenas multilínea. Uno de los mayores beneficios en comparación con otras cadenas multilínea es que puede usar comillas sin tener que escapar de ellas usando un backtick.

Aquí cadena

Aquí, las cadenas comienzan con `@"` y un salto de línea y terminan con `"@` en su propia línea (`"@` **deben ser los primeros caracteres de la línea, ni siquiera los espacios en blanco / tabulador**).

```
@"
Simple
    Multiline string
with "quotes"
"@
```

Literal aquí-cadena

También puede crear una cadena literal aquí utilizando comillas simples, cuando no desee que las expresiones se expandan como una cadena literal normal.

```
@'
The following line won't be expanded
$(Get-Date)
because this is a literal here-string
'@
```

Cuerdas de concatenacion

Usando variables en una cadena

Puede concatenar cadenas utilizando variables dentro de una cadena entre comillas dobles. Esto no funciona con propiedades.

```
$string1 = "Power"
$string2 = "Shell"
"Greetings from $string1$string2"
```

Usando el operador `+`

También puedes unir cadenas utilizando el operador `+` .

```
$string1 = "Greetings from"
$string2 = "PowerShell"
$string1 + " " + $string2
```

Esto también funciona con las propiedades de los objetos.

```
"The title of this console is '" + $host.Name + "'"
```

Usando subexpresiones

La salida / resultado de una subexpresión `$()` se puede utilizar en una cadena. Esto es útil al acceder a propiedades de un objeto o al realizar una expresión compleja. Las subexpresiones pueden contener varias declaraciones separadas por punto `;` coma `;`

```
"Tomorrow is $((Get-Date).AddDays(1).DayOfWeek)"
```

Caracteres especiales

Cuando se utiliza dentro de una cadena de comillas dobles, el carácter de escape (backtick `` ` ``) representa un carácter especial.

```
`0    #Null
`a    #Alert/Beep
`b    #Backspace
`f    #Form feed (used for printer output)
`n    #New line
`r    #Carriage return
`t    #Horizontal tab
`v    #Vertical tab (used for printer output)
```

Ejemplo:

```
> "This`tuses`ttab`r`nThis is on a second line"
This    uses    tab
This is on a second line
```

También puedes escapar de caracteres especiales con significados especiales:

```
`#    #Comment-operator
`$    #Variable operator
``    #Escape character
`'    #Single quote
`"    #Double quote
```

Lea Instrumentos de cuerda en línea:
https://riptutorial.com/es/powershell/topic/5124/instrumentos-de-cuerda

Capítulo 33: Introducción a Pester

Observaciones

Pester es un marco de prueba para PowerShell que le permite ejecutar casos de prueba para su código de PowerShell. Se puede utilizar para ejecutar ex. Pruebas unitarias para ayudarlo a verificar que sus módulos, scripts, etc. funcionen según lo previsto.

¿Qué es Pester y por qué debería importarme?

Examples

Empezando con Pester

Para comenzar con las pruebas unitarias del código de PowerShell usando el módulo Pester, debe estar familiarizado con tres palabras clave / comandos:

- **Describa** : define un grupo de pruebas. Todos los archivos de prueba de Pester necesitan al menos un bloque Describe.
- **Es** : Define una prueba individual. Puedes tener múltiples It-blocks dentro de un Describe-block.
- **Debe** : El comando de verificación / prueba. Se utiliza para definir el resultado que debe considerarse una prueba exitosa.

Muestra:

```
Import-Module Pester

#Sample function to run tests against
function Add-Numbers{
    param($a, $b)
    return [int]$a + [int]$b
}

#Group of tests
Describe "Validate Add-Numbers" {

        #Individual test cases
        It "Should add 2 + 2 to equal 4" {
            Add-Numbers 2 2 | Should Be 4
        }

        It "Should handle strings" {
            Add-Numbers "2" "2" | Should Be 4
        }

        It "Should return an integer"{
            Add-Numbers 2.3 2 | Should BeOfType Int32
        }

}
```

Salida:

```
Describing Validate Add-Numbers
 [+] Should add 2 + 2 to equal 4 33ms
 [+] Should handle strings 19ms
 [+] Should return an integer 23ms
```

Lea Introducción a Pester en línea: https://riptutorial.com/es/powershell/topic/5753/introduccion-a-pester

Capítulo 34: Introducción a Psake

Sintaxis

- Tarea - función principal para ejecutar un paso de su script de compilación
- Depends - propiedad que especifica de qué depende el paso actual
- predeterminado: siempre debe haber una tarea predeterminada que se ejecutará si no se especifica una tarea inicial
- FormatTaskName: especifica cómo se muestra cada paso en la ventana de resultados.

Observaciones

psake es una herramienta de automatización de compilación escrita en PowerShell, y está inspirada en Rake (Ruby make) y Bake (Boo make). Se utiliza para crear compilaciones utilizando el patrón de dependencia. Documentación disponible aquí.

Examples

Esquema básico

```
Task Rebuild -Depends Clean, Build  {
   "Rebuild"
 }

Task Build {
   "Build"
 }

Task Clean {
   "Clean"
 }

Task default -Depends Build
```

Ejemplo de FormatTaskName

```
# Will display task as:
# -------- Rebuild --------
# -------- Build --------
FormatTaskName "-------- {0} --------"

# will display tasks in yellow colour:
# Running Rebuild
FormatTaskName {
    param($taskName)
    "Running $taskName" - foregroundcolor yellow
}

Task Rebuild -Depends Clean, Build  {
```

```
   "Rebuild"
 }

Task Build {
   "Build"
 }

Task Clean {
   "Clean"
 }

Task default -Depends Build
```

Ejecutar tarea condicionalmente

```
propreties {
    $isOk = $false
}

# By default the Build task won't run, unless there is a param $true
Task Build -precondition { return $isOk } {
   "Build"
 }

Task Clean {
   "Clean"
 }

Task default -Depends Build
```

ContinueOnError

```
Task Build -depends Clean {
   "Build"
 }

Task Clean -ContinueOnError {
   "Clean"
    throw "throw on purpose, but the task will continue to run"
 }

Task default -Depends Build
```

Lea Introducción a Psake en línea: https://riptutorial.com/es/powershell/topic/5019/introduccion-a-psake

Capítulo 35: Línea de comandos de PowerShell.exe

Parámetros

Parámetro	Descripción
-Ayuda | -? | /?	Muestra la ayuda
-File <FilePath> [<Args>]	Ruta al archivo de script que debe ejecutarse y argumentos (opcional)
-Comandante {- | <script-block> [-args <arg-array>] | <string> [<CommandParameters>]}	Comandos a ejecutar seguidos de argumentos.
-EncodedCommand <Base64EncodedCommand>	Comandos codificados en base64
-ExecutionPolicy <ExecutionPolicy>	Establece la política de ejecución solo para este proceso.
-InputFormat {Texto | XML}	Establece el formato de entrada para los datos enviados a procesar. Texto (cadenas) o XML (CLIXML serializado)
-MTA	PowerShell 3.0+: ejecuta PowerShell en un apartamento con múltiples subprocesos (STA es la opción predeterminada)
-Sta	PowerShell 2.0: ejecuta PowerShell en un apartamento de un solo hilo (MTA es el predeterminado)
-Sin salida	Deja la consola PowerShell en ejecución después de ejecutar el script / comando
-Sin logo	Oculta banner de copyright en el lanzamiento
-No interactivo	Oculta la consola del usuario.
-Sin perfil	Evitar la carga de perfiles PowerShell para máquina o usuario.
-OutputFormat {Texto | XML}	Establece el formato de salida para los datos devueltos desde PowerShell. Texto

Parámetro	Descripción
	(cadenas) o XML (CLIXML serializado)
-PSConsoleFile <FilePath>	Carga un archivo de consola creado previamente que configura el entorno (creado con `Export-Console`)
-Versión <versión de Windows PowerShell>	Especifique una versión de PowerShell para ejecutar. Utilizado principalmente con `2.0`
-WindowStyle <style>	Especifica si iniciar el proceso de PowerShell como una ventana `normal` , `hidden` , `minimized` o `maximized` .

Examples

Ejecutando un comando

El parámetro `-Command` se usa para especificar los comandos que se ejecutarán en el inicio. Es compatible con múltiples entradas de datos.

-Comando <cadena>

Puede especificar comandos para ejecutar en el lanzamiento como una cadena. Punto ; coma múltiple -Se pueden ejecutar declaraciones separadas.

```
>PowerShell.exe -Command "(Get-Date).ToShortDateString()"
10.09.2016

>PowerShell.exe -Command "(Get-Date).ToShortDateString(); 'PowerShell is fun!'"
10.09.2016
PowerShell is fun!
```

-Comando {scriptblock}

El parámetro `-Command` también admite una entrada de scriptblock (una o varias declaraciones envueltas entre llaves `{ #code }` . Esto solo funciona cuando se llama a `PowerShell.exe` desde otra sesión de Windows PowerShell.

```
PS > powershell.exe -Command {
"This can be useful, sometimes..."
(Get-Date).ToShortDateString()
}
This can be useful, sometimes...
10.09.2016
```

-Comando - (entrada estándar)

Puede pasar comandos desde la entrada estándar usando `-Command -` . La entrada estándar puede provenir de `echo` , leer un archivo, una aplicación de consola heredada, etc.

```
>echo "Hello World";"Greetings from PowerShell" | PowerShell.exe -NoProfile -Command -
Hello World
Greetings from PowerShell
```

Ejecutando un archivo de script

Puede especificar un archivo en un `ps1` para ejecutar su contenido al `ps1` utilizando el parámetro `-File` .

Guion basico

MyScript.ps1

```
(Get-Date).ToShortDateString()
"Hello World"
```

Salida:

```
>PowerShell.exe -File Desktop\MyScript.ps1
10.09.2016
Hello World
```

Uso de parámetros y argumentos.

Puede agregar parámetros y / o argumentos después de filepath para usarlos en el script. Los argumentos se utilizarán como valores para parámetros de script disponibles / indefinidos, el resto estará disponible en la matriz `$args`

MyScript.ps1

```
param($Name)

"Hello $Name! Today's date it $((Get-Date).ToShortDateString())"
"First arg: $($args[0])"
```

Salida:

```
>PowerShell.exe -File Desktop\MyScript.ps1 -Name StackOverflow foo
Hello StackOverflow! Today's date it 10.09.2016
First arg: foo
```

Lea Línea de comandos de PowerShell.exe en línea:
https://riptutorial.com/es/powershell/topic/5839/linea-de-comandos-de-powershell-exe

Capítulo 36: Lógica condicional

Sintaxis

- si (expresión) {}
- if (expresión) {} else {}
- if (expresión) {} elseif (expresión) {}
- if (expresión) {} elseif (expresión) {} else {}

Observaciones

Consulte también Operadores de comparación , que se pueden usar en expresiones condicionales.

Examples

si, si no y si

Powershell es compatible con operadores lógicos condicionales estándar, al igual que muchos lenguajes de programación. Estos permiten que ciertas funciones o comandos se ejecuten en circunstancias particulares.

Con un `if` los comandos dentro de los corchetes (`{}`) solo se ejecutan si se cumplen las condiciones dentro de if (`()`)

```
$test = "test"
if ($test -eq "test"){
    Write-Host "if condition met"
}
```

También puedes hacer otra `else` . Aquí se ejecutan los comandos `else if` **no se** cumplen las condiciones `if` :

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
else{
    Write-Host "if condition not met"
}
```

o un `elseif` . An else if ejecuta los comandos si no se cumplen las condiciones `if` y se cumplen las condiciones `elseif` :

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
```

```
}
elseif ($test -eq "test"){
    Write-Host "ifelse condition met"
}
```

Tenga en cuenta el uso anterior `-eq` (igualdad) CmdLet y no `=` o `==` como muchos otros idiomas hacen para equlaity.

Negación

Es posible que desee negar un valor booleano, es decir, ingrese una sentencia `if` cuando una condición sea falsa en lugar de verdadera. Esto se puede hacer mediante el uso de la `-Not` cmdlet

```
$test = "test"
if (-Not $test -eq "test2"){
    Write-Host "if condition not met"
}
```

También puedes usar `!` :

```
$test = "test"
if (!($test -eq "test2")){
    Write-Host "if condition not met"
}
```

También existe el `-ne` (no igual):

```
$test = "test"
if ($test -ne "test2"){
    Write-Host "variable test is not equal to 'test2'"
}
```

Si la taquigrafía condicional

Si desea utilizar la taquigrafía, puede utilizar la lógica condicional con la siguiente taquigrafía. Solo la cadena 'false' se evaluará como verdadera (2.0).

```
#Done in Powershell 2.0
$boolean = $false;
$string = "false";
$emptyString = "";

If($boolean){
    # this does not run because $boolean is false
    Write-Host "Shorthand If conditions can be nice, just make sure they are always boolean."
}

If($string){
    # This does run because the string is non-zero length
    Write-Host "If the variable is not strictly null or Boolean false, it will evaluate to
true as it is an object or string with length greater than 0."
}
```

```
If($emptyString){
    # This does not run because the string is zero-length
    Write-Host "Checking empty strings can be useful as well."
}

If($null){
    # This does not run because the condition is null
    Write-Host "Checking Nulls will not print this statement."
}
```

Lea Lógica condicional en línea: https://riptutorial.com/es/powershell/topic/7208/logica-condicional

Capítulo 37: Los operadores

Introducción

Un operador es un personaje que representa una acción. Le indica al compilador / intérprete que realice operaciones matemáticas, relacionales o lógicas específicas y produzca un resultado final. PowerShell interpreta de una manera específica y clasifica en consecuencia, como los operadores aritméticos realizan operaciones principalmente en números, pero también afectan a cadenas y otros tipos de datos. Junto con los operadores básicos, PowerShell tiene una serie de operadores que ahorran tiempo y esfuerzo de codificación (por ejemplo: -like, -match, -replace, etc.).

Examples

Operadores aritméticos

```
1 + 2      # Addition
1 - 2      # Subtraction
-1         # Set negative value
1 * 2      # Multiplication
1 / 2      # Division
1 % 2      # Modulus
100 -shl 2 # Bitwise Shift-left
100 -shr 1 # Bitwise Shift-right
```

Operadores logicos

```
-and # Logical and
-or  # Logical or
-xor # Logical exclusive or
-not # Logical not
!    # Logical not
```

Operadores de Asignación

Aritmética simple:

```
$var = 1      # Assignment. Sets the value of a variable to the specified value
$var += 2     # Addition. Increases the value of a variable by the specified value
$var -= 1     # Subtraction. Decreases the value of a variable by the specified value
$var *= 2     # Multiplication. Multiplies the value of a variable by the specified value
$var /= 2     # Division. Divides the value of a variable by the specified value
$var %= 2     # Modulus. Divides the value of a variable by the specified value and then
              # assigns the remainder (modulus) to the variable
```

Incremento y decremento:

```
$var++   # Increases the value of a variable, assignable property, or array element by 1
$var--   # Decreases the value of a variable, assignable property, or array element by 1
```

Operadores de comparación

Los operadores de comparación de PowerShell están compuestos por un guión inicial (`-`) seguido de un nombre (`eq` para `equal` , `gt` para `greater than` , etc ...).

Los nombres pueden ir precedidos por caracteres especiales para modificar el comportamiento del operador:

```
i # Case-Insensitive Explicit (-ieq)
c # Case-Sensitive Explicit (-ceq)
```

Case-insensitive es el valor predeterminado si no se especifica, ("a" -eq "A") igual que ("a" -ieq "A").

Operadores de comparación simples:

```
2 -eq 2    # Equal to (==)
2 -ne 4    # Not equal to (!=)
5 -gt 2    # Greater-than (>)
5 -ge 5    # Greater-than or equal to (>=)
5 -lt 10   # Less-than (<)
5 -le 5    # Less-than or equal to (<=)
```

Operadores de comparación de cadenas:

```
"MyString" -like "*String"           # Match using the wildcard character (*)
"MyString" -notlike "Other*"         # Does not match using the wildcard character (*)
"MyString" -match "$String^"         # Matches a string using regular expressions
"MyString" -notmatch "$Other^"       # Does not match a string using regular expressions
```

Operadores de comparación de colecciones:

```
"abc", "def" -contains "def"           # Returns true when the value (right) is present
                                       # in the array (left)
"abc", "def" -notcontains "123"        # Returns true when the value (right) is not present
                                       # in the array (left)
"def" -in "abc", "def"                 # Returns true when the value (left) is present
                                       # in the array (right)
"123" -notin "abc", "def"              # Returns true when the value (left) is not present
                                       # in the array (right)
```

Operadores de redireccionamiento

Flujo de salida de éxito:

```
cmdlet > file     # Send success output to file, overwriting existing content
cmdlet >> file    # Send success output to file, appending to existing content
cmdlet 1>&2       # Send success and error output to error stream
```

Corriente de salida de error:

```
cmdlet 2> file    # Send error output to file, overwriting existing content
cmdlet 2>> file   # Send error output to file, appending to existing content
cmdlet 2>&1       # Send success and error output to success output stream
```

Corriente de salida de advertencia: (PowerShell 3.0+)

```
cmdlet 3> file    # Send warning output to file, overwriting existing content
cmdlet 3>> file   # Send warning output to file, appending to existing content
cmdlet 3>&1       # Send success and warning output to success output stream
```

Secuencia de salida detallada: (PowerShell 3.0+)

```
cmdlet 4> file    # Send verbose output to file, overwriting existing content
cmdlet 4>> file   # Send verbose output to file, appending to existing content
cmdlet 4>&1       # Send success and verbose output to success output stream
```

Corriente de salida de depuración: (PowerShell 3.0+)

```
cmdlet 5> file    # Send debug output to file, overwriting existing content
cmdlet 5>> file   # Send debug output to file, appending to existing content
cmdlet 5>&1       # Send success and debug output to success output stream
```

Flujo de salida de información: (PowerShell 5.0+)

```
cmdlet 6> file    # Send information output to file, overwriting existing content
cmdlet 6>> file   # Send information output to file, appending to existing content
cmdlet 6>&1       # Send success and information output to success output stream
```

Todos los flujos de salida:

```
cmdlet *> file    # Send all output streams to file, overwriting existing content
cmdlet *>> file   # Send all output streams to file, appending to existing content
cmdlet *>&1       # Send all output streams to success output stream
```

Diferencias con el operador de la tubería (|)

Los operadores de redireccionamiento solo redirigen flujos a archivos o flujos a flujos. El operador de tuberías bombea un objeto por la tubería hasta un cmdlet o la salida. La forma en que funciona la canalización difiere en general de cómo funciona la redirección y se puede leer en Cómo trabajar con la tubería de PowerShell

Mezcla de tipos de operandos: el tipo del operando izquierdo dicta el comportamiento.

Para la adición

```
"4" + 2         # Gives "42"
4 + "2"         # Gives 6
```

```
1,2,3 + "Hello" # Gives 1,2,3,"Hello"
"Hello" + 1,2,3 # Gives "Hello1 2 3"
```

Para la multiplicación

```
"3" * 2   # Gives "33"
2 * "3"   # Gives 6
1,2,3 * 2 # Gives 1,2,3,1,2,3
2 * 1,2,3 # Gives an error op_Multiply is missing
```

El impacto puede tener consecuencias ocultas en los operadores de comparación:

```
$a = Read-Host "Enter a number"
Enter a number : 33
$a -gt 5
False
```

Operadores de manipulación de cuerdas

Reemplazar operador:

El operador `-replace` reemplaza un patrón en un valor de entrada usando una expresión regular. Este operador utiliza dos argumentos (separados por una coma): un patrón de expresión regular y su valor de reemplazo (que es opcional y una cadena vacía de forma predeterminada).

```
"The rain in Seattle" -replace 'rain','hail'        #Returns: The hail in Seattle
"kenmyer@contoso.com" -replace '^[\w]+@(.+)', '$1'  #Returns: contoso.com
```

Dividir y unir los operadores:

El operador `-split` divide una cadena en una matriz de `-split` .

```
"A B C" -split " "      #Returns an array string collection object containing A,B and C.
```

El operador `-join` une una matriz de cadenas en una sola cadena.

```
"E","F","G" -join ":"   #Returns a single string: E:F:G
```

Lea Los operadores en línea: https://riptutorial.com/es/powershell/topic/1071/los-operadores

Capítulo 38: Manejo de errores

Introducción

Este tema trata sobre los tipos de error y el manejo de errores en PowerShell.

Examples

Tipos de error

Un error es un error, uno podría preguntarse cómo podría haber tipos en él. Bueno, con powershell el error se divide en dos criterios,

- Error de terminación
- Error de no terminación

Como su nombre lo indica, los errores de terminación terminarán la ejecución y los errores de no terminación permiten que la ejecución continúe con la siguiente declaración.

> Esto es cierto suponiendo que el valor **$ ErrorActionPreference** es el valor predeterminado (Continuar). $ ErrorActionPreference es una Variable de Prefrence que le dice a powershell qué hacer en caso de un error de "No Terminación".

Error de terminación

Un error de terminación se puede manejar con una captura de prueba típica, como se muestra a continuación

```
Try
{
    Write-Host "Attempting Divide By Zero"
    1/0
}
Catch
{
    Write-Host "A Terminating Error: Divide by Zero Caught!"
}
```

El fragmento de código anterior se ejecutará y el error se detectará a través del bloque catch.

Error de no terminación

Un error de no terminación en la otra parte no se detectará en el bloque catch por defecto. La razón detrás de eso es un error de no terminación no se considera un error crítico.

```
Try
{
    Stop-Process -Id 123456
```

```
}
Catch
{
    Write-Host "Non-Terminating Error: Invalid Process ID"
}
```

Si ejecuta lo anterior, no obtendrá la salida del bloque catch, ya que el error no se considera crítico y la ejecución simplemente continuará a partir del siguiente comando. Sin embargo, el error se mostrará en la consola. Para manejar un error de no terminación, simplemente tiene que cambiar la preferencia de error.

```
Try
{
    Stop-Process -Id 123456 -ErrorAction Stop
}
Catch
{
    "Non-Terminating Error: Invalid Process ID"
}
```

Ahora, con la preferencia Error actualizada, este error se considerará un error de terminación y se detectará en el bloque catch.

Invocando errores de terminación y no de terminación:

El cmdlet **Write-Error** simplemente escribe el error en el programa host que invoca. No detiene la ejecución. Donde como **lanzar** te dará un error de terminación y detendrá la ejecución.

```
Write-host "Going to try a non terminating Error "
Write-Error "Non terminating"
Write-host "Going to try a terminating Error "
throw "Terminating Error "
Write-host "This Line wont be displayed"
```

Lea Manejo de errores en línea: https://riptutorial.com/es/powershell/topic/8075/manejo-de-errores

Capítulo 39: Manejo de secretos y credenciales

Introducción

En Powershell, para evitar almacenar la contraseña en *texto claro* , utilizamos diferentes métodos de cifrado y la almacenamos como una cadena segura. Cuando no esté especificando una clave o una clave segura, esto solo funcionará para el mismo usuario que en la misma computadora podrá descifrar la cadena cifrada si no está usando Claves / Claves de seguridad. Cualquier proceso que se ejecute bajo esa misma cuenta de usuario podrá descifrar esa cadena cifrada en esa misma máquina.

Examples

Solicitando Credenciales

Para solicitar credenciales, casi siempre debe usar el cmdlet `Get-Credential` :

```
$credential = Get-Credential
```

Nombre de usuario rellenado previamente:

```
$credential = Get-Credential -UserName 'myUser'
```

Agregue un mensaje de solicitud personalizado:

```
$credential = Get-Credential -Message 'Please enter your company email address and password.'
```

Acceso a la contraseña de texto sin formato

La contraseña en un objeto de credencial es un `[SecureString]` cifrado. La forma más sencilla es obtener un `[NetworkCredential]` que no almacena la contraseña cifrada:

```
$credential = Get-Credential
$plainPass = $credential.GetNetworkCredential().Password
```

El método auxiliar (`.GetNetworkCredential()`) solo existe en los objetos `[PSCredential]` .
Para tratar directamente con un `[SecureString]` , use los métodos .NET:

```
$bstr = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($secStr)
$plainPass = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
```

Trabajar con credenciales almacenadas

Para almacenar y recuperar credenciales cifradas fácilmente, use la serialización XML incorporada de PowerShell (Clixml):

```
$credential = Get-Credential

$credential | Export-CliXml -Path 'C:\My\Path\cred.xml'
```

Para volver a importar:

```
$credential = Import-CliXml -Path 'C:\My\Path\cred.xml'
```

Lo importante a recordar es que, de forma predeterminada, utiliza la API de protección de datos de Windows, y la clave utilizada para cifrar la contraseña es específica tanto para el *usuario como para la máquina en la* que se ejecuta el código.

Como resultado, la credencial cifrada no puede ser importada por un usuario diferente ni por el mismo usuario en una computadora diferente.

Al encriptar varias versiones de la misma credencial con diferentes usuarios en ejecución y en diferentes computadoras, puede tener el mismo secreto disponible para múltiples usuarios.

Al poner el nombre del usuario y la computadora en el nombre del archivo, puede almacenar todos los secretos encriptados de una manera que permita que el mismo código los use sin codificar nada:

Encriptador

```
# run as each user, and on each computer

$credential = Get-Credential

$credential | Export-CliXml -Path
"C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

El código que utiliza las credenciales almacenadas:

```
$credential = Import-CliXml -Path
"C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

La versión correcta del archivo para el usuario en ejecución se cargará automáticamente (o fallará porque el archivo no existe).

Almacenar las credenciales en forma cifrada y pasarlas como parámetro cuando sea necesario

```
$username = "user1@domain.com"
$pwdTxt = Get-Content "C:\temp\Stored_Password.txt"
$securePwd = $pwdTxt | ConvertTo-SecureString
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd
# Now, $credObject is having the credentials stored and you can pass it wherever you want.


## Import Password with AES

$username = "user1@domain.com"
$AESKey = Get-Content $AESKeyFilePath
$pwdTxt = Get-Content $SecurePwdFilePath
$securePwd = $pwdTxt | ConvertTo-SecureString -Key $AESKey
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd

# Now, $credObject is having the credentials stored with AES Key and you can pass it wherever
you want.
```

Lea Manejo de secretos y credenciales en línea:
https://riptutorial.com/es/powershell/topic/2917/manejo-de-secretos-y-credenciales

Capítulo 40: Módulo ActiveDirectory

Introducción

Este tema le presentará algunos de los cmdlets básicos utilizados en el Módulo de Active Directory para PowerShell, para manipular usuarios, grupos, computadoras y objetos.

Observaciones

Recuerde que el sistema de ayuda de PowerShell es uno de los mejores recursos que puede utilizar.

```
Get-Help Get-ADUser -Full
Get-Help Get-ADGroup -Full
Get-Help Get-ADComputer -Full
Get-Help Get-ADObject -Full
```

Toda la documentación de ayuda proporcionará ejemplos, la sintaxis y la ayuda de parámetros.

Examples

Módulo

```
#Add the ActiveDirectory Module to current PowerShell Session
Import-Module ActiveDirectory
```

Usuarios

Recuperar usuario de Active Directory

```
Get-ADUser -Identity JohnSmith
```

Recuperar todas las propiedades asociadas con el usuario

```
Get-ADUser -Identity JohnSmith -Properties *
```

Recuperar las propiedades seleccionadas para el usuario

```
Get-ADUser -Identity JohnSmith -Properties * | Select-Object -Property sAMAccountName, Name,
Mail
```

Nuevo usuario de AD

```
New-ADUser -Name "MarySmith" -GivenName "Mary" -Surname "Smith" -DisplayName "MarySmith" -Path
"CN=Users,DC=Domain,DC=Local"
```

Los grupos

Recuperar grupo de Active Directory

```
Get-ADGroup -Identity "My-First-Group" #Ensure if group name has space quotes are used
```

Recuperar todas las propiedades asociadas con el grupo

```
Get-ADGroup -Identity "My-First-Group" -Properties *
```

Recuperar todos los miembros de un grupo

```
Get-ADGroupMember -Identity "My-First-Group" | Select-Object -Property sAMAccountName
Get-ADgroup "MY-First-Group" -Properties Members | Select -ExpandProperty Members
```

Agregar usuario de AD a un grupo de AD

```
Add-ADGroupMember -Identity "My-First-Group" -Members "JohnSmith"
```

Nuevo grupo de anuncios

```
New-ADGroup -GroupScope Universal -Name "My-Second-Group"
```

Ordenadores

Recuperar AD Computer

```
Get-ADComputer -Identity "JohnLaptop"
```

Recuperar todas las propiedades asociadas con la computadora

```
Get-ADComputer -Identity "JohnLaptop" -Properties *
```

Recuperar propiedades seleccionadas de la computadora

```
Get-ADComputer -Identity "JohnLaptop" -Properties * | Select-Object -Property Name, Enabled
```

Objetos

Recuperar un objeto de Active Directory

```
#Identity can be ObjectGUID, Distinguished Name or many more
Get-ADObject -Identity "ObjectGUID07898"
```

Mueve un objeto de Active Directory

```
Move-ADObject -Identity "CN=JohnSmith,OU=Users,DC=Domain,DC=Local" -TargetPath
"OU=SuperUser,DC=Domain,DC=Local"
```

Modificar un objeto de Active Directory

```
Set-ADObject -Identity "CN=My-First-Group,OU=Groups,DC=Domain,DC=local" -Description "This is
My First Object Modification"
```

Lea Módulo ActiveDirectory en línea: https://riptutorial.com/es/powershell/topic/8213/modulo-activedirectory

Capítulo 41: Módulo de archivo

Introducción

El módulo de archivo `Microsoft.PowerShell.Archive` proporciona funciones para almacenar archivos en archivos ZIP (`Compress-Archive`) y extraerlos (`Expand-Archive`). Este módulo está disponible en PowerShell 5.0 y superior.

En versiones anteriores de PowerShell, se podrían usar las extensiones comunitarias o .NET System.IO.Compression.FileSystem .

Sintaxis

- **Expandir-Archivo** / **Comprimir-Archivo**
- **-Camino**
  - la ruta del archivo (s) a comprimir (Compress-Archive) o la ruta del archivo para extraer el archivo (s) form (Expand-Archive)
  - hay varias otras opciones relacionadas con la ruta, por favor ver más abajo.
- **-DestinationPath** (opcional)
  - Si no proporciona esta ruta, el archivo se creará en el directorio de trabajo actual (Compress-Archive) o el contenido del archivo se extraerá en el directorio de trabajo actual (Expand-Archive)

Parámetros

Parámetro	Detalles
Nivel de compresión	*(Compress-Archive only)* Configura el nivel de compresión en `Fastest` , `Optimal` o Sin `NoCompression`
Confirmar	Solicita confirmación antes de ejecutar
Fuerza	Obliga a ejecutar el comando sin confirmación.
LiteralPath	Ruta que se usa literalmente, *no se admiten comodines* , use ， para especificar múltiples rutas
Camino	Ruta que puede contener comodines, use ， para especificar múltiples rutas
Actualizar	*(Compress-Archive only)* Actualizar el archivo existente
Y si	Simular el comando

Observaciones

Ver MSDN Microsoft.PowerShell.Archive (5.1) para referencia adicional

Examples

Compress-Archive con comodines

```
Compress-Archive -Path C:\Documents\* -CompressionLevel Optimal -DestinationPath
C:\Archives\Documents.zip
```

Este comando:

- Comprime todos los archivos en `C:\Documents`
- Utiliza la compresión `Optimal`
- Guarde el archivo resultante en `C:\Archives\Documents.zip`
    - `-DestinationPath` agregará `.zip` si no está presente.
    - `-LiteralPath` se puede usar si necesita asignarle un nombre sin `.zip` .

Actualizar el ZIP existente con Compress-Archive

```
Compress-Archive -Path C:\Documents\* -Update -DestinationPath C:\Archives\Documents.zip
```

- esto agregará o reemplazará todos los archivos `Documents.zip` con los nuevos de `C:\Documents`

Extraer un Zip con Expandir-Archivo

```
Expand-Archive -Path C:\Archives\Documents.zip -DestinationPath C:\Documents
```

- esto extraerá todos los archivos de `Documents.zip` en la carpeta `C:\Documents`

Lea Módulo de archivo en línea: https://riptutorial.com/es/powershell/topic/9896/modulo-de-archivo

Capítulo 42: Módulo de SharePoint

Examples

Cargando complemento de SharePoint

La carga del complemento de SharePoint se puede hacer usando lo siguiente:

```
Add-PSSnapin "Microsoft.SharePoint.PowerShell"
```

Esto solo funciona en la versión de 64 bits de PowerShell. Si la ventana dice "Windows PowerShell (x86)" en el título, está utilizando la versión incorrecta.

Si el complemento ya está cargado, el código anterior causará un error. El uso de lo siguiente se cargará solo si es necesario, que se puede usar en los Cmdlets / funciones:

```
if ((Get-PSSnapin "Microsoft.SharePoint.PowerShell" -ErrorAction SilentlyContinue) -eq $null)
{
    Add-PSSnapin "Microsoft.SharePoint.PowerShell"
}
```

Alternativamente, si inicia el Shell de administración de SharePoint, incluirá automáticamente el complemento.

Para obtener una lista de todos los Cmdlets de SharePoint disponibles, ejecute lo siguiente:

```
Get-Command -Module Microsoft.SharePoint.PowerShell
```

Iterando sobre todas las listas de una colección de sitios

Imprima todos los nombres de lista y el recuento de elementos.

```
$site = Get-SPSite -Identity https://mysharepointsite/sites/test
foreach ($web in $site.AllWebs)
{
    foreach ($list in $web.Lists)
    {
        # Prints list title and item count
        Write-Output "$($list.Title), Items: $($list.ItemCount)"
    }
}
$site.Dispose()
```

Obtenga todas las características instaladas en una colección de sitios

```
Get-SPFeature -Site https://mysharepointsite/sites/test
```

Get-SPFeature también se puede ejecutar en el ámbito web (`-Web <WebUrl>`), ámbito agrícola (`-Farm`) y ámbito de aplicación web (`-WebApplication <WebAppUrl>`).

Obtenga todas las características huérfanas en una colección de sitios

Otro uso de Get-SPFeature puede ser encontrar todas las características que no tienen alcance:

```
Get-SPFeature -Site https://mysharepointsite/sites/test |? { $_.Scope -eq $null )
```

Lea Módulo de SharePoint en línea: https://riptutorial.com/es/powershell/topic/5147/modulo-de-sharepoint

Capítulo 43: Módulo de tareas programadas

Introducción

Ejemplos de cómo usar el módulo Tareas programadas disponible en Windows 8 / Server 2012 y en.

Examples

Ejecutar PowerShell Script en tareas programadas

Crea una tarea programada que se ejecuta de inmediato, luego en el inicio para ejecutar `C:\myscript.ps1` como `SYSTEM`

```
$ScheduledTaskPrincipal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType
ServiceAccount
$ScheduledTaskTrigger1 = New-ScheduledTaskTrigger -AtStartup
$ScheduledTaskTrigger2 = New-ScheduledTaskTrigger -Once -At $(Get-Date) -RepetitionInterval
"00:01:00" -RepetitionDuration $([timeSpan] "24855.03:14:07")
$ScheduledTaskActionParams = @{
    Execute = "PowerShell.exe"
    Argument = '-executionpolicy Bypass -NonInteractive -c C:\myscript.ps1 -verbose >>
C:\output.log 2>&1"'
}
$ScheduledTaskAction = New-ScheduledTaskAction @ScheduledTaskActionParams
Register-ScheduledTask -Principal $ScheduledTaskPrincipal -Trigger
@($ScheduledTaskTrigger1,$ScheduledTaskTrigger2) -TaskName "Example Task" -Action
$ScheduledTaskAction
```

Lea Módulo de tareas programadas en línea:
https://riptutorial.com/es/powershell/topic/10940/modulo-de-tareas-programadas

Capítulo 44: Módulo ISE

Introducción

Windows PowerShell Integrated Scripting Environment (ISE) es una aplicación host que le permite escribir, ejecutar y probar scripts y módulos en un entorno gráfico e intuitivo. Las características clave de Windows PowerShell ISE incluyen colores de sintaxis, finalización de pestañas, Intellisense, depuración visual, conformidad con Unicode y ayuda sensible al contexto, y proporcionan una rica experiencia de scripting.

Examples

Scripts de prueba

El uso simple y poderoso del ISE es, por ejemplo, escribir código en la sección superior (con color de sintaxis intuitiva) y ejecutar el código simplemente marcándolo y presionando la tecla F8.

```
function Get-Sum
{
    foreach ($i in $Input)
    {$Sum += $i}
    $Sum

1..10 | Get-Sum

#output
55
```

Lea Módulo ISE en línea: https://riptutorial.com/es/powershell/topic/10954/modulo-ise

Capítulo 45: Módulos Powershell

Introducción

A partir de la versión 2.0 de PowerShell, los desarrolladores pueden crear módulos de PowerShell. Los módulos de PowerShell encapsulan un conjunto de funcionalidades comunes. Por ejemplo, hay módulos PowerShell específicos del proveedor que administran varios servicios en la nube. También hay módulos genéricos de PowerShell que interactúan con los servicios de redes sociales y realizan tareas de programación comunes, como la codificación Base64, el trabajo con Named Pipes, y más.

Los módulos pueden exponer alias de comandos, funciones, variables, clases y más.

Examples

Crear un módulo de manifiesto

```
@{
  RootModule = 'MyCoolModule.psm1'
  ModuleVersion = '1.0'
  CompatiblePSEditions = @('Core')
  GUID = '6b42c995-67da-4139-be79-597a328056cc'
  Author = 'Bob Schmob'
  CompanyName = 'My Company'
  Copyright = '(c) 2017 Administrator. All rights reserved.'
  Description = 'It does cool stuff.'
  FunctionsToExport = @()
  CmdletsToExport = @()
  VariablesToExport = @()
  AliasesToExport = @()
  DscResourcesToExport = @()
}
```

Todo buen módulo de PowerShell tiene un módulo manifiesto. El manifiesto del módulo simplemente contiene metadatos sobre un módulo de PowerShell, y no define el contenido real del módulo.

El archivo de manifiesto es un archivo de script de PowerShell, con una extensión de archivo `.psd1` , que contiene una HashTable. La tabla hash en el manifiesto debe contener claves específicas para que PowerShell pueda interpretarlo correctamente como un archivo de módulo de PowerShell.

El ejemplo anterior proporciona una lista de las claves básicas de HashTable que conforman un módulo manifiesto, pero hay muchos otros. El comando `New-ModuleManifest` ayuda a crear un nuevo esqueleto de manifiesto de módulo.

Ejemplo de módulo simple

```
function Add {
  [CmdletBinding()]
  param (
    [int] $x
  , [int] $y
  )

  return $x + $y
}

Export-ModuleMember -Function Add
```

Este es un ejemplo simple de cómo podría verse un archivo de módulo de script de PowerShell. Este archivo se llamaría `MyCoolModule.psm1` y se hace referencia desde el archivo de manifiesto del módulo (.psd1). Notará que el comando `Export-ModuleMember` nos permite especificar qué funciones en el módulo queremos "exportar" o exponer al usuario del módulo. Algunas funciones serán solo internas, y no deberían estar expuestas, por lo que se omitirán de la llamada a `Export-ModuleMember` .

Exportando una variable desde un módulo

```
$FirstName = 'Bob'
Export-ModuleMember -Variable FirstName
```

Para exportar una variable desde un módulo, use el comando `Export-ModuleMember` , con el parámetro `-Variable` . Sin embargo, recuerde que si la variable tampoco se exporta explícitamente en el archivo de manifiesto del módulo (.psd1), entonces la variable no será visible para el consumidor del módulo. Piense en el módulo manifest como un "portero". Si una función o variable no está permitida en el manifiesto del módulo, no será visible para el consumidor del módulo.

Nota: Exportar una variable es similar a hacer público un campo en una clase. No es aconsejable. Sería mejor exponer una función para obtener el campo y una función para establecer el campo.

Estructuración de módulos PowerShell

En lugar de definir todas sus funciones en un solo archivo de módulo de script PowerShell `.psm1` , es posible que desee dividir su función en archivos individuales. A continuación, puede crear puntos de estos archivos desde su archivo de módulo de script, que en esencia los trata como si fueran parte del archivo `.psm1` .

Considere la estructura de directorio de este módulo:

```
\MyCoolModule
  \Functions
    Function1.ps1
    Function2.ps1
    Function3.ps1
MyCoolModule.psd1
MyCoolModule.psm1
```

Dentro de su archivo `MyCoolModule.psm1` , puede insertar el siguiente código:

```
Get-ChildItem -Path $PSScriptRoot\Functions |
  ForEach-Object -Process { . $PSItem.FullName }
```

Esto generaría los archivos de funciones individuales en el archivo de módulo `.psm1` .

Ubicación de los módulos

PowerShell busca módulos en los directorios listados en $ Env: PSModulepath.

Un módulo llamado *foo* , en una carpeta llamada *foo* se encontrará con `Import-Module foo`

En esa carpeta, PowerShell buscará un manifiesto de módulo (foo.psd1), un archivo de módulo (foo.psm1), una DLL (foo.dll).

Visibilidad del miembro del módulo

Por defecto, solo las funciones definidas en un módulo son visibles fuera del módulo. En otras palabras, si define variables y alias en un módulo, no estarán disponibles excepto en el código del módulo.

Para anular este comportamiento, puede usar el cmdlet `Export-ModuleMember` . Tiene parámetros denominados `-Function` , `-Variable` y `-Alias` que le permiten especificar exactamente qué miembros se exportan.

Es importante tener en cuenta que si utiliza `Export-ModuleMember` , **solo** estarán visibles los elementos que especifique.

Lea Módulos Powershell en línea: https://riptutorial.com/es/powershell/topic/8734/modulos-powershell

Capítulo 46: Módulos, Scripts y Funciones.

Introducción

Los módulos de PowerShell brindan extensibilidad al administrador de sistemas, al DBA y al desarrollador. Ya sea simplemente como un método para compartir funciones y scripts.

Las funciones de Powershell son para evitar códigos repetitivos. Consulte [Funciones PS] [1] [1]: Funciones de PowerShell

Los scripts de PowerShell se utilizan para automatizar tareas administrativas que consisten en un shell de línea de comandos y cmdlets asociados construidos sobre .NET Framework.

Examples

Función

Una función es un bloque de código con nombre que se utiliza para definir un código reutilizable que debería ser fácil de usar. Normalmente se incluye dentro de un script para ayudar a reutilizar el código (para evitar el código duplicado) o se distribuye como parte de un módulo para que sea útil para otros en múltiples scripts.

Escenarios donde una función podría ser útil:

- Calcula el promedio de un grupo de números.
- Generar un informe para ejecutar procesos.
- Escribir una función que pruebe que una computadora está "en buen estado" haciendo ping a la computadora y accediendo a `c$` -share

Las funciones se crean utilizando la palabra clave de `function` , seguida de un nombre de una sola palabra y un bloque de script que contiene el código que se ejecutará cuando se llame el nombre de la función.

```
function NameOfFunction {
    Your code
}
```

Manifestación

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}
```

Uso:

```
> HelloWorld
Greetings from PowerShell!
```

Guión

Una secuencia de comandos es un archivo de texto con la extensión de archivo `.ps1` que contiene los comandos de PowerShell que se ejecutarán cuando se llame la secuencia de comandos. Debido a que los scripts son archivos guardados, son fáciles de transferir entre computadoras.

Los guiones se escriben a menudo para resolver un problema específico, ej .:

- Ejecutar una tarea de mantenimiento semanal
- Para instalar y configurar una solución / aplicación en una computadora

Manifestación

MyFirstScript.ps1:

```
Write-Host "Hello World!"
2+2
```

Puede ejecutar un script ingresando la ruta al archivo usando un:

- Camino absoluto, ej. `c:\MyFirstScript.ps1`
- `.\MyFirstScript.ps1` relativa, por ejemplo, `.\MyFirstScript.ps1` si el directorio actual de su consola PowerShell era `C:\`

Uso:

```
> .\MyFirstScript.ps1
Hello World!
4
```

Un script también puede importar módulos, definir sus propias funciones, etc.

MySecondScript.ps1:

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}

HelloWorld
Write-Host "Let's get started!"
2+2
HelloWorld
```

Uso:

```
> .\MySecondScript.ps1
Greetings from PowerShell!
```

```
Let's get started!
4
Greetings from PowerShell!
```

Módulo

Un módulo es una colección de funciones reutilizables relacionadas (o cmdlets) que pueden distribuirse fácilmente a otros usuarios de PowerShell y usarse en múltiples scripts o directamente en la consola. Un módulo generalmente se guarda en su propio directorio y consta de:

- Uno o más archivos de código con la extensión de archivo `.psm1` que contiene funciones o ensamblajes binarios (`.dll`) que contienen cmdlets
- Un módulo manifest `.psd1` describe el nombre, la versión, el autor, la descripción de los módulos, las funciones / cmdlets que proporciona, etc.
- Otros requisitos para que funcione incl. dependencias, scripts etc.

Ejemplos de módulos:

- Un módulo que contiene funciones / cmdlets que realizan estadísticas en un conjunto de datos
- Un módulo para consultar y configurar bases de datos.

Para facilitar a PowerShell encontrar e importar un módulo, a menudo se coloca en una de las ubicaciones de módulos de PowerShell conocidas definidas en `$env:PSModulePath` .

Manifestación

Enumere los módulos que están instalados en una de las ubicaciones de módulos conocidas:

```
Get-Module -ListAvailable
```

Importar un módulo, ej. Módulo `Hyper-V` :

```
Import-Module Hyper-V
```

Listar los comandos disponibles en un módulo, ej. el módulo de `Microsoft.PowerShell.Archive`

```
> Import-Module Microsoft.PowerShell.Archive
> Get-Command -Module Microsoft.PowerShell.Archive

CommandType Name             Version Source
----------- ----             ------- ------
Function    Compress-Archive 1.0.1.0 Microsoft.PowerShell.Archive
Function    Expand-Archive   1.0.1.0 Microsoft.PowerShell.Archive
```

Funciones avanzadas

Las funciones avanzadas se comportan de la misma manera que los cmdlets. El ISE de

PowerShell incluye dos esqueletos de funciones avanzadas. Acceda a ellos a través del menú, editar, fragmentos de código o con Ctrl + J. (A partir de PS 3.0, las versiones posteriores pueden diferir)

Las cosas clave que incluyen funciones avanzadas son,

- `Get-Help` incorporada y personalizada para la función, accesible a través de `Get-Help`
- puede usar [CmdletBinding ()] que hace que la función actúe como un cmdlet
- amplias opciones de parámetros

Versión simple:

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                   ValueFromPipelineByPropertyName=$true,
                   Position=0)]
        $Param1,

        # Param2 help description
        [int]
        $Param2
    )

    Begin
    {
    }
    Process
    {
    }
    End
    {
    }
}
```

Versión completa:

```
<#
.Synopsis
   Short description
.DESCRIPTION
   Long description
```

```
.EXAMPLE
   Example of how to use this cmdlet
.EXAMPLE
   Another example of how to use this cmdlet
.INPUTS
   Inputs to this cmdlet (if any)
.OUTPUTS
   Output from this cmdlet (if any)
.NOTES
   General notes
.COMPONENT
   The component this cmdlet belongs to
.ROLE
   The role this cmdlet belongs to
.FUNCTIONALITY
   The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
                   SupportsShouldProcess=$true,
                   PositionalBinding=$false,
                   HelpUri = 'http://www.microsoft.com/',
                   ConfirmImpact='Medium')]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                    ValueFromPipeline=$true,
                    ValueFromPipelineByPropertyName=$true,
                    ValueFromRemainingArguments=$false,
                    Position=0,
                    ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,

        # Param2 help description
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,

        # Param3 help description
        [Parameter(ParameterSetName='Another Parameter Set')]
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    Begin
    {
```

```
    }
    Process
    {
        if ($pscmdlet.ShouldProcess("Target", "Operation"))
        {
        }
    }
    End
    {
    }
}
```

Lea Módulos, Scripts y Funciones. en línea:
https://riptutorial.com/es/powershell/topic/5755/modulos--scripts-y-funciones-

Capítulo 47: MongoDB

Observaciones

La parte más difícil es adjuntar un **subdocumento** al documento que aún no se ha creado. Si necesitamos que el subdocumento tenga el aspecto esperado, deberemos iterar con un bucle for de la matriz en una variable y usar `$doc2.add("Key", "Value")` lugar de usar la matriz actual de `foreach` con índice. Esto hará que el subdocumento en dos líneas, como se puede ver en las `"Tags" = [MongoDB.Bson.BsonDocument] $doc2 .`

Examples

MongoDB con controlador C # 1.7 utilizando PowerShell

Necesito consultar todos los detalles de la máquina virtual y actualizar en MongoDB.

```
Which require the output look like this.
{
    "_id" : ObjectId("5800509f23888a12bccf2347"),
    "ResourceGrp" : "XYZZ-MachineGrp",
    "ProcessTime" : ISODate("2016-10-14T03:27:16.586Z"),
    "SubscriptionName" : "GSS",
    "OS" : "Windows",
    "HostName" : "VM1",
    "IPAddress" : "192.168.22.11",
    "Tags" : {
        "costCenter" : "803344",
        "BusinessUNIT" : "WinEng",
        "MachineRole" : "App",
        "OwnerEmail" : "zteffer@somewhere.com",
        "appSupporter" : "Steve",
        "environment" : "Prod",
        "implementationOwner" : "xyzr@somewhere.com",
        "appSoftware" : "WebServer",
        "Code" : "Gx",
        "WholeOwner" : "zzzgg@somewhere.com"
    },
    "SubscriptionID" : "",
    "Status" : "running fine",
    "ResourceGroupName" : "XYZZ-MachineGrp",
    "LocalTime" : "14-10-2016-11:27"
}
```

Tengo 3 conjuntos de matriz en Powershell

```
        $MachinesList  # Array
        $ResourceList # Array
        $MachineTags  # Array

    pseudo code
```

```
        $mongoDriverPath = 'C:\Program Files (x86)\MongoDB\CSharpDriver 1.7';
        Add-Type -Path "$($mongoDriverPath)\MongoDB.Bson.dll";
        Add-Type -Path "$($mongoDriverPath)\MongoDB.Driver.dll";

$db = [MongoDB.Driver.MongoDatabase]::Create('mongodb://127.0.0.1:2701/RGrpMachines');
   [System.Collections.ArrayList]$TagList = $vm.tags
     $A1 = $Taglist.key
     $A2 = $Taglist.value
   foreach ($Machine in $MachinesList)
   {
       foreach($Resource in $ResourceList)
       {
                   $doc2 = $null
                  [MongoDB.Bson.BsonDocument] $doc2 = @{}; #Create a Document here
                   for($i = 0; $i -lt $TagList.count; $i++)
                          {
                                $A1Key = $A1[$i].ToString()
                                $A2Value = $A2[$i].toString()
                                $doc2.add("$A1Key", "$A2Value")
                          }

                    [MongoDB.Bson.BsonDocument] $doc = @{
                       "_id"= [MongoDB.Bson.ObjectId]::GenerateNewId();
                       "ProcessTime"= [MongoDB.Bson.BsonDateTime] $ProcessTime;
                       "LocalTime" = "$LocalTime";
                       "Tags" = [MongoDB.Bson.BsonDocument] $doc2;
                       "ResourceGrp" = "$RGName";
                       "HostName"= "$VMName";
                       "Status"= "$VMStatus";
                       "IPAddress"= "$IPAddress";
                       "ResourceGroupName"= "$RGName";
                       "SubscriptionName"= "$CurSubName";
                       "SubscriptionID"= "$subid";
                       "OS"= "$OSType";
                   }; #doc loop close

                   $collection.Insert($doc);
           }
   }
```

Lea MongoDB en línea: https://riptutorial.com/es/powershell/topic/7438/mongodb

Capítulo 48: Operadores Especiales

Examples

Operador de Expresión de Array

Devuelve la expresión como una matriz.

```
@(Get-ChildItem $env:windir\System32\ntdll.dll)
```

Devolverá una matriz con un elemento

```
@(Get-ChildItem $env:windir\System32)
```

Devolverá una matriz con todos los elementos en la carpeta (que no es un cambio de comportamiento de la expresión interna.

Operación de llamada

```
$command = 'Get-ChildItem'
& $Command
```

Ejecutará `Get-ChildItem`

Operador de abastecimiento de puntos

. . \ myScript.ps1

ejecuta `.\myScript.ps1` en el ámbito actual haciendo que todas las funciones y variables estén disponibles en el ámbito actual.

Lea Operadores Especiales en línea: https://riptutorial.com/es/powershell/topic/8981/operadores-especiales

Capítulo 49: Parámetros comunes

Observaciones

Los parámetros comunes se pueden usar con cualquier cmdlet (eso significa que tan pronto como marque su función como cmdlet [vea `CmdletBinding()`], obtendrá todos estos parámetros de forma gratuita).

Aquí está la lista de todos los parámetros comunes (el alias está entre paréntesis después del parámetro correspondiente):

```
-Debug (db)
-ErrorAction (ea)
-ErrorVariable (ev)
-InformationAction (ia) # introduced in v5
-InformationVariable (iv) # introduced in v5
-OutVariable (ov)
-OutBuffer (ob)
-PipelineVariable (pv)
-Verbose (vb)
-WarningAction (wa)
-WarningVariable (wv)
-WhatIf (wi)
-Confirm (cf)
```

Examples

Parámetro ErrorAction

Los valores posibles son `Continue | Ignore | Inquire | SilentlyContinue | Stop | Suspend`

El valor de este parámetro determinará cómo el cmdlet manejará los errores de no terminación (los generados por Write-Error, por ejemplo; para obtener más información sobre el manejo de errores, vea [*tema aún no creado*]).

El valor predeterminado (si se omite este parámetro) es `Continue` .

-ErrorAction Continuar

Esta opción producirá un mensaje de error y continuará con la ejecución.

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second comma
Write-Error "test" -ErrorAction Continue ; Write-Host "Second command" : te
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorEx
    + FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorExcep

Second command
```

-ErrorAction Ignore

Esta opción no producirá ningún mensaje de error y continuará con la ejecución. Tampoco se añadirán `$Error` la variable automática `$Error` .
Esta opción fue introducida en v3.

```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second comman
Second command
```

-ErrorAction Consultar

Esta opción producirá un mensaje de error y le pedirá al usuario que elija una acción para realizar.

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second comma

Confirm
test
[Y] Yes  [A] Yes to All  [H] Halt Command  [S] Suspend  [?] Help (default
```

-ErrorAction SilentlyContinue

Esta opción no producirá un mensaje de error y continuará con la ejecución. Todos los errores serán agregados a la variable automática `$Error` .

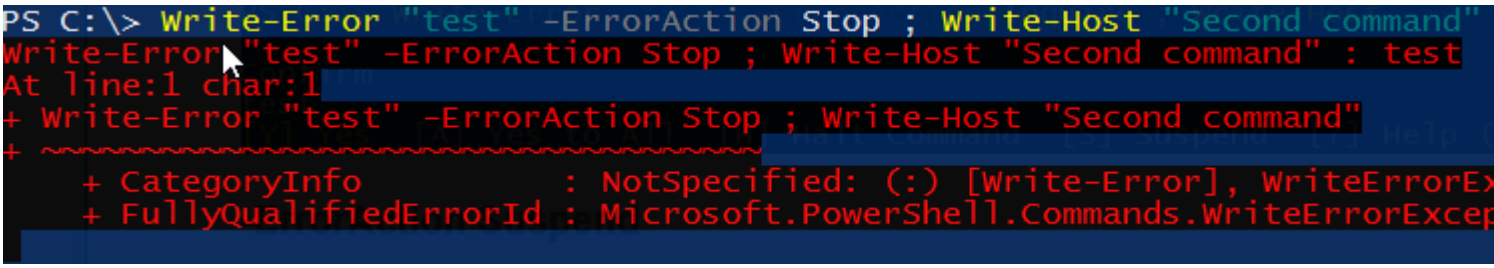
```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Seco
Second command
```

-ErrorAction Stop

Esta opción producirá un mensaje de error y no continuará con la ejecución.

```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
```



-ErrorAction Suspend

Solo disponible en flujos de trabajo Powershell. Cuando se usa, si el comando se ejecuta en un error, el flujo de trabajo se suspende. Esto permite la investigación de dicho error y ofrece la posibilidad de reanudar el flujo de trabajo. Para obtener más información sobre el sistema de flujo de trabajo, consulte [tema aún no creado].

Lea Parámetros comunes en línea: https://riptutorial.com/es/powershell/topic/5951/parametros-comunes

Capítulo 50: Parámetros dinámicos de PowerShell

Examples

Parámetro dinámico "simple"

Este ejemplo agrega un nuevo parámetro a MyTestFunction si `$SomeUsefulNumber` es mayor que 5.

```
function MyTestFunction
{
    [CmdletBinding(DefaultParameterSetName='DefaultConfiguration')]
    Param
    (
        [Parameter(Mandatory=$true)][int]$SomeUsefulNumber
    )

    DynamicParam
    {
        $paramDictionary = New-Object -Type
System.Management.Automation.RuntimeDefinedParameterDictionary
        $attributes = New-Object System.Management.Automation.ParameterAttribute
        $attributes.ParameterSetName = "__AllParameterSets"
        $attributes.Mandatory = $true
        $attributeCollection = New-Object -Type
System.Collections.ObjectModel.Collection[System.Attribute]
        $attributeCollection.Add($attributes)
        # If "SomeUsefulNumber" is greater than 5, then add the "MandatoryParam1" parameter
        if($SomeUsefulNumber -gt 5)
        {
            # Create a mandatory string parameter called "MandatoryParam1"
            $dynParam1 = New-Object -Type
System.Management.Automation.RuntimeDefinedParameter("MandatoryParam1", [String],
$attributeCollection)
            # Add the new parameter to the dictionary
            $paramDictionary.Add("MandatoryParam1", $dynParam1)
        }
        return $paramDictionary
    }

    process
    {
        Write-Host "SomeUsefulNumber = $SomeUsefulNumber"
        # Notice that dynamic parameters need a specific syntax
        Write-Host ("MandatoryParam1 = {0}" -f $PSBoundParameters.MandatoryParam1)
    }

}
```

Uso:

```
PS >  MyTestFunction -SomeUsefulNumber 3
SomeUsefulNumber = 3
```

```
MandatoryParam1 =

PS >  MyTestFunction -SomeUsefulNumber 6
cmdlet MyTestFunction at command pipeline position 1
Supply values for the following parameters:
MandatoryParam1:

PS >MyTestFunction -SomeUsefulNumber 6 -MandatoryParam1 test
SomeUsefulNumber = 6
MandatoryParam1 = test
```

En el segundo ejemplo de uso, puede ver claramente que falta un parámetro.

Los parámetros dinámicos también se tienen en cuenta con la finalización automática.
Esto es lo que sucede si presionas ctrl + espacio al final de la línea:

```
PS >MyTestFunction -SomeUsefulNumber 3 -<ctrl+space>
Verbose              WarningAction        WarningVariable      OutBuffer
Debug                InformationAction    InformationVariable  PipelineVariable
ErrorAction          ErrorVariable        OutVariable

PS >MyTestFunction -SomeUsefulNumber 6 -<ctrl+space>
MandatoryParam1      ErrorAction          ErrorVariable        OutVariable
Verbose              WarningAction        WarningVariable      OutBuffer
Debug                InformationAction    InformationVariable  PipelineVariable
```

Lea Parámetros dinámicos de PowerShell en línea:
https://riptutorial.com/es/powershell/topic/6704/parametros-dinamicos-de-powershell

Capítulo 51: Perfiles de Powershell

Observaciones

El archivo de perfil es un script de powershell que se ejecutará mientras se inicia la consola powershell. De esta manera, podemos tener nuestro entorno preparado para nosotros cada vez que iniciemos una nueva sesión de powershell.

Las cosas típicas que queremos hacer en el inicio de powershell son:

- Importación de módulos que usamos a menudo (ActiveDirectory, Exchange, algunas DLL específicas)
- explotación florestal
- cambiando el aviso
- diagnóstico

Hay varios archivos de perfil y ubicaciones que tienen diferentes usos y también una jerarquía de orden de inicio:

Anfitrión	Usuario	Camino	Orden de inicio	Variable
Todos	Todos	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ profile.ps1	1	$ profile.AllUsersAllHosts
Todos	Corriente	% USERPROFILE% \ Documents \ WindowsPowerShell \ profile.ps1	3	$ profile.CurrentUserAllHosts
Consola	Todos	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShell_profile.ps1	2	$ profile.AllUsersCurrentHost
Consola	Corriente	% USERPROFILE% \ Documents \ WindowsPowerShell \ Microsoft.PowerShell_profile.ps1	4	$ profile.CurrentUserCurrentHost
ISE	Todos	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShellISE_profile.ps1	2	$ profile.AllUsersCurrentHost
ISE	Corriente	% USERPROFILE% \ Documents \ WindowsPowerShell \ Microsoft.PowerShellISE_profile.ps1	4	$ profile.CurrentUserCurrentHost

Examples

Crear un perfil básico.

Un perfil de PowerShell se utiliza para cargar las funciones y variables definidas por el usuario automáticamente.

Los perfiles de PowerShell no se crean automáticamente para los usuarios.

Para crear un perfil de PowerShell `C:>New-Item -ItemType File $profile .`

Si está en ISE, puede usar el editor integrado en `C:>psEdit $profile`

Una manera fácil de comenzar con su perfil personal para el host actual es guardar algo de texto en la ruta almacenada en la variable `$profile`

```
"#Current host, current user" > $profile
```

Se pueden realizar modificaciones adicionales en el perfil utilizando PowerShell ISE, el bloc de notas, Visual Studio Code o cualquier otro editor.

La variable `$profile` -variable devuelve el perfil de usuario actual para el host actual de manera predeterminada, pero puede acceder a la ruta a la política de la máquina (todos los usuarios) y / o al perfil para todos los hosts (consola, ISE, terceros) utilizando es propiedades

```
PS> $PROFILE | Format-List -Force

AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost    :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts    : C:\Users\user\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost :
C:\Users\user\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Length                 : 75

PS> $PROFILE.AllUsersAllHosts
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
```

Lea Perfiles de Powershell en línea: https://riptutorial.com/es/powershell/topic/5636/perfiles-de-powershell

Capítulo 52: PowerShell "Streams"; Depuración, detallado, advertencia, error, salida e información

Observaciones

https://technet.microsoft.com/en-us/library/hh849921.aspx

Examples

Escritura-salida

`Write-Output` genera salida. Esta salida puede ir al siguiente comando después de la canalización o a la consola, por lo que simplemente se muestra.

El cmdlet envía objetos a la tubería principal, también conocida como "flujo de salida" o "canalización exitosa". Para enviar objetos de error por la tubería de error, use Write-Error.

```
# 1.) Output to the next Cmdlet in the pipeline
Write-Output 'My text' | Out-File -FilePath "$env:TEMP\Test.txt"

Write-Output 'Bob' | ForEach-Object {
    "My name is $_"
}

# 2.) Output to the console since Write-Output is the last command in the pipeline
Write-Output 'Hello world'

# 3.) 'Write-Output' CmdLet missing, but the output is still considered to be 'Write-Output'
'Hello world'
```

1. El cmdlet Write-Output envía el objeto especificado por la tubería al siguiente comando.
2. Si el comando es el último comando en la canalización, el objeto se muestra en la consola.
3. El intérprete de PowerShell trata esto como una salida-escritura implícita.

Debido a que el comportamiento predeterminado de `Write-Output` es mostrar los objetos al final de una canalización, generalmente no es necesario usar el Cmdlet. Por ejemplo, `Get-Process | Write-Output` es equivalente a `Get-Process` .

Preferencias de escritura

Los mensajes se pueden escribir con;

```
Write-Verbose "Detailed Message"
Write-Information "Information Message"
Write-Debug "Debug Message"
```

```
Write-Progress "Progress Message"
Write-Warning "Warning Message"
```

Cada uno de estos tiene una variable de preferencia;

```
$VerbosePreference = "SilentlyContinue"
$InformationPreference = "SilentlyContinue"
$DebugPreference = "SilentlyContinue"
$ProgressPreference = "Continue"
$WarningPreference = "Continue"
```

La variable de preferencia controla cómo se manejan el mensaje y la ejecución posterior del script;

```
$InformationPreference = "SilentlyContinue"
Write-Information "This message will not be shown and execution continues"

$InformationPreference = "Continue"
Write-Information "This message is shown and execution continues"

$InformationPreference = "Inquire"
Write-Information "This message is shown and execution will optionally continue"

$InformationPreference = "Stop"
Write-Information "This message is shown and execution terminates"
```

El color de los mensajes se puede controlar para `Write-Error` configurando;

```
$host.PrivateData.ErrorBackgroundColor = "Black"
$host.PrivateData.ErrorForegroundColor = "Red"
```

Configuraciones similares están disponibles para `Write-Verbose` , `Write-Debug` y `Write-Warning` .

Lea PowerShell "Streams"; Depuración, detallado, advertencia, error, salida e información en línea: https://riptutorial.com/es/powershell/topic/3255/powershell--streams---depuracion--detallado--advertencia--error--salida-e-informacion

Capítulo 53: Powershell Remoting

Observaciones

- about_Remote
- about_RemoteFAQ
- about_RemoteTroubleshooting

Examples

Habilitando el control remoto de PowerShell

El control remoto de PowerShell primero debe estar habilitado en el servidor al que desea conectarse de forma remota.

```
Enable-PSRemoting -Force
```

Este comando hace lo siguiente:

- Ejecuta el cmdlet Set-WSManQuickConfig, que realiza las siguientes tareas:
- Inicia el servicio WinRM.
- Establece el tipo de inicio en el servicio WinRM en Automático.
- Crea una escucha para aceptar solicitudes en cualquier dirección IP, si no existe una.
- Habilita una excepción de cortafuegos para comunicaciones WS-Management.
- Registra las configuraciones de sesión de Microsoft.PowerShell y Microsoft.PowerShell.Workflow, si aún no están registradas.
- Registra la configuración de la sesión de Microsoft.PowerShell32 en computadoras de 64 bits, si aún no está registrada.
- Habilita todas las configuraciones de sesión.
- Cambia el descriptor de seguridad de todas las configuraciones de sesión para permitir el acceso remoto.
- Reinicia el servicio WinRM para hacer efectivos los cambios anteriores.

Solo para entornos sin dominio

Para los servidores en un dominio de AD, la autenticación remota de PS se realiza a través de Kerberos ('Predeterminado'), o NTLM ('Negociar'). Si desea permitir la comunicación remota a un servidor que no sea de dominio, tiene dos opciones.

Configure la comunicación WSMan a través de HTTPS (que requiere la generación de certificados) o habilite la autenticación básica que envía sus credenciales a través del código codificado en base64 (básicamente es lo mismo que texto sin formato, así que tenga cuidado).

En cualquier caso, deberá agregar los sistemas remotos a su lista de hosts de confianza de

WSMan.

Habilitar la autenticación básica

```
Set-Item WSMan:\localhost\Service\AllowUnencrypted $true
```

A continuación, en el equipo que desee *conectar,* usted debe indicarle a confiar en el equipo *que* se está conectando.

```
Set-Item WSMan:\localhost\Client\TrustedHosts '192.168.1.1,192.168.1.2'
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *.contoso.com
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *
```

Importante : debe decirle a su cliente que confíe en la computadora direccionada de la manera en que desea conectarse (por ejemplo, si se conecta a través de IP, debe confiar en la IP, no en el nombre de host)

Conexión a un servidor remoto a través de PowerShell

Usando credenciales de su computadora local:

```
Enter-PSSession 192.168.1.1
```

Pedir credenciales en la computadora remota

```
Enter-PSSession 192.168.1.1 -Credential $(Get-Credential)
```

Ejecutar comandos en una computadora remota

Una vez que Powershell remoting esté habilitado (Enable-PSRemoting), puede ejecutar comandos en la computadora remota de esta manera:

```
Invoke-Command -ComputerName "RemoteComputerName" -ScriptBlock {
    Write host "Remote Computer Name: $ENV:ComputerName"
}
```

El método anterior crea una sesión temporal y la cierra justo después de que finalice el comando o scriptblock.

Para dejar la sesión abierta y ejecutar otro comando más adelante, primero debe crear una sesión remota:

```
$Session = New-PSSession -ComputerName "RemoteComputerName"
```

Luego, puede usar esta sesión cada vez que invoque comandos en la computadora remota:

```
Invoke-Command -Session $Session -ScriptBlock {
    Write host "Remote Computer Name: $ENV:ComputerName"
}

Invoke-Command -Session $Session -ScriptBlock {
    Get-Date
}
```

Si necesita usar diferentes credenciales, puede agregarlas con el parámetro `-Credential` :

```
$Cred = Get-Credential
Invoke-Command -Session $Session -Credential $Cred -ScriptBlock {...}
```

Advertencia de serialización remota

> Nota:
>
> Es importante saber que el control remoto serializa los objetos de PowerShell en el sistema remoto y los deserializa al final de la sesión remota, es decir, se convierten a XML durante el transporte y pierden todos sus métodos.

```
$output = Invoke-Command -Session $Session -ScriptBlock {
    Get-WmiObject -Class win32_printer
}

$output | Get-Member -MemberType Method

   TypeName: Deserialized.System.Management.ManagementObject#root\cimv2\Win32_Printer

Name      MemberType Definition
----      ---------- ----------
GetType   Method     type GetType()
ToString  Method     string ToString(), string ToString(string format, System.IFormatProvi...
```

Mientras que tienes los métodos en el objeto PS normal:

```
Get-WmiObject -Class win32_printer | Get-Member -MemberType Method

 TypeName: System.Management.ManagementObject#root\cimv2\Win32_Printer

Name                   MemberType Definition

----                   ---------- ----------

CancelAllJobs          Method     System.Management.ManagementBaseObject CancelAllJobs()

GetSecurityDescriptor Method     System.Management.ManagementBaseObject
GetSecurityDescriptor()
Pause                  Method     System.Management.ManagementBaseObject Pause()

PrintTestPage          Method     System.Management.ManagementBaseObject PrintTestPage()
```

```
RenamePrinter          Method     System.Management.ManagementBaseObject
RenamePrinter(System.String NewPrinterName)
Reset                  Method     System.Management.ManagementBaseObject Reset()

Resume                 Method     System.Management.ManagementBaseObject Resume()

SetDefaultPrinter      Method     System.Management.ManagementBaseObject SetDefaultPrinter()

SetPowerState          Method     System.Management.ManagementBaseObject
SetPowerState(System.UInt16 PowerState, System.String Time)
SetSecurityDescriptor Method     System.Management.ManagementBaseObject
SetSecurityDescriptor(System.Management.ManagementObject#Win32_SecurityDescriptor Descriptor)
```

Uso del argumento

Para usar los argumentos como parámetros para el bloque de secuencias de comandos remotos, uno puede usar el parámetro `ArgumentList` de `Invoke-Command` o la sintaxis de `$Using:` .

Usando `ArgumentList` con parámetros sin nombre (es decir, en el orden en que se pasan al bloque de secuencias de comandos):

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $args[0]
    Remove-Item -Path $args[1] -ErrorAction SilentlyContinue -Force
}
```

Usando `ArgumentList` con parámetros nombrados:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Param($serviceToShowInRemoteSession,$fileToDelete)

    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $serviceToShowInRemoteSession
    Remove-Item -Path $fileToDelete -ErrorAction SilentlyContinue -Force
}
```

Utilizando `$Using:` sintaxis:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ScriptBlock {
    Get-Service $Using:servicesToShow
    Remove-Item -Path $fileName -ErrorAction SilentlyContinue -Force
}
```

Una buena práctica para la limpieza automática de PSSessions

Cuando se crea una sesión remota a través del cmdlet `New-PSsession` , la PSSession continúa hasta que finaliza la sesión actual de PowerShell. Lo que significa que, de forma predeterminada, la `PSSession` y todos los recursos asociados se seguirán utilizando hasta que finalice la sesión actual de PowerShell.

Múltiples `PSSessions` activas pueden convertirse en una carga para los recursos, en particular para los scripts de larga ejecución o interconectados que crean cientos de `PSSessions` en una sola sesión de PowerShell.

Es una buena práctica eliminar explícitamente cada `PSSession` después de que se termine de usar. [1]

La siguiente plantilla de código utiliza `try-catch-finally` para lograr lo anterior, combinando el manejo de errores con una forma segura de asegurar que todas las `PSSessions` creadas se eliminen cuando se terminen de usar:

```
try
{
    $session = New-PSsession -Computername "RemoteMachineName"
    Invoke-Command -Session $session -ScriptBlock {write-host "This is running on
$ENV:ComputerName"}
}
catch
{
    Write-Output "ERROR: $_"
}
finally
{
    if ($session)
    {
        Remove-PSSession $session
    }
}
```

Referencias: [1] https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/new-pssession

Lea Powershell Remoting en línea: https://riptutorial.com/es/powershell/topic/3087/powershell-remoting

Capítulo 54: Propiedades calculadas

Introducción

Las propiedades calculadas en Powershell son propiedades derivadas personalizadas (calculadas). Le permite al usuario dar formato a una determinada propiedad de la forma en que quiere que sea. El cálculo (expresión) puede ser una cosa muy posiblemente.

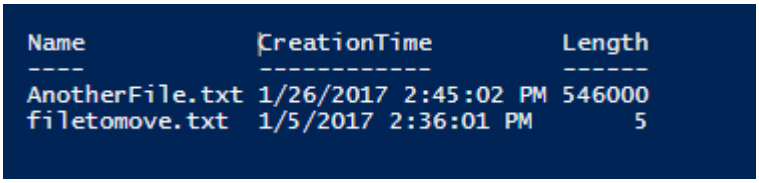
Examples

Mostrar tamaño de archivo en KB - Propiedades calculadas

Consideremos el siguiente fragmento,

```
Get-ChildItem -Path C:\MyFolder | Select-Object Name, CreationTime, Length
```

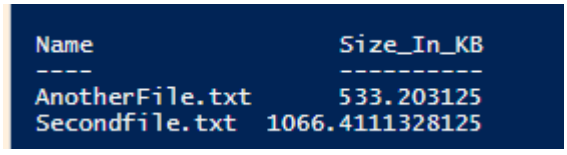
Simplemente genera el contenido de la carpeta con las propiedades seleccionadas. Algo como,



¿Qué pasa si quiero mostrar el tamaño del archivo en KB? Aquí es donde las propiedades calculadas son útiles.

```
Get-ChildItem C:\MyFolder | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length /
1Kb}}
```

Lo que produce,



La `Expression` es lo que contiene el cálculo para la propiedad calculada. Y sí, puede ser cualquier cosa!

Lea Propiedades calculadas en línea:
https://riptutorial.com/es/powershell/topic/8913/propiedades-calculadas

Capítulo 55: PSScriptAnalyzer - Analizador de scripts de PowerShell

Introducción

PSScriptAnalyzer, https://github.com/PowerShell/PSScriptAnalyzer , es un comprobador de código estático para los módulos y scripts de Windows PowerShell. PSScriptAnalyzer verifica la calidad del código de Windows PowerShell ejecutando un conjunto de reglas basadas en las mejores prácticas de PowerShell identificadas por el equipo de PowerShell y la comunidad. Genera Resultados de Diagnóstico (errores y advertencias) para informar a los usuarios sobre posibles defectos de código y sugiere posibles soluciones para mejoras.

```
PS> Install-Module -Name PSScriptAnalyzer
```

Sintaxis

1. `Get-ScriptAnalyzerRule [-CustomizedRulePath <string[]>] [-Name <string[]>] [-Severity <string[]>] [<CommonParameters>]`
2. `Invoke-ScriptAnalyzer [-Path] <string> [-CustomizedRulePath <string[]>] [-ExcludeRule <string[]>] [-IncludeRule<string[]>] [-Severity <string[]>] [-Recurse] [-SuppressedOnly] [<CommonParameters>]`

Examples

Análisis de scripts con los conjuntos de reglas preestablecidos incorporados

ScriptAnalyzer incluye conjuntos de reglas predefinidas incorporadas que se pueden usar para analizar scripts. Estos incluyen: `PSGallery` , `DSC` y `CodeFormatting` . Se pueden ejecutar de la siguiente manera:

Reglas de la Galería PowerShell

Para ejecutar las reglas de la Galería PowerShell use el siguiente comando:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

Reglas DSC

Para ejecutar las reglas DSC use el siguiente comando:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings DSC -Recurse
```

Reglas de formato de código

Para ejecutar las reglas de formateo de código use el siguiente comando:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings CodeFormatting -Recurse
```

Analizar scripts contra cada regla incorporada

Para ejecutar el analizador de scripts contra un solo archivo de script, ejecute:

```
Invoke-ScriptAnalyzer -Path myscript.ps1
```

Esto analizará su secuencia de comandos en contra de cada regla incorporada. Si su secuencia de comandos es lo suficientemente grande como para dar lugar a una gran cantidad de advertencias y / o errores.

Para ejecutar el analizador de secuencias de comandos en un directorio completo, especifique la carpeta que contiene la secuencia de comandos, el módulo y los archivos DSC que desea analizar. Especifique el parámetro Recurse si también desea que los subdirectorios busquen archivos para analizar.

```
Invoke-ScriptAnalyzer -Path . -Recurse
```

Listar todas las reglas incorporadas

Para ver todas las reglas incorporadas ejecute:

```
Get-ScriptAnalyzerRule
```

Lea PSScriptAnalyzer - Analizador de scripts de PowerShell en línea: https://riptutorial.com/es/powershell/topic/9619/psscriptanalyzer---analizador-de-scripts-de-powershell

Capítulo 56: Reconocimiento de Amazon Web Services (AWS)

Introducción

Amazon Rekognition es un servicio que facilita agregar análisis de imágenes a sus aplicaciones. Con Rekognition, puede detectar objetos, escenas y rostros en las imágenes. También puedes buscar y comparar caras. La API de Rekognition le permite agregar rápidamente la búsqueda visual sofisticada basada en el aprendizaje profundo y la clasificación de imágenes a sus aplicaciones.

Examples

Detectar etiquetas de imagen con AWS Rekognition

```
$BucketName = 'trevorrekognition'
$FileName = 'kitchen.jpg'

New-S3Bucket -BucketName $BucketName
Write-S3Object -BucketName $BucketName -File $FileName
$REKResult = Find-REKLabel -Region us-east-1 -ImageBucket $BucketName -ImageName $FileName

$REKResult.Labels
```

Después de ejecutar el script anterior, debería tener resultados impresos en su host de PowerShell que tengan un aspecto similar al siguiente:

```
RESULTS:

Confidence Name
---------- ----
86.87605   Indoors
86.87605   Interior Design
86.87605   Room
77.4853    Kitchen
77.25354   Housing
77.25354   Loft
66.77325   Appliance
66.77325   Oven
```

Al usar el módulo AWS PowerShell junto con el servicio de reconocimiento de AWS, puede detectar etiquetas en una imagen, como identificar objetos en una habitación, los atributos sobre las fotos que tomó y el nivel de confianza correspondiente que tiene el reconocimiento de AWS para cada uno de esos atributos.

El comando `Find-REKLabel` es el que le permite invocar una búsqueda de estos atributos / etiquetas. Si bien puede proporcionar contenido de imagen como una matriz de bytes durante la llamada a la API, un mejor método es cargar sus archivos de imagen en un AWS S3 Bucket y

luego apuntar el servicio de Reconocimiento a los Objetos S3 que desea analizar. El ejemplo anterior muestra cómo lograr esto.

Compare la similitud facial con el reconocimiento de AWS

```
$BucketName = 'trevorrekognition'

### Create a new AWS S3 Bucket
New-S3Bucket -BucketName $BucketName

### Upload two different photos of myself to AWS S3 Bucket
Write-S3Object -BucketName $BucketName -File myphoto1.jpg
Write-S3Object -BucketName $BucketName -File myphoto2.jpg

### Perform a facial comparison between the two photos with AWS Rekognition
$Comparison = @{
    SourceImageBucket = $BucketName
    TargetImageBucket = $BucketName
    SourceImageName = 'myphoto1.jpg'
    TargetImageName = 'myphoto2.jpg'
    Region = 'us-east-1'
}
$Result = Compare-REKFace @Comparison
$Result.FaceMatches
```

La secuencia de comandos de ejemplo proporcionada anteriormente debería proporcionarle resultados similares a los siguientes:

```
Face                                Similarity
----                                ----------
Amazon.Rekognition.Model.ComparedFace 90
```

El servicio de reconocimiento de AWS le permite realizar una comparación facial entre dos fotos. El uso de este servicio es bastante sencillo. Simplemente cargue dos archivos de imagen, que desea comparar, a un AWS S3 Bucket. Luego, invoque el comando `Compare-REKFace` , similar al ejemplo proporcionado anteriormente. Por supuesto, deberá proporcionar su propio nombre de S3 Bucket y nombres de archivo únicos a nivel mundial.

Lea Reconocimiento de Amazon Web Services (AWS) en línea:
https://riptutorial.com/es/powershell/topic/9581/reconocimiento-de-amazon-web-services--aws-

Capítulo 57: Salpicaduras

Introducción

Splatting es un método para pasar múltiples parámetros a un comando como una sola unidad. Esto se hace almacenando los parámetros y sus valores como pares clave-valor en una tabla hash y dividiéndolos en un cmdlet usando el operador de distribución `@` .

Splatting puede hacer que un comando sea más legible y le permite reutilizar parámetros en múltiples llamadas de comando.

Observaciones

Nota: El operador de expresión de Array o `@()` tienen un comportamiento muy diferente al del operador de Splatting `@` .

Lea más en about_Splatting @ TechNet

Examples

Parámetros de salpicadura

La salpicadura se realiza reemplazando el signo de dólar `$` con el operador de salpicadura `@` cuando se usa una variable que contiene una HashTable de parámetros y valores en una llamada de comando.

```
$MyParameters = @{
    Name = "iexplore"
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Sin salpicaduras:

```
Get-Process -Name "iexplore" -FileVersionInfo
```

Puede combinar parámetros normales con parámetros distribuidos para agregar fácilmente parámetros comunes a sus llamadas.

```
$MyParameters = @{
    ComputerName = "StackOverflow-PC"
}

Get-Process -Name "iexplore" @MyParameters

Invoke-Command -ScriptBlock { "Something to excute remotely" } @MyParameters
```

Pasando un parámetro Switch usando Splatting

Para usar Splatting para llamar a `Get-Process` con el interruptor `-FileVersionInfo` similar a esto:

```
Get-Process -FileVersionInfo
```

Esta es la llamada utilizando splatting:

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Nota: esto es útil porque puede crear un conjunto predeterminado de parámetros y hacer la llamada muchas veces de esta manera

```
$MyParameters = @{
    FileVersionInfo = $true
}

Get-Process @MyParameters -Name WmiPrvSE
Get-Process @MyParameters -Name explorer
```

Tubería y salpicaduras

Declarar el splat es útil para reutilizar conjuntos de parámetros varias veces o con ligeras variaciones:

```
$splat = @{
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
   ErrorAction = "Stop"
}

Get-WmiObject -ComputerName $env:COMPUTERNAME @splat
Get-WmiObject -ComputerName "Computer2" @splat
Get-WmiObject -ComputerName "Computer3" @splat
```

Sin embargo, si el splat no tiene sangría para su reutilización, es posible que no desee declararlo. Se puede canalizar en su lugar:

```
@{
   ComputerName = $env:COMPUTERNAME
   Class = "Win32_SystemEnclosure"
   Property = "Manufacturer"
   ErrorAction = "Stop"
} | % { Get-WmiObject @_ }
```

Splatting de la función de nivel superior a una serie de funciones internas

Sin salpicaduras es muy complicado intentar pasar los valores a través de la pila de llamadas. Pero si combina splatting con la potencia de **@PSBoundParameters** , puede pasar la colección de parámetros de nivel superior a través de las capas.

```
Function Outer-Method
{
    Param
    (
        [string]
        $First,

        [string]
        $Second
    )

    Write-Host ($First) -NoNewline

    Inner-Method @PSBoundParameters
}

Function Inner-Method
{
    Param
    (
        [string]
        $Second
    )

    Write-Host (" {0}!" -f $Second)
}

$parameters = @{
    First = "Hello"
    Second = "World"
}


Outer-Method @parameters
```

Lea Salpicaduras en línea: https://riptutorial.com/es/powershell/topic/5647/salpicaduras

Capítulo 58: Seguridad y criptografía

Examples

Cálculo de los códigos hash de una cadena a través de .Net Cryptography

Utilizando .Net `System.Security.Cryptography.HashAlgorithm` espacio de nombres para generar el código hash del mensaje con los algoritmos compatibles.

```
$example="Nobody expects the Spanish Inquisition."

#calculate
$hash=[System.Security.Cryptography.HashAlgorithm]::Create("sha256").ComputeHash(
[System.Text.Encoding]::UTF8.GetBytes($example))

#convert to hex
[System.BitConverter]::ToString($hash)

#2E-DF-DA-DA-56-52-5B-12-90-FF-16-FB-17-44-CF-B4-82-DD-29-14-FF-BC-B6-49-79-0C-0E-58-9E-46-2D-
3D
```

La parte `"sha256"` era el algoritmo hash utilizado.

el `-` se puede quitar o cambiar a minúsculas

```
#convert to lower case hex without '-'
[System.BitConverter]::ToString($hash).Replace("-","").ToLower()

#2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d
```

Si se prefiere el formato base64, usar el convertidor base64 para la salida

```
#convert to base64
[Convert]::ToBase64String($hash)

#Lt/a2lZSWxKQ/xb7F0TPtILdKRT/vLZJeQwOWJ5GLT0=
```

Lea Seguridad y criptografía en línea: https://riptutorial.com/es/powershell/topic/5683/seguridad-y-criptografia

Capítulo 59: Servicio de almacenamiento simple de Amazon Web Services (AWS) (S3)

Introducción

Esta sección de documentación se centra en el desarrollo contra el Servicio de almacenamiento simple (S3) de Amazon Web Services (AWS). S3 es realmente un servicio simple para interactuar. Crea "cubos" de S3 que pueden contener cero o más "objetos". Una vez que crea un grupo, puede cargar archivos o datos arbitrarios en el grupo S3 como un "objeto". Hace referencia a los objetos S3, dentro de un grupo, por la "clave" (nombre) del objeto.

Parámetros

Parámetro	Detalles
BucketName	El nombre del grupo AWS S3 en el que está operando.
CannedACLName	El nombre de la Lista de control de acceso (ACL) incorporada (predefinida) que se asociará con el grupo S3.
Expediente	El nombre de un archivo en el sistema de archivos local que se cargará a AWS S3 Bucket.

Examples

Crear un nuevo cubo S3

```
New-S3Bucket -BucketName trevor
```

El nombre del depósito del Servicio de almacenamiento simple (S3) debe ser globalmente único. Esto significa que si alguien más ya ha usado el nombre del depósito que desea usar, entonces debe decidir un nombre nuevo.

Cargar un archivo local en un cubo S3

```
Set-Content -Path myfile.txt -Value 'PowerShell Rocks'
Write-S3Object -BucketName powershell -File myfile.txt
```

Cargar archivos de su sistema de archivos local en AWS S3 es fácil, utilizando el comando `Write-S3Object` . En su forma más básica, solo necesita especificar el parámetro `-BucketName` , para indicar en qué grupo de S3 desea cargar un archivo, y el parámetro `-File` , que indica la ruta relativa o absoluta al archivo local que desea subir en el cubo S3.

Eliminar un S3 Bucket

```
Get-S3Object -BucketName powershell | Remove-S3Object -Force
Remove-S3Bucket -BucketName powershell -Force
```

Para eliminar una cubeta S3, primero debe eliminar todos los objetos S3 que se almacenan dentro de la cubeta, siempre que tenga permiso para hacerlo. En el ejemplo anterior, estamos recuperando una lista de todos los objetos dentro de un cubo, y luego `Remove-S3Object` comando `Remove-S3Object` para eliminarlos. Una vez que se hayan eliminado todos los objetos, podemos usar el comando `Remove-S3Bucket` para eliminar el depósito.

Lea Servicio de almacenamiento simple de Amazon Web Services (AWS) (S3) en línea: https://riptutorial.com/es/powershell/topic/9579/servicio-de-almacenamiento-simple-de-amazon-web-services--aws---s3-

Capítulo 60: Set básico de operaciones

Introducción

Un conjunto es una colección de elementos que puede ser cualquier cosa. Cualquiera que sea el operador que necesitamos para trabajar en estos conjuntos son, en resumen, los *operadores establecidos* y la operación también se conoce como *operación establecida* . La operación de configuración básica incluye Unión, Intersección, así como suma, resta, etc.

Sintaxis

- Objeto grupal
- Group-Object -Property <propertyName>
- Group-Object -Property <propertyName>, <propertyName2>
- Group-Object -Property <propertyName> -CaseSensitive
- Group-Object -Property <propertyName> -Culture <culture>
- Objeto de grupo - Propiedad <ScriptBlock>
- Ordenar-Objeto
- Sort-Object -Property <propertyName>
- Sort-Object -Property <ScriptBlock>
- Sort-Object -Property <propertyName>, <propertyName2>
- Sort-Object -Property <propertyObject> -CaseSensitive
- Sort-Object -Property <propertyObject> -Descending
- Sort-Object -Property <propertyObject> -Unique
- Sort-Object -Property <propertyObject> -Culture <culture>

Examples

Filtrado: ¿Dónde-Objeto / dónde /?

Filtra una enumeración usando una expresión condicional

Sinónimos:

```
Where-Object
where
?
```

Ejemplo:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Where-Object { $_ -like "A*" }
$names | where { $_ -like "A*" }
$names | ? { $_ -like "A*" }
```

Devoluciones:

Aaron
Albert
Alphonse

Ordenar: Ordenar-Objeto / ordenar

Ordenar una enumeración en orden ascendente o descendente

Sinónimos:

```
Sort-Object
sort
```

Asumiendo:

```
$names = @( "Aaron", "Aaron", "Bernie", "Charlie", "Danny" )
```

El ordenamiento ascendente es el predeterminado:

```
$names | Sort-Object
$names | sort
```

Aaron
Aaron
Bernie
Charlie
Danny

Para solicitar el orden descendente:

```
$names | Sort-Object -Descending
$names | sort -Descending
```

Danny
Charlie

Bernie
Aaron
Aaron

Puedes ordenar usando una expresión.

```
$names | Sort-Object { $_.length }
```

Aaron
Aaron
Danny
Bernie
Charlie

Agrupación: Grupo-Objeto / grupo

Puede agrupar una enumeración basada en una expresión.

Sinónimos:

```
Group-Object
group
```

Ejemplos:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Group-Object -Property Length
$names | group -Property Length
```

Respuesta:

Contar	Nombre	Grupo
4	5	{Aaron, Danny, Ernie, Frank}
2	6	{Albert, Bernie}
1	8	{Alphonse}
1	7	{Charlie}

Proyección: Seleccionar-objeto / seleccionar

Proyectar una enumeración le permite extraer miembros específicos de cada objeto, extraer todos los detalles o calcular valores para cada objeto

Sinónimos:

```
Select-Object
select
```

Seleccionando un subconjunto de las propiedades:

```
$dir = dir "C:\MyFolder"

$dir | Select-Object Name, FullName, Attributes
$dir | select Name, FullName, Attributes
```

Nombre	Nombre completo	Atributos
Imágenes	C: \ MyFolder \ Images	Directorio
data.txt	C: \ MyFolder \ data.txt	Archivo
fuente.c	C: \ MyFolder \ source.c	Archivo

Seleccionando el primer elemento, y mostrar todas sus propiedades:

```
$d | select -first 1 *
```

PSPath
PSParentPath
PSChildName
PSDrive
PSProvider
PSIsContainer
Nombre base
Modo
Nombre
Padre
Existe
Raíz
Nombre completo
Extensión

Tiempo de creación
CreationTimeUtc
LastAccessTime
LastAccessTimeUtc
LastWriteTime
LastWriteTimeUtc
Atributos

Lea Set básico de operaciones en línea: https://riptutorial.com/es/powershell/topic/1557/set-basico-de-operaciones

Capítulo 61: Trabajando con archivos XML

Examples

Accediendo a un archivo XML

```
<!-- file.xml -->
<people>
    <person id="101">
        <name>Jon Lajoie</name>
        <age>22</age>
    </person>
    <person id="102">
        <name>Lord Gaben</name>
        <age>65</age>
    </person>
    <person id="103">
        <name>Gordon Freeman</name>
        <age>29</age>
    </person>
</people>
```

Cargando un archivo XML

Para cargar un archivo XML, puede utilizar cualquiera de estos:

```
# First Method
$xdoc = New-Object System.Xml.XmlDocument
$file = Resolve-Path(".\file.xml")
$xdoc.load($file)

# Second Method
[xml] $xdoc = Get-Content ".\file.xml"

# Third Method
$xdoc = [xml] (Get-Content ".\file.xml")
```

Acceso a XML como objetos

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.people

person
--------
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.people.person

id                                      name                                    age
--                                      ----                                    ---
```

```
101                                     Jon Lajoie                              22
102                                     Lord Gaben                              65
103                                     Gordon Freeman                          29

PS C:\> $xml.people.person[0].name
Jon Lajoie

PS C:\> $xml.people.person[1].age
65

PS C:\> $xml.people.person[2].id
103
```

Accediendo a XML con XPath

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.SelectNodes("//people")

person
--------
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.SelectNodes("//people//person")

id                                      name                                    age
--                                      ----                                    ---
101                                     Jon Lajoie                              22
102                                     Lord Gaben                              65
103                                     Gordon Freeman                          29

PS C:\> $xml.SelectSingleNode("people//person[1]//name")
Jon Lajoie

PS C:\> $xml.SelectSingleNode("people//person[2]//age")
65

PS C:\> $xml.SelectSingleNode("people//person[3]//@id")
103
```

Acceso a XML que contiene espacios de nombres con XPath

```
PS C:\> [xml]$xml = @"
<ns:people xmlns:ns="http://schemas.xmlsoap.org/soap/envelope/">
    <ns:person id="101">
        <ns:name>Jon Lajoie</ns:name>
    </ns:person>
    <ns:person id="102">
        <ns:name>Lord Gaben</ns:name>
    </ns:person>
    <ns:person id="103">
        <ns:name>Gordon Freeman</ns:name>
    </ns:person>
</ns:people>
"@

PS C:\> $ns = new-object Xml.XmlNamespaceManager $xml.NameTable
```

```
PS C:\> $ns.AddNamespace("ns", $xml.DocumentElement.NamespaceURI)
PS C:\> $xml.SelectNodes("//ns:people/ns:person", $ns)

id                                      name
--                                      ----
101                                     Jon Lajoie
102                                     Lord Gaben
103                                     Gordon Freeman
```

Creando un documento XML usando XmlWriter ()

```
# Set The Formatting
$xmlsettings = New-Object System.Xml.XmlWriterSettings
$xmlsettings.Indent = $true
$xmlsettings.IndentChars = "    "

# Set the File Name Create The Document
$XmlWriter = [System.XML.XmlWriter]::Create("C:\YourXML.xml", $xmlsettings)

# Write the XML Decleration and set the XSL
$xmlWriter.WriteStartDocument()
$xmlWriter.WriteProcessingInstruction("xml-stylesheet", "type='text/xsl' href='style.xsl'")

# Start the Root Element
$xmlWriter.WriteStartElement("Root")

    $xmlWriter.WriteStartElement("Object") # <-- Start <Object>

        $xmlWriter.WriteElementString("Property1","Value 1")
        $xmlWriter.WriteElementString("Property2","Value 2")

        $xmlWriter.WriteStartElement("SubObject") # <-- Start <SubObject>
            $xmlWriter.WriteElementString("Property3","Value 3")
        $xmlWriter.WriteEndElement() # <-- End <SubObject>

    $xmlWriter.WriteEndElement() # <-- End <Object>

$xmlWriter.WriteEndElement() # <-- End <Root>

# End, Finalize and close the XML Document
$xmlWriter.WriteEndDocument()
$xmlWriter.Flush()
$xmlWriter.Close()
```

Archivo XML de salida

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<Root>
    <Object>
        <Property1>Value 1</Property1>
        <Property2>Value 2</Property2>
        <SubObject>
            <Property3>Value 3</Property3>
        </SubObject>
    </Object>
</Root>
```

Data de muestra

Documento XML

Primero, definamos un documento XML de muestra llamado " **books.xml** " en nuestro directorio actual:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book>
        <title>Of Mice And Men</title>
        <author>John Steinbeck</author>
        <pageCount>187</pageCount>
        <publishers>
            <publisher>
                <isbn>978-88-58702-15-4</isbn>
                <name>Pascal Covici</name>
                <year>1937</year>
                <binding>Hardcover</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-05-82461-46-8</isbn>
                <name>Longman</name>
                <year>2009</year>
                <binding>Hardcover</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="Lennie Small" />
            <character name="Curley's Wife" />
            <character name="George Milton" />
            <character name="Curley" />
        </characters>
        <film>True</film>
    </book>
    <book>
        <title>The Hunt for Red October</title>
        <author>Tom Clancy</author>
        <pageCount>387</pageCount>
        <publishers>
            <publisher>
                <isbn>978-08-70212-85-7</isbn>
                <name>Naval Institute Press</name>
                <year>1984</year>
                <binding>Hardcover</binding>
                <first>true</first>
            </publisher>
            <publisher>
                <isbn>978-04-25083-83-3</isbn>
                <name>Berkley</name>
                <year>1986</year>
                <binding>Paperback</binding>
            </publisher>
```

```
            <publisher>
                <isbn>978-08-08587-35-4</isbn>
                <name>Penguin Putnam</name>
                <year>2010</year>
                <binding>Paperback</binding>
            </publisher>
        </publishers>
        <characters>
            <character name="Marko Alexadrovich Ramius" />
            <character name="Jack Ryan" />
            <character name="Admiral Greer" />
            <character name="Bart Mancuso" />
            <character name="Vasily Borodin" />
        </characters>
        <film>True</film>
    </book>
</books>
```

Nuevos datos

Lo que queremos hacer es agregar algunos libros nuevos a este documento, digamos *Patriot Games* por Tom Clancy (sí, soy un fan de los trabajos de Clancy ^ __ ^) y un favorito de ciencia ficción: *La guía del autoestopista de la galaxia.* por Douglas Adams principalmente porque Zaphod Beeblebrox es divertido de leer.

De alguna manera, hemos adquirido los datos para los nuevos libros y los hemos guardado como una lista de PSCustomObjects:

```
$newBooks = @(
    [PSCustomObject] @{
        "Title" = "Patriot Games";
        "Author" = "Tom Clancy";
        "PageCount" = 540;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-39-913241-4";
                "Year" = "1987";
                "First" = $True;
                "Name" = "Putnam";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Jack Ryan", "Prince of Wales", "Princess of Wales",
            "Robby Jackson", "Cathy Ryan", "Sean Patrick Miller"
        );
        "film" = $True;
    },
    [PSCustomObject] @{
        "Title" = "The Hitchhiker's Guide to the Galaxy";
        "Author" = "Douglas Adams";
        "PageCount" = 216;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-33-025864-7";
                "Year" = "1979";
                "First" = $True;
```

```
                "Name" = "Pan Books";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Arthur Dent", "Marvin", "Zaphod Beeblebrox", "Ford Prefect",
            "Trillian", "Slartibartfast", "Dirk Gently"
        );
        "film" = $True;
    }
);
```

Plantillas

Ahora necesitamos definir unas pocas estructuras XML esqueléticas para que nuestros nuevos datos ingresen. Básicamente, desea crear un esqueleto / plantilla para cada lista de datos. En nuestro ejemplo, eso significa que necesitamos una plantilla para el libro, los personajes y los editores. También podemos usar esto para definir algunos valores predeterminados, como el valor de la etiqueta de la `film` .

```
$t_book = [xml] @'
<book>
    <title />
    <author />
    <pageCount />
    <publishers />
    <characters />
    <film>False</film>
</book>
'@;

$t_publisher = [xml] @'
<publisher>
    <isbn/>
    <name/>
    <year/>
    <binding/>
    <first>false</first>
</publisher>
'@;

$t_character = [xml] @'
<character name="" />
'@;
```

Hemos terminado con la configuración.

Añadiendo los nuevos datos.

Ahora que todos estamos configurados con nuestros datos de muestra, agreguemos los objetos personalizados al objeto de documento XML.

```
# Read the xml document
$xml = [xml] Get-Content .\books.xml;

# Let's show a list of titles to see what we've got currently:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};;

# Outputs:
# title                                author         ISBN
# -----                                ------         ----
# Of Mice And Men                      John Steinbeck 978-88-58702-15-4
# The Hunt for Red October             Tom Clancy     978-08-70212-85-7

# Let's show our new books as well:
$newBooks | Select Title, Author, @{N="ISBN";E={$_.Publishers[0].ISBN}};

# Outputs:
# Title                                Author        ISBN
# -----                                ------        ----
# Patriot Games                        Tom Clancy    978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams 978-0-33-025864-7

# Now to merge the two:

ForEach ( $book in $newBooks ) {
    $root = $xml.SelectSingleNode("/books");

    # Add the template for a book as a new node to the root element
    [void]$root.AppendChild($xml.ImportNode($t_book.book, $true));

    # Select the new child element
    $newElement = $root.SelectSingleNode("book[last()]");

    # Update the parameters of that new element to match our current new book data
    $newElement.title     = [String]$book.Title;
    $newElement.author    = [String]$book.Author;
    $newElement.pageCount = [String]$book.PageCount;
    $newElement.film      = [String]$book.Film;

    # Iterate through the properties that are Children of our new Element:
    ForEach ( $publisher in $book.Publishers ) {
        # Create the new child publisher element
        # Note the use of "SelectSingleNode" here, this allows the use of the "AppendChild"
method as it returns
        # a XmlElement type object instead of the $Null data that is currently stored in that
leaf of the
        # XML document tree

[void]$newElement.SelectSingleNode("publishers").AppendChild($xml.ImportNode($t_publisher.publisher,
$true));

        # Update the attribute and text values of our new XML Element to match our new data
        $newPublisherElement = $newElement.SelectSingleNode("publishers/publisher[last()]");
        $newPublisherElement.year = [String]$publisher.Year;
        $newPublisherElement.name = [String]$publisher.Name;
        $newPublisherElement.binding = [String]$publisher.Binding;
        $newPublisherElement.isbn = [String]$publisher.ISBN;
        If ( $publisher.first ) {
            $newPublisherElement.first = "True";
        }
    }
```

```
    ForEach ( $character in $book.Characters ) {
        # Select the characters xml element
        $charactersElement = $newElement.SelectSingleNode("characters");

        # Add a new character child element
        [void]$charactersElement.AppendChild($xml.ImportNode($t_character.character, $true));

        # Select the new characters/character element
        $characterElement = $charactersElement.SelectSingleNode("character[last()]");

        # Update the attribute and text values to match our new data
        $characterElement.name = [String]$character;
    }
}

# Check out the new XML:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_.Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};

# Outputs:
# title                              author         ISBN
# -----                              ------         ----
# Of Mice And Men                    John Steinbeck 978-88-58702-15-4
# The Hunt for Red October           Tom Clancy     978-08-70212-85-7
# Patriot Games                      Tom Clancy     978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams  978-0-33-025864-7
```

¡Ahora podemos escribir nuestro XML en disco, pantalla, web o en cualquier lugar!

Lucro

Si bien este puede no ser el procedimiento para todos, lo encontré para ayudar a evitar un montón de

`[void]$xml.SelectSingleNode("/complicated/xpath/goes[here]").AppendChild($xml.CreateElement("newElementN`

seguido de `$xml.SelectSingleNode("/complicated/xpath/goes/here/newElementName") = $textValue`

Creo que el método detallado en el ejemplo es más limpio y más fácil de analizar para los humanos normales.

Mejoras

Puede ser posible cambiar la plantilla para incluir elementos con niños en lugar de desglosar cada sección como una plantilla separada. Solo tienes que tener cuidado de clonar el elemento anterior cuando recorres la lista.

Lea Trabajando con archivos XML en línea:
https://riptutorial.com/es/powershell/topic/4882/trabajando-con-archivos-xml

Capítulo 62: Trabajando con la tubería de PowerShell

Introducción

PowerShell presenta un modelo de canalización de objetos, que le permite enviar objetos enteros a través de la tubería para consumir comandos o (al menos) la salida. A diferencia de la canalización clásica basada en cadenas, la información de los objetos canalizados no tiene que estar en posiciones específicas. Los Commandlets pueden declarar que interactúan con los objetos de la canalización como entrada, mientras que los valores de retorno se envían automáticamente a la canalización.

Sintaxis

- COMENZAR El primer bloque. Ejecutado una vez al principio. La entrada de canalización aquí es $ null, ya que no se ha establecido.
- PROCESO El segundo bloque. Ejecutado para cada elemento de la tubería. El parámetro pipeline es igual al elemento procesado actualmente.
- FIN del último bloque. Ejecutado una vez al final. El parámetro de canalización es igual al último elemento de la entrada, porque no se ha cambiado desde que se estableció.

Observaciones

En la mayoría de los casos, la entrada de la tubería será una matriz de objetos. Aunque el comportamiento del bloque `PROCESS{}` puede parecer similar al bloque `foreach{}` , omitir un elemento de la matriz requiere un proceso diferente.

Si, como en `foreach{}` , usó `continue` dentro del bloque `PROCESS{}` , se rompería la tubería, omitiendo todas las siguientes declaraciones, incluido el bloque `END{}` . En su lugar, use `return` : solo finalizará el bloque `PROCESS{}` para el elemento actual y se moverá al siguiente.

En algunos casos, es necesario generar el resultado de funciones con codificación diferente. La codificación de la salida de los CmdLets está controlada por la variable `$OutputEncoding` . Cuando se pretende que la salida se coloque en una tubería hacia aplicaciones nativas, podría ser una buena idea corregir la codificación para que coincida con el destino `$OutputEncoding = [Console]::OutputEncoding`

Referencias adicionales:

Artículo del blog con más información sobre `$OutputEncoding`
https://blogs.msdn.microsoft.com/powershell/2006/12/11/outputencoding-to-the-rescue/

Examples

Funciones de escritura con ciclo de vida avanzado

Este ejemplo muestra cómo una función puede aceptar entradas canalizadas e iterar de manera eficiente.

Tenga en cuenta que las estructuras de `begin` y `end` de la función son opcionales cuando se canalizan, pero ese `process` es necesario cuando se utiliza `ValueFromPipeline` o `ValueFromPipelineByPropertyName` .

```
function Write-FromPipeline{
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline)]
        $myInput
    )
    begin {
        Write-Verbose -Message "Beginning Write-FromPipeline"
    }
    process {
        Write-Output -InputObject $myInput
    }
    end {
        Write-Verbose -Message "Ending Write-FromPipeline"
    }
}

$foo = 'hello','world',1,2,3

$foo | Write-FromPipeline -Verbose
```

Salida:

```
VERBOSE: Beginning Write-FromPipeline
hello
world
1
2
3
VERBOSE: Ending Write-FromPipeline
```

Soporte básico de oleoducto en funciones

Este es un ejemplo de una función con el soporte más simple posible para la canalización. Cualquier función con soporte de canalización debe tener al menos un parámetro con el parámetro ParameterAttribute `ValueFromPipeline` o `ValueFromPipelineByPropertyName` , como se muestra a continuación.

```
function Write-FromPipeline {
    param(
        [Parameter(ValueFromPipeline)]  # This sets the ParameterAttribute
        [String]$Input
    )
    Write-Host $Input
}
```

```
$foo = 'Hello World!'

$foo | Write-FromPipeline
```

Salida:

```
Hello World!
```

Nota: en PowerShell 3.0 y versiones posteriores, se admiten los valores predeterminados para ParameterAttributes. En versiones anteriores, debe especificar `ValueFromPipeline=$true` .

Concepto de trabajo de la tubería

En una serie de tuberías, cada función corre paralela a las demás, como hilos paralelos. El primer objeto procesado se transmite a la siguiente canalización y el siguiente procesamiento se ejecuta inmediatamente en otro hilo. Esto explica la ganancia de alta velocidad en comparación con el estándar `ForEach`

```
@( bigFile_1, bigFile_2, ..., bigFile_n) | Copy-File | Encrypt-File | Get-Md5
```

1. paso - copia el primer archivo (en `Copy-file` subproceso del `Copy-file`)
2. paso: copiar el segundo archivo (en `Copy-file` subproceso del `Copy-file`) y, simultáneamente, cifrar el primero (en `Encrypt-File`)
3. paso: copiar el tercer archivo (en `Copy-file` subproceso del `Copy-file`) y, a la vez, cifrar el segundo archivo (en el archivo de `Encrypt-File`) y al mismo tiempo `get-Md5` del primero (en `Get-Md5`)

Lea Trabajando con la tubería de PowerShell en línea:
https://riptutorial.com/es/powershell/topic/3937/trabajando-con-la-tuberia-de-powershell

Capítulo 63: Trabajando con objetos

Examples

Actualizando objetos

Añadiendo propiedades

Si desea agregar propiedades a un objeto existente, puede usar el cmdlet Add-Member. Con PSObjects, los valores se mantienen en un tipo de "Propiedades de nota"

```
$object = New-Object -TypeName PSObject -Property @{
        Name = $env:username
        ID = 12
        Address = $null
    }

Add-Member -InputObject $object -Name "SomeNewProp" -Value "A value" -MemberType NoteProperty

# Returns
PS> $Object
Name ID Address SomeNewProp
---- -- ------- -----------
nem  12         A value
```

También puede agregar propiedades con el Cmdlet Select-Object (denominados propiedades calculadas):

```
$newObject = $Object | Select-Object *, @{label='SomeOtherProp'; expression={'Another value'}}

# Returns
PS> $newObject
Name ID Address SomeNewProp SomeOtherProp
---- -- ------- ----------- -------------
nem  12         A value     Another value
```

El comando anterior se puede acortar a esto:

```
$newObject = $Object | Select *,@{l='SomeOtherProp';e={'Another value'}}
```

Eliminando propiedades

Puede usar el Cmdlet Seleccionar objeto para eliminar propiedades de un objeto:

```
$object = $newObject | Select-Object * -ExcludeProperty ID, Address

# Returns
```

```
PS> $object
Name SomeNewProp SomeOtherProp
---- ----------- -------------
nem  A value     Another value
```

Creando un nuevo objeto

PowerShell, a diferencia de otros lenguajes de scripting, envía objetos a través de la tubería. Lo que esto significa es que cuando envía datos de un comando a otro, es esencial poder crear, modificar y recopilar objetos.

Crear un objeto es simple. La mayoría de los objetos que cree serán objetos personalizados en PowerShell, y el tipo a usar para eso es PSObject. PowerShell también te permitirá crear cualquier objeto que puedas crear en .NET.

Aquí hay un ejemplo de cómo crear nuevos objetos con algunas propiedades:

Opción 1: Nuevo objeto

```
$newObject = New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Puede almacenar el objeto en una variable precediendo el comando con `$newObject =`

También puede ser necesario almacenar colecciones de objetos. Esto se puede hacer creando una variable de colección vacía y agregando objetos a la colección, así:

```
$newCollection = @()
$newCollection += New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}
```

Es posible que desee recorrer esta colección objeto por objeto. Para ello, localice la sección Loop en la documentación.

Opción 2: Seleccionar objeto

Una forma menos común de crear objetos que todavía encontrará en Internet es la siguiente:

```
$newObject = 'unuseddummy' | Select-Object -Property Name, ID, Address
$newObject.Name = $env:username
$newObject.ID = 12

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Opción 3: acelerador de tipo pscustomobject (se requiere PSv3 +)

El acelerador de tipo ordenado obliga a PowerShell a mantener nuestras propiedades en el orden en que las definimos. No necesita el acelerador de tipos ordenado para usar `[PSCustomObject]` :

```
$newObject = [PSCustomObject][Ordered]@{
    Name = $env:Username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -------
nem  12
```

Examinando un objeto

Ahora que tiene un objeto, podría ser bueno averiguar qué es. Puede usar el cmdlet Get-Member para ver qué es un objeto y qué contiene:

```
Get-Item c:\windows | Get-Member
```

Esto produce:

```
TypeName: System.IO.DirectoryInfo
```

Seguido de una lista de propiedades y métodos que tiene el objeto.

Otra forma de obtener el tipo de un objeto es usar el método GetType, así:

```
C:\> $Object = Get-Item C:\Windows
C:\> $Object.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DirectoryInfo                            System.IO.FileSystemInfo
```

Para ver una lista de propiedades que tiene el objeto, junto con sus valores, puede usar el cmdlet Format-List con su parámetro de propiedad establecido en: * (es decir, todo).

Aquí hay un ejemplo, con el resultado resultante:

```
C:\> Get-Item C:\Windows | Format-List -Property *


PSPath            : Microsoft.PowerShell.Core\FileSystem::C:\Windows
PSParentPath      : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName       : Windows
PSDrive           : C
PSProvider        : Microsoft.PowerShell.Core\FileSystem
PSIsContainer     : True
Mode              : d-----
BaseName          : Windows
Target            : {}
LinkType          :
Name              : Windows
Parent            :
Exists            : True
Root              : C:\
FullName          : C:\Windows
Extension         :
CreationTime      : 30/10/2015 06:28:30
CreationTimeUtc   : 30/10/2015 06:28:30
LastAccessTime    : 16/08/2016 17:32:04
LastAccessTimeUtc : 16/08/2016 16:32:04
LastWriteTime     : 16/08/2016 17:32:04
LastWriteTimeUtc  : 16/08/2016 16:32:04
Attributes        : Directory
```

Creación de instancias de clases genéricas

Nota: ejemplos escritos para PowerShell 5.1. Puede crear instancias de clases genéricas.

```
#Nullable System.DateTime
[Nullable[datetime]]$nullableDate = Get-Date -Year 2012
$nullableDate
$nullableDate.GetType().FullName
$nullableDate = $null
$nullableDate

#Normal System.DateTime
[datetime]$aDate = Get-Date -Year 2013
$aDate
$aDate.GetType().FullName
$aDate = $null #Throws exception when PowerShell attempts to convert null to
```

Da la salida:

```
Saturday, 4 August 2012 08:53:02
System.DateTime
Sunday, 4 August 2013 08:53:02
System.DateTime
Cannot convert null to type "System.DateTime".
```

```
At line:14 char:1
+ $aDate = $null
+ ~~~~~~~~~~~~~~
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

Colecciones genéricas también son posibles

```
[System.Collections.Generic.SortedDictionary[int, String]]$dict =
[System.Collections.Generic.SortedDictionary[int, String]]::new()
$dict.GetType().FullName

$dict.Add(1, 'a')
$dict.Add(2, 'b')
$dict.Add(3, 'c')


$dict.Add('4', 'd') #powershell auto converts '4' to 4
$dict.Add('5.1', 'c') #powershell auto converts '5.1' to 5

$dict

$dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so it throws an error
```

Da la salida:

```
System.Collections.Generic.SortedDictionary`2[[System.Int32, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]

Key Value
--- -----
  1 a
  2 b
  3 c
  4 d
  5 c
Cannot convert argument "key", with value: "z", for "Add" to type "System.Int32": "Cannot
convert value "z" to type "System.Int32". Error: "Input string was not in a correct format.""
At line:15 char:1
+ $dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so  ...
+ ~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodException
    + FullyQualifiedErrorId : MethodArgumentConversionInvalidCastArgument
```

Lea Trabajando con objetos en línea: https://riptutorial.com/es/powershell/topic/1328/trabajando-con-objetos

Capítulo 64: Trabajos de fondo de PowerShell

Introducción

Los trabajos se introdujeron en PowerShell 2.0 y ayudaron a resolver un problema inherente en las herramientas de línea de comandos. En pocas palabras, si inicia una tarea de ejecución prolongada, su indicador no estará disponible hasta que la tarea finalice. Como ejemplo de una tarea de larga ejecución, piense en este simple comando de PowerShell:

Get-ChildItem -Path c: \ -Recurse

Tomará un tiempo recuperar la lista completa de directorios de su unidad C :. Si lo ejecuta como trabajo, la consola recuperará el control y podrá capturar el resultado más adelante.

Observaciones

PowerShell Jobs se ejecuta en un nuevo proceso. Esto tiene pros y contras que están relacionados.

Pros:

1. El trabajo se ejecuta en un proceso limpio, incluido el entorno.
2. El trabajo puede ejecutarse de forma asíncrona a su proceso principal de PowerShell

Contras:

1. Los cambios en el entorno del proceso no estarán presentes en el trabajo.
2. Los parámetros que pasan y los resultados devueltos se serializan.
    - Esto significa que si cambia un objeto de parámetro mientras el trabajo se está ejecutando, no se reflejará en el trabajo.
    - Esto también significa que si un objeto no puede ser serializado, no puede pasarlo o devolverlo (aunque PowerShell puede copiar cualquier parámetro y pasar / devolver un PSObject).

Examples

Creación de empleo básico

Iniciar un bloque de script como trabajo de fondo:

```
$job = Start-Job -ScriptBlock {Get-Process}
```

Iniciar una secuencia de comandos como trabajo de fondo:

```
$job = Start-Job -FilePath "C:\YourFolder\Script.ps1"
```

Comience un trabajo utilizando `Invoke-Command` en una máquina remota:

```
$job = Invoke-Command -ComputerName "ComputerName" -ScriptBlock {Get-Service winrm} -JobName
"WinRM" -ThrottleLimit 16 -AsJob
```

Iniciar trabajo como un usuario diferente (Solicita contraseña):

```
Start-Job -ScriptBlock {Get-Process} -Credential "Domain\Username"
```

O

```
Start-Job -ScriptBlock {Get-Process} -Credential (Get-Credential)
```

Iniciar trabajo como un usuario diferente (No se solicita):

```
$username = "Domain\Username"
$password = "password"
$secPassword = ConvertTo-SecureString -String $password -AsPlainText -Force
$credentials = New-Object System.Management.Automation.PSCredential -ArgumentList @($username,
$secPassword)
Start-Job -ScriptBlock {Get-Process} -Credential $credentials
```

Gestión de trabajos básicos

Obtenga una lista de todos los trabajos en la sesión actual:

```
Get-Job
```

Esperando un trabajo para terminar antes de obtener el resultado:

```
$job | Wait-job | Receive-Job
```

Tiempo de espera de un trabajo si se ejecuta demasiado tiempo (10 segundos en este ejemplo)

```
$job | Wait-job -Timeout 10
```

Detener un trabajo (completa todas las tareas que están pendientes en esa cola de trabajos antes de finalizar):

```
$job | Stop-Job
```

Eliminar trabajo de la lista de trabajos en segundo plano de la sesión actual:

```
$job | Remove-Job
```

Nota : lo siguiente solo funcionará en trabajos de `Workflow` trabajo.

Suspender un `Workflow` trabajo (pausa):

```
$job | Suspend-Job
```

Reanudar un `Workflow` trabajo:

```
$job | Resume-Job
```

Lea Trabajos de fondo de PowerShell en línea:
https://riptutorial.com/es/powershell/topic/3970/trabajos-de-fondo-de-powershell

Capítulo 65: Usando clases estáticas existentes

Introducción

Estas clases son bibliotecas de referencia de métodos y propiedades que no cambian de estado, en una palabra, inmutables. No necesitas crearlos, simplemente los usas. Las clases y los métodos como estos se llaman clases estáticas porque no se crean, destruyen o cambian. Puede referirse a una clase estática rodeando el nombre de la clase entre corchetes.

Examples

Creando nuevo GUID al instante

Use las clases .NET existentes al instante con PowerShell usando [class] :: Method (args):

```
PS C:\> [guid]::NewGuid()

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

De manera similar, en PowerShell 5+ puede usar el cmdlet `New-Guid` :

```
PS C:\> New-Guid

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Para obtener el GUID solo como una `[String]` , haga referencia a la propiedad `.Guid` :

```
[guid]::NewGuid().Guid
```

Usando la clase de matemática .Net

Puede usar la clase .Net Math para hacer cálculos ([System.Math])

Si quieres saber qué métodos están disponibles puedes usar:

```
[System.Math] | Get-Member -Static -MemberType Methods
```

Aquí hay algunos ejemplos de cómo usar la clase de matemáticas:

```
PS C:\> [System.Math]::Floor(9.42)
```

```
9
PS C:\> [System.Math]::Ceiling(9.42)
10
PS C:\> [System.Math]::Pow(4,3)
64
PS C:\> [System.Math]::Sqrt(49)
7
```

Sumando tipos

Por nombre de la asamblea, agregar biblioteca

```
Add-Type -AssemblyName "System.Math"
```

o por la ruta del archivo:

```
Add-Type -Path "D:\Libs\CustomMath.dll"
```

Para usar el tipo agregado:

```
[CustomMath.NameSpace]::Method(param1, $variableParam, [int]castMeAsIntParam)
```

Lea Usando clases estáticas existentes en línea:
https://riptutorial.com/es/powershell/topic/1522/usando-clases-estaticas-existentes

Capítulo 66: Usando la barra de progreso

Introducción

Se puede usar una barra de progreso para mostrar que algo se encuentra en un proceso. Es una característica que le permite ahorrar tiempo y es elegante. Las barras de progreso son increíblemente útiles para depurar qué parte de la secuencia de comandos se está ejecutando, y son satisfactorias para las personas que ejecutan secuencias de comandos para realizar un seguimiento de lo que está sucediendo. Es común mostrar algún tipo de progreso cuando una secuencia de comandos tarda mucho tiempo en completarse. Cuando un usuario inicia la secuencia de comandos y no sucede nada, uno comienza a preguntarse si la secuencia de comandos se inició correctamente.

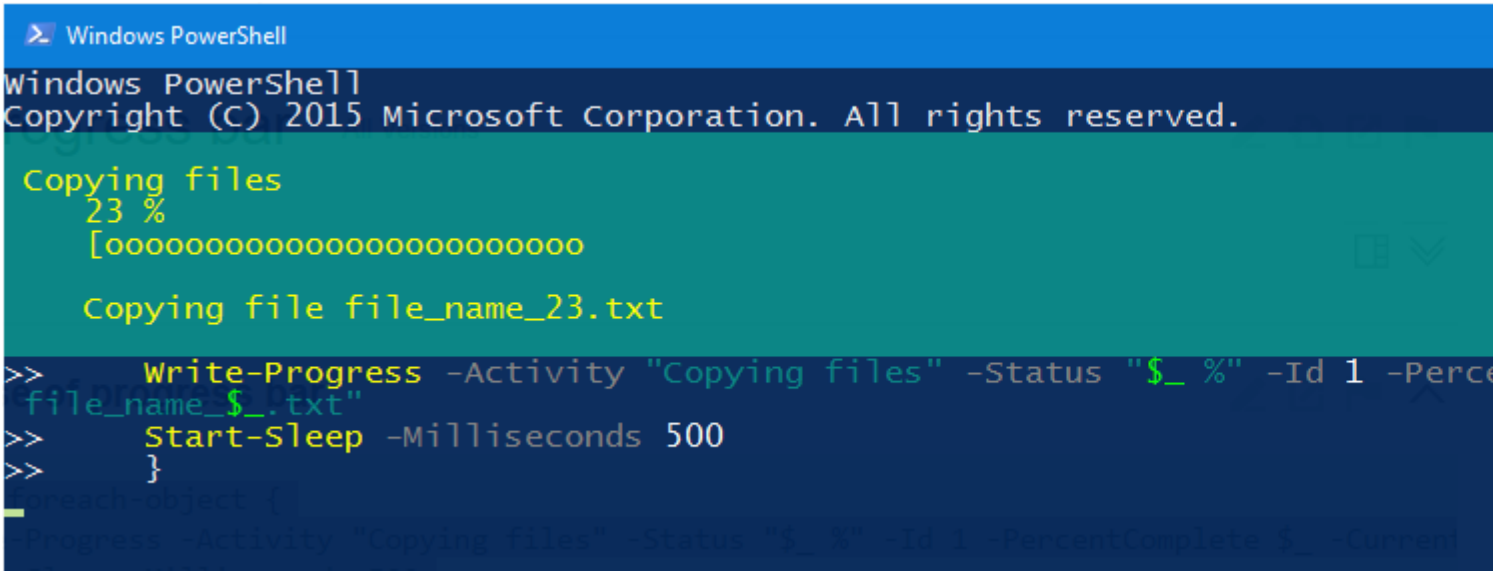
Examples

Uso simple de la barra de progreso.

```
1..100 | ForEach-Object {
        Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -PercentComplete $_ -
CurrentOperation "Copying file file_name_$_.txt"
        Start-Sleep -Milliseconds 500    # sleep simulates working code, replace this line
with your executive code (i.e. file copying)
    }
```

Tenga en cuenta que, por brevedad, este ejemplo no contiene ningún código ejecutivo (simulado con `Start-Sleep`). Sin embargo, es posible ejecutarlo directamente como está y luego modificarlo y jugar con él.

Así es como se ve el resultado en la consola PS:



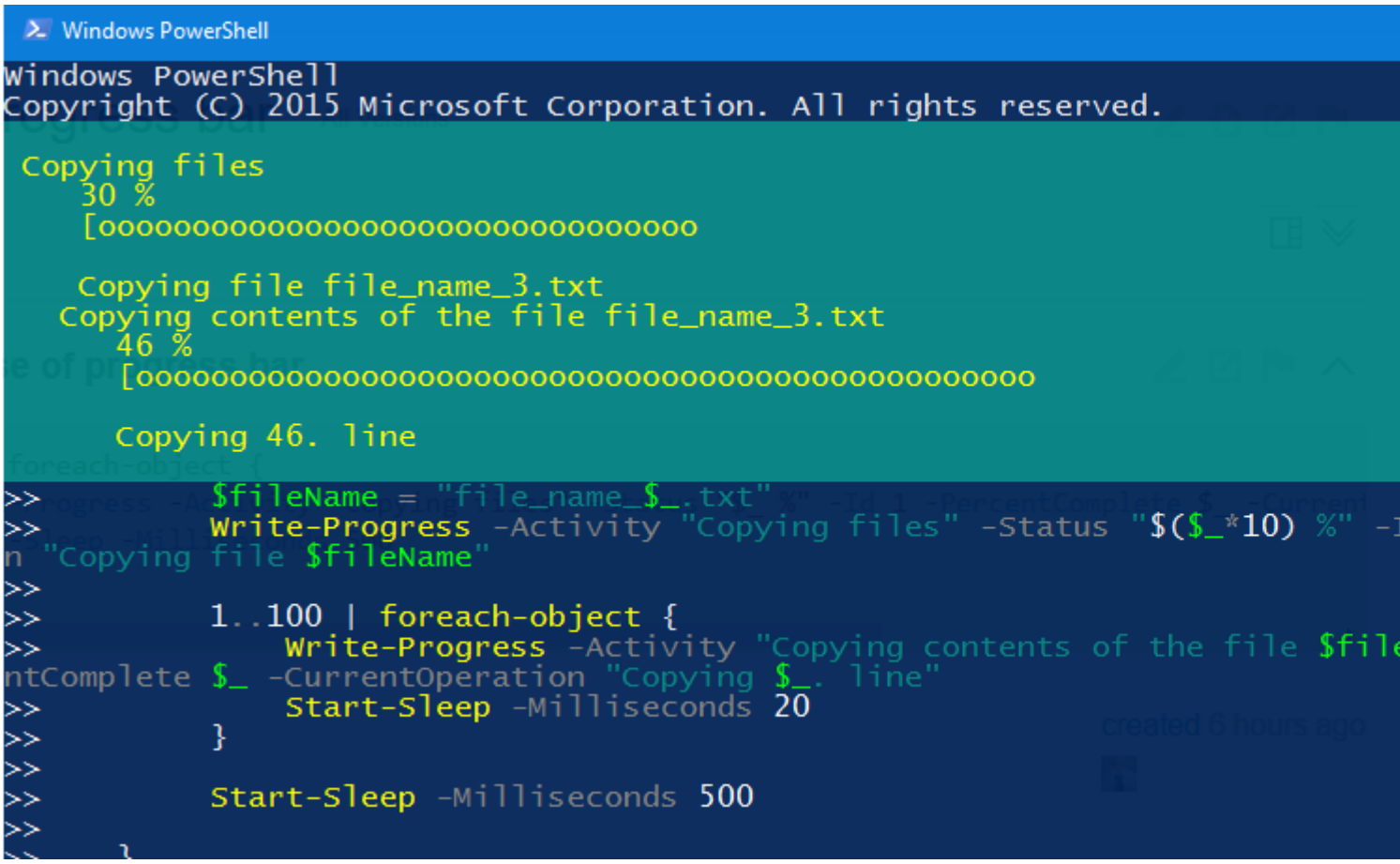
Así es como se ve el resultado en PS ISE:

Uso de la barra de progreso interior

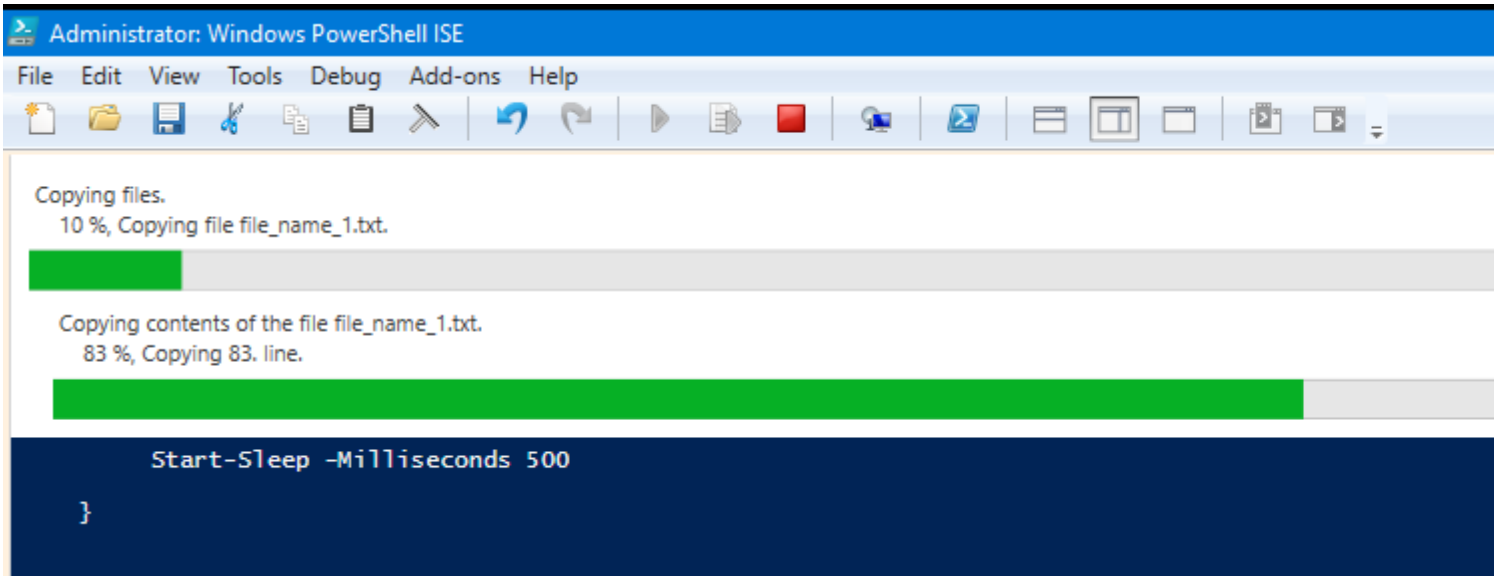
```
1..10 | foreach-object {
        $fileName = "file_name_$_.txt"
        Write-Progress -Activity "Copying files" -Status "$($_*10) %" -Id 1 -PercentComplete
($_*10) -CurrentOperation "Copying file $fileName"

        1..100 | foreach-object {
            Write-Progress -Activity "Copying contents of the file $fileName" -Status "$_ %" -
Id 2 -ParentId 1 -PercentComplete $_ -CurrentOperation "Copying $_. line"

            Start-Sleep -Milliseconds 20 # sleep simulates working code, replace this line
with your executive code (i.e. file copying)
        }

        Start-Sleep -Milliseconds 500 # sleep simulates working code, replace this line with
your executive code (i.e. file search)

   }
```

Tenga en cuenta que, por brevedad, este ejemplo no contiene ningún código ejecutivo (simulado con `Start-Sleep`). Sin embargo, es posible ejecutarlo directamente como está y luego modificarlo y jugar con él.

Así es como se ve el resultado en la consola PS:



Así es como se ve el resultado en PS ISE:



Lea Usando la barra de progreso en línea: https://riptutorial.com/es/powershell/topic/5020/usando-la-barra-de-progreso

Capítulo 67: Usando ShouldProcess

Sintaxis

- $ PSCmdlet.ShouldProcess ("Target")
- $ PSCmdlet.ShouldProcess ("Target", "Action")

Parámetros

Parámetro	Detalles
Objetivo	El recurso está siendo cambiado.
Acción	La operación que se está realizando. De forma predeterminada, el nombre del cmdlet.

Observaciones

`$PSCmdlet.ShouldProcess()` también escribirá automáticamente un mensaje en la salida detallada.

```
PS> Invoke-MyCmdlet -Verbose
VERBOSE: Performing the operation "Invoke-MyCmdlet" on target "Target of action"
```

Examples

Agregando soporte de -WhatIf y -Confirm a su cmdlet

```
function Invoke-MyCmdlet {
    [CmdletBinding(SupportsShouldProcess = $true)]
    param()
    # ...
}
```

Usando ShouldProcess () con un argumento

```
if ($PSCmdlet.ShouldProcess("Target of action")) {
    # Do the thing
}
```

Al usar `-WhatIf` :

```
What if: Performing the action "Invoke-MyCmdlet" on target "Target of action"
```

Al usar `-Confirm` :

```
Are you sure you want to perform this action?
Performing operation "Invoke-MyCmdlet" on target "Target of action"
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help (default is "Y"):
```

Ejemplo de uso completo

Otros ejemplos no pudieron explicarme claramente cómo activar la lógica condicional.

Este ejemplo también muestra que los comandos subyacentes también escucharán la marca -Confirmar!

```
<#
Restart-Win32Computer
#>

function Restart-Win32Computer
{
    [CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact="High")]
    param (
    [parameter(Mandatory=$true,ValueFromPipeline=$true,ValueFromPipelineByPropertyName=$true)]
    [string[]]$computerName,
    [parameter(Mandatory=$true)]
    [string][ValidateSet("Restart","LogOff","Shutdown","PowerOff")] $action,
    [boolean]$force = $false
)
BEGIN {
# translate action to numeric value required by the method
switch($action) {
    "Restart"
    {
        $_action = 2
        break
    }
    "LogOff"
    {
        $_action = 0
        break
    }
    "Shutdown"
    {
        $_action = 2
        break
    }
    "PowerOff"
    {
        $_action = 8
        break
    }
}
# to force, add 4 to the value
if($force)
{
    $_action += 4
}
write-verbose "Action set to $action"
}
PROCESS {
    write-verbose "Attempting to connect to $computername"
```

```
    # this is how we support -whatif and -confirm
    # which are enabled by the SupportsShouldProcess
    # parameter in the cmdlet bindnig
    if($pscmdlet.ShouldProcess($computername)) {
        get-wmiobject win32_operatingsystem -computername $computername | invoke-wmimethod -
name Win32Shutdown -argumentlist $_action
    }
}
}
#Usage:
#This will only output a description of the actions that this command would execute if -WhatIf
is removed.
'localhost','server1'| Restart-Win32Computer -action LogOff -whatif

#This will request the permission of the caller to continue with this item.
#Attention: in this example you will get two confirmation request because all cmdlets called
by this cmdlet that also support ShouldProcess, will ask for their own confirmations...
'localhost','server1'| Restart-Win32Computer -action LogOff -Confirm
```

Lea Usando ShouldProcess en línea: https://riptutorial.com/es/powershell/topic/1145/usando-shouldprocess

Capítulo 68: Uso del sistema de ayuda

Observaciones

`Get-Help` es un cmdlet para leer temas de ayuda en PowerShell.

Leer más en TechNet

Examples

Actualización del sistema de ayuda

3.0

A partir de PowerShell 3.0, puede descargar y actualizar la documentación de ayuda sin conexión con un solo cmdlet.

```
Update-Help
```

Para actualizar la ayuda en varias computadoras (o computadoras no conectadas a internet).

Ejecuta lo siguiente en una computadora con los archivos de ayuda

```
Save-Help -DestinationPath \\Server01\Share\PSHelp -Credential $Cred
```

Para ejecutar en muchas computadoras de forma remota

```
Invoke-Command -ComputerName (Get-Content Servers.txt) -ScriptBlock {Update-Help -SourcePath
\\Server01\Share\Help -Credential $cred}
```

Usando Get-Help

`Get-Help` puede usar `Get-Help` para ver la ayuda en PowerShell. Puede buscar cmdlets, funciones, proveedores u otros temas.

Para ver la documentación de ayuda sobre los trabajos, use:

```
Get-Help about_Jobs
```

Puedes buscar temas usando comodines. Si desea enumerar los temas de ayuda disponibles con un título que comienza con `about_` , intente:

```
Get-Help about_*
```

Si quisieras ayuda en `Select-Object` , usarías:

```
Get-Help Select-Object
```

También puedes usar los alias `help` o `man` .

Ver la versión en línea de un tema de ayuda

Puede acceder a la documentación de ayuda en línea utilizando:

```
Get-Help Get-Command -Online
```

Ejemplos de visualización

Mostrar ejemplos de uso para un cmdlet específico.

```
Get-Help Get-Command -Examples
```

Viendo la página de ayuda completa

Ver la documentación completa para el tema.

```
Get-Help Get-Command -Full
```

Ver ayuda para un parámetro específico

Puede ver la ayuda para un parámetro específico usando:

```
Get-Help Get-Content -Parameter Path
```

Lea Uso del sistema de ayuda en línea: https://riptutorial.com/es/powershell/topic/5644/uso-del-sistema-de-ayuda

Capítulo 69: Variables automáticas

Introducción

Las variables automáticas son creadas y mantenidas por Windows PowerShell. Uno tiene la capacidad de llamar a una variable casi cualquier nombre en el libro; Las únicas excepciones a esto son las variables que ya están siendo administradas por PowerShell. Estas variables, sin duda, serán los objetos más repetitivos que utilice en PowerShell junto a las funciones (como **$?** - indica el estado de éxito / falla de la última operación)

Sintaxis

- `$$` - Contiene el último token en la última línea recibida por la sesión.
- `$^` : Contiene el primer token en la última línea recibida por la sesión.
- `$?` - Contiene el estado de ejecución de la última operación.
- `$_` - Contiene el objeto actual en la tubería

Examples

$ pid

Contiene el ID de proceso del proceso de alojamiento actual.

```
PS C:\> $pid
26080
```

Valores booleanos

`$true` y `$false` son dos variables que representan lógicas VERDADERO y FALSO.

Tenga en cuenta que debe especificar el signo de dólar como primer carácter (que es diferente de C #).

```
$boolExpr = "abc".Length -eq 3 # length of "abc" is 3, hence $boolExpr will be True
if($boolExpr -eq $true){
    "Length is 3"
}
# result will be "Length is 3"
$boolExpr -ne $true
#result will be False
```

Tenga en cuenta que cuando utiliza booleano verdadero / falso en su código, escribe `$true` o `$false` , pero cuando Powershell devuelve un valor booleano, parece `True` o `False`

$ nulo

`$null` se utiliza para representar el valor ausente o indefinido.
`$null` se puede usar como un marcador de posición vacío para el valor vacío en las matrices:

```
PS C:\> $array = 1, "string", $null
PS C:\> $array.Count
3
```

Cuando usamos la misma matriz que la fuente para `ForEach-Object` , procesará los tres elementos (incluido $ null):

```
PS C:\> $array | ForEach-Object {"Hello"}
Hello
Hello
Hello
```

¡Ten cuidado! Esto significa que `ForEach-Object` **procesará** incluso `$null` por sí mismo:

```
PS C:\> $null | ForEach-Object {"Hello"} # THIS WILL DO ONE ITERATION !!!
Hello
```

Lo que es un resultado muy inesperado si lo comparas con el bucle `foreach` clásico:

```
PS C:\> foreach($i in $null) {"Hello"} # THIS WILL DO NO ITERATION
PS C:\>
```

$ OFS

La variable llamada Output Field Separator contiene un valor de cadena que se utiliza al convertir una matriz en una cadena. Por defecto `$OFS = " "` (*un espacio*), pero se puede cambiar:

```
PS C:\> $array = 1,2,3
PS C:\> "$array" # default OFS will be used
1 2 3
PS C:\> $OFS = ",." # we change OFS to comma and dot
PS C:\> "$array"
1,.2,.3
```

$ _ / $ PSItem

Contiene el objeto / elemento que está siendo procesado actualmente por la canalización.

```
PS C:\> 1..5 | % { Write-Host "The current item is $_" }
The current item is 1
The current item is 2
The current item is 3
The current item is 4
The current item is 5
```

`$PSItem` y `$_` son idénticos y se pueden usar indistintamente, pero `$_` es, con mucho, el más utilizado.

PS

Contiene el estado de la última operación. Cuando no hay error, se establece en `True` :

```
PS C:\> Write-Host "Hello"
Hello
PS C:\> $?
True
```

Si hay algún error, se establece en `False` :

```
PS C:\> wrt-host
wrt-host : The term 'wrt-host' is not recognized as the name of a cmdlet, function, script
file, or operable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and
try again.
At line:1 char:1
+ wrt-host
+ ~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (wrt-host:String) [], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException

PS C:\> $?
False
```

$ error

Matriz de los objetos de error más recientes. El primero en la matriz es el más reciente:

```
PS C:\> throw "Error" # resulting output will be in red font
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
    + FullyQualifiedErrorId : Error

PS C:\> $error[0] # resulting output will be normal string (not red    )
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~~~~~~~~~
    + CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
    + FullyQualifiedErrorId : Error
```

Consejos de uso: cuando use la variable `$error` en un cmdlet de formato (por ejemplo, format-list), tenga en cuenta que use el interruptor `-Force` . De lo contrario, el cmdlet de formato generará el contenido de `$error` en la forma mostrada anteriormente.

Las entradas de error pueden eliminarse, por ejemplo, `$Error.Remove($Error[0])` .

Lea Variables automáticas en línea: https://riptutorial.com/es/powershell/topic/5353/variables-automaticas

Capítulo 70: Variables automáticas - parte 2

Introducción

El tema "Variables automáticas" ya tiene 7 ejemplos enumerados y no podemos agregar más. Este tema tendrá una continuación de Variables automáticas.

Las variables automáticas son variables que almacenan información de estado para PowerShell. Estas variables son creadas y mantenidas por Windows PowerShell.

Observaciones

No estoy seguro de si esta es la mejor manera de manejar la documentación de variables automáticas, pero esto es mejor que nada. Por favor comenta si encuentras una manera mejor :)

Examples

$ PSVersionTable

Contiene una tabla hash de solo lectura (Constante, AllScope) que muestra detalles sobre la versión de PowerShell que se está ejecutando en la sesión actual.

```
$PSVersionTable    #this call results in this:
Name                           Value
----                           -----
PSVersion                      5.0.10586.117
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.10586.117
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

La forma más rápida de ejecutar una versión de PowerShell:

```
$PSVersionTable.PSVersion
# result :
Major  Minor  Build  Revision
-----  -----  -----  --------
5      0      10586  117
```

Lea Variables automáticas - parte 2 en línea:
https://riptutorial.com/es/powershell/topic/8639/variables-automaticas---parte-2

Capítulo 71: Variables de entorno

Examples

Las variables de entorno de Windows son visibles como una unidad PS llamada Env:

Puedes ver la lista con todas las variables de entorno con:
Get-Childitem env:

Llamada instantánea de variables de entorno con $ env:

```
$env:COMPUTERNAME
```

Lea Variables de entorno en línea: https://riptutorial.com/es/powershell/topic/5635/variables-de-entorno

Capítulo 72: Variables en PowerShell

Introducción

Las variables se utilizan para almacenar valores. Deje que el valor sea de cualquier tipo, necesitamos almacenarlo en algún lugar para poder usarlo en toda la consola / script. Los nombres de variables en PowerShell comienzan con **$** , como en *$ Variable1* , y los valores se asignan usando **=** , como **$ Variable1 = "Valor 1".** PowerShell admite una gran cantidad de tipos de variables; como cadenas de texto, enteros, decimales, matrices e incluso tipos avanzados como números de versión o direcciones IP.

Examples

Variable simple

Todas las variables en powershell comienzan con un signo de dólar estadounidense (`$`). El ejemplo más simple de esto es:

```
$foo = "bar"
```

Esta declaración asigna una variable llamada `foo` con un valor de cadena de "barra".

Eliminando una variable

Para eliminar una variable de la memoria, se puede usar el cmdlet `Remove-Item` . Nota: El nombre de la variable NO incluye el `$` .

```
Remove-Item Variable:\foo
```

`Variable` tiene un proveedor para permitir que la mayoría de los cmdlets * -item funcionen de manera similar a los sistemas de archivos.

Otro método para eliminar la variable es usar el cmdlet Remove-Variable y su alias rv

```
$var = "Some Variable" #Define variable 'var' containing the string 'Some Variable'
$var #For test and show string 'Some Variable' on the console

Remove-Variable -Name var
$var

#also can use alias 'rv'
rv var
```

Alcance

El ámbito predeterminado para una variable es el contenedor adjunto. Si está fuera de un script u

otro contenedor, el alcance es `Global` . Para especificar un alcance , se le `$scope:varname` prefijo al nombre de variable `$scope:varname` así:

```
$foo = "Global Scope"
function myFunc {
    $foo = "Function (local) scope"
    Write-Host $global:foo
    Write-Host $local:foo
    Write-Host $foo
}
myFunc
Write-Host $local:foo
Write-Host $foo
```

Salida:

```
    Global Scope
    Function (local) scope
    Function (local) scope
    Global Scope
    Global Scope
```

Leyendo una salida de CmdLet

De forma predeterminada, powershell devolvería la salida a la entidad llamante. Considere el siguiente ejemplo,

```
Get-Process -Name excel
```

Esto simplemente devolvería el proceso en ejecución que coincide con el nombre excel, a la entidad llamante. En este caso, el host de PowerShell. Imprime algo como

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName

-------  ------    -----      ----- -----   ------     --  -- -----------

   1037      54    67632      62544   617     5.23   4544   1 EXCEL
```

Ahora, si asigna la salida a una variable, simplemente no imprimirá nada. Y, por supuesto, la variable mantiene la salida. (Ya sea una cadena, Objeto - Cualquier tipo para esa materia)

```
$allExcel = Get-Process -Name excel
```

Entonces, digamos que tiene un escenario donde desea asignar una variable por un nombre dinámico, puede usar el parámetro **`-OutVariable`**

```
Get-Process -Name excel -OutVariable AllRunningExcel
```

Tenga en cuenta que aquí falta el $. Una diferencia importante entre estas dos asignaciones es que, también imprime el resultado además de asignarlo a la variable AllRunningExcel. También

puede optar por asignarlo a otra variable.

```
$VarOne = Get-Process -Name excel -OutVariable VarTwo
```

Aunque, el escenario anterior es muy raro, ambas variables $ VarOne y $ VarTwo tendrán el mismo valor.

Ahora considera esto,

```
Get-Process -Name EXCEL -OutVariable MSOFFICE
Get-Process -Name WINWORD -OutVariable +MSOFFICE
```

La primera declaración simplemente obtendría el proceso de Excel y lo asignaría a la variable MSOFFICE, y luego ejecutaría los procesos de ms word y lo "agregaría" al valor existente de MSOFFICE. Se vería algo como esto,

```
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id  SI ProcessName

-------  ------    -----      ----- -----   ------     --  -- -----------

   1047      54    67720      64448   618     5.70   4544   1 EXCEL

   1172      70    50052      81780   584     1.83  14968   1 WINWORD
```

Asignación de listas de múltiples variables

Powershell permite la asignación múltiple de variables y trata casi todo como una matriz o lista. Esto significa que en lugar de hacer algo como esto:

```
$input = "foo.bar.baz"
$parts = $input.Split(".")
$foo = $parts[0]
$bar = $parts[1]
$baz = $parts[2]
```

Simplemente puede hacer esto:

```
$foo, $bar, $baz = $input.Split(".")
```

Dado que Powershell trata las asignaciones de esta manera como si fueran listas, si hay más valores en la lista que elementos en su lista de variables a las que asignarlos, la última variable se convierte en una matriz de los valores restantes. Esto significa que también puedes hacer cosas como esta:

```
$foo, $leftover = $input.Split(".") #Sets $foo = "foo", $leftover = ["bar","baz"]
$bar = $leftover[0] # $bar = "bar"
$baz = $leftover[1] # $baz = "baz"
```

Arrays

La declaración de arrays en Powershell es casi lo mismo que crear una instancia de cualquier otra variable, es decir, se usa una sintaxis de `$name =` . Los elementos de la matriz se declaran separándolos con comas (`,`):

```
$myArrayOfInts = 1,2,3,4
$myArrayOfStrings = "1","2","3","4"
```

Añadiendo a un arry

Agregar a una matriz es tan simple como usar el operador `+` :

```
$myArrayOfInts = $myArrayOfInts + 5
//now contains 1,2,3,4 & 5!
```

Combinando matrices juntas

De nuevo, esto es tan simple como usar el operador `+`

```
$myArrayOfInts = 1,2,3,4
$myOtherArrayOfInts = 5,6,7
$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts
//now 1,2,3,4,5,6,7
```

Lea Variables en PowerShell en línea: https://riptutorial.com/es/powershell/topic/3457/variables-en-powershell

Capítulo 73: Variables incorporadas

Introducción

PowerShell ofrece una variedad de variables "automáticas" (incorporadas) útiles. Algunas variables automáticas solo se completan en circunstancias especiales, mientras que otras están disponibles globalmente.

Examples

$ PSScriptRoot

```
Get-ChildItem -Path $PSScriptRoot
```

Este ejemplo recupera la lista de elementos secundarios (directorios y archivos) de la carpeta donde reside el archivo de script.

La variable automática `$PSScriptRoot` es `$null` si se usa desde fuera de un archivo de código de PowerShell. Si se utiliza *dentro de* un script de PowerShell, definió automáticamente la ruta del sistema de archivos completamente calificado al directorio que contiene el archivo de script.

En Windows PowerShell 2.0, esta variable solo es válida en módulos de script (.psm1). A partir de Windows PowerShell 3.0, es válido en todos los scripts.

$ Args

```
$Args
```

Contiene una matriz de parámetros no declarados y / o valores de parámetros que se pasan a una función, secuencia de comandos o bloque de secuencia de comandos. Cuando crea una función, puede declarar los parámetros usando la palabra clave param o agregando una lista de parámetros separados por comas entre paréntesis después del nombre de la función.

En una acción de evento, la variable $ Args contiene objetos que representan los argumentos de evento del evento que se está procesando. Esta variable se llena solo dentro del bloque de acción de un comando de registro de eventos. El valor de esta variable también se puede encontrar en la propiedad SourceArgs del objeto PSEventArgs (System.Management.Automation.PSEventArgs) que devuelve Get-Event.

$ PSItem

```
Get-Process | ForEach-Object -Process {
  $PSItem.Name
}
```

Igual que `$_` . Contiene el objeto actual en el objeto de canalización. Puede utilizar esta variable en comandos que realizan una acción en cada objeto o en objetos seleccionados en una tubería.

PS

```
Get-Process -Name doesnotexist
Write-Host -Object "Was the last operation successful? $?"
```

Contiene el estado de ejecución de la última operación. Contiene VERDADERO si la última operación tuvo éxito y FALSO si falló.

$ error

```
Get-Process -Name doesnotexist
Write-Host -Object ('The last error that occurred was: {0}' -f $error[0].Exception.Message)
```

Contiene una matriz de objetos de error que representan los errores más recientes. El error más reciente es el primer objeto de error en la matriz ($ Error [0]).

Para evitar que un error se agregue a la matriz $ Error, use el parámetro común ErrorAction con un valor de Ignorar. Para obtener más información, consulte about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216) .

Lea Variables incorporadas en línea: https://riptutorial.com/es/powershell/topic/8732/variables-incorporadas

Capítulo 74: WMI y CIM

Observaciones

CIM vs WMI

A partir de PowerShell 3.0, hay dos formas de trabajar con clases de administración en PowerShell, WMI y CIM. PowerShell 1.0 y 2.0 solo admitían el módulo WMI, que ahora está superpuesto al nuevo y mejorado módulo CIM. En una versión posterior de PowerShell, se eliminarán los cmdlets de WMI.

Comparación de módulos CIM y WMI:

Cmdlet CIM	WMI-cmdlet	Que hace
Get-CimInstance	Get-WmiObject	Obtiene objetos CIM / WMI para una clase
Invocar-Método Cim	Invocar-WmiMethod	Invoca un método de clase CIM / WMI
Registro-CimIndicationEvent	Registrarse-WmiEvent	Registra un evento para una clase CIM / WMI
Eliminar CimInstance	Remove-WmiObject	Eliminar objeto CIM / WMI
Set-CimInstance	Set-WmiInstance	Actualiza / Guarda objeto CIM / WMI
Get-CimAssociatedInstance	N / A	Obtener instancias asociadas (objeto / clases vinculadas)
Get-CimClass	Get-WmiObject -List	Lista de clases CIM / WMI
New-CimInstance	N / A	Crear nuevo objeto CIM
Get-CimSession	N / A	Listas de sesiones CIM
Nueva-cimSession	N / A	Crear nueva sesión CIM
New-CimSessionOption	N / A	Crea objeto con opciones de sesión; protocolo, codificación, deshabilitar el cifrado, etc. (para uso con `New-CimSession`)
Eliminar CimSession	N / A	Elimina / detiene sesión CIM

Recursos adicionales

¿Debo usar CIM o WMI con Windows PowerShell? @ ¡Oye, chico del scripting! Blog

Examples

Consulta de objetos

CIM / WMI se usa más comúnmente para consultar información o configuración en un dispositivo. En una clase que representa una configuración, proceso, usuario, etc. En PowerShell hay varias formas de acceder a estas clases e instancias, pero las formas más comunes son mediante el uso de los `Get-CimInstance` (CIM) o `Get-WmiObject` (WMI).

Listar todos los objetos para la clase CIM

Puedes enumerar todas las instancias de una clase.

3.0

CIM:

```
> Get-CimInstance -ClassName Win32_Process

ProcessId Name                         HandleCount WorkingSetSize VirtualSize
--------- ----                         ----------- -------------- -----------
0         System Idle Process          0           4096           65536
4         System                       1459        32768          3563520
480       Secure System                0           3731456        0
484       smss.exe                     52          372736         2199029891072
....
....
```

WMI:

```
Get-WmiObject -Class Win32_Process
```

Usando un filtro

Puede aplicar un filtro para obtener solo instancias específicas de una clase CIM / WMI. Los filtros se escriben utilizando `WQL` (predeterminado) o CQL (agregar `-QueryDialect CQL`). `-Filter` utiliza la parte `WHERE` de una consulta WQL / CQL completa.

3.0

CIM:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name = 'powershell.exe'"

ProcessId Name           HandleCount WorkingSetSize VirtualSize
--------- ----           ----------- -------------- -----------
4800      powershell.exe 676         88305664       2199697199104
```

WMI:

```
Get-WmiObject -Class Win32_Process -Filter "Name = 'powershell.exe'"

...
Caption                   : powershell.exe
CommandLine               : "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
CreationClassName         : Win32_Process
CreationDate              : 20160913184324.393887+120
CSCreationClassName       : Win32_ComputerSystem
CSName                    : STACKOVERFLOW-PC
Description               : powershell.exe
ExecutablePath            : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
ExecutionState            :
Handle                    : 4800
HandleCount               : 673
....
```

Usando una consulta WQL:

También puede usar una consulta WQL / CQL para consultar y filtrar instancias.

3.0

CIM:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE Name = 'powershell.exe'"

ProcessId Name           HandleCount WorkingSetSize VirtualSize
--------- ----           ----------- -------------- -----------
4800      powershell.exe 673         88387584       2199696674816
```

Consultando objetos en un espacio de nombres diferente:

3.0

CIM:

```
> Get-CimInstance -Namespace "root/SecurityCenter2" -ClassName AntiVirusProduct


displayName               : Windows Defender
instanceGuid              : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe    : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe  : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState              : 397568
timestamp                 : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName            :
```

WMI:

```
> Get-WmiObject -Namespace "root\SecurityCenter2" -Class AntiVirusProduct

__GENUS                   : 2
__CLASS                   : AntiVirusProduct
__SUPERCLASS              :
__DYNASTY                 : AntiVirusProduct
__RELPATH                 : AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-
DA132C1ACF46}"
__PROPERTY_COUNT          : 6
__DERIVATION              : {}
__SERVER                  : STACKOVERFLOW-PC
__NAMESPACE               : ROOT\SecurityCenter2
__PATH                    : \\STACKOVERFLOW-
PC\ROOT\SecurityCenter2:AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
displayName               : Windows Defender
instanceGuid              : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe    : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe  : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState              : 397568
timestamp                 : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName            : STACKOVERFLOW-PC
```

Clases y espacios de nombres

Hay muchas clases disponibles en CIM y WMI que están separadas en múltiples espacios de nombres. El espacio de nombres más común (y predeterminado) en Windows es `root/cimv2` . Para encontrar la clase correcta, puede ser útil hacer una lista de todos o buscar.

Listar clases disponibles

Puede enumerar todas las clases disponibles en el espacio de nombres predeterminado (`root/cimv2`) en una computadora.

3.0

CIM:

```
Get-CimClass
```

WMI:

```
Get-WmiObject -List
```

Buscar una clase

Puedes buscar clases específicas usando comodines. Ej: encontrar clases que contengan el `process` palabra.

3.0

CIM:

```
> Get-CimClass -ClassName "*Process*"

   NameSpace: ROOT/CIMV2

CimClassName                        CimClassMethods      CimClassProperties

------------                        ---------------      ------------------

Win32_ProcessTrace                  {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStartTrace             {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStopTrace              {}                   {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
CIM_Process                         {}                   {Caption, Description, InstallDate,
Name...}
Win32_Process                       {Create, Terminat... {Caption, Description, InstallDate,
Name...}
CIM_Processor                       {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
Win32_Processor                     {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
...
```

WMI:

```
Get-WmiObject -List -Class "*Process*"
```

Listar clases en un espacio de nombres diferente

El espacio de nombres de la raíz se llama simplemente `root` . Puede listar las clases en otro espacio de nombres usando el parámetro `-NameSpace` .

3.0

CIM:

```
> Get-CimClass -Namespace "root/SecurityCenter2"

   NameSpace: ROOT/SecurityCenter2

CimClassName                        CimClassMethods      CimClassProperties

------------                        ---------------      ------------------
....
AntiSpywareProduct                  {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
AntiVirusProduct                    {}                   {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
```

```
FirewallProduct                       {}                    {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
```

WMI:

```
Get-WmiObject -Class "__Namespace" -Namespace "root"
```

Listar espacios de nombres disponibles

Para encontrar los espacios de nombres secundarios disponibles de la `root` (u otro espacio de nombres), consulte los objetos en la clase `__NAMESPACE` para ese espacio de nombres.

3.0

CIM:

```
> Get-CimInstance -Namespace "root" -ClassName "__Namespace"

Name            PSComputerName
----            --------------
subscription
DEFAULT
CIMV2
msdtc
Cli
SECURITY
HyperVCluster
SecurityCenter2
RSOP
PEH
StandardCimv2
WMI
directory
Policy
virtualization
Interop
Hardware
ServiceModel
SecurityCenter
Microsoft
aspnet
Appv
```

WMI:

```
Get-WmiObject -List -Namespace "root"
```

Lea WMI y CIM en línea: https://riptutorial.com/es/powershell/topic/6808/wmi-y-cim

Creditos

S. No	Capítulos	Contributors
1	Empezando con PowerShell	4444, autosvet, Brant Bobby, Chris N, Clijsters, Community, DarkLite1, DAXaholic, Eitan, FoxDeploy, Gordon Bell, Greg Bray, Ian Miller, It-Z, JNYRanger, Jonas, Luboš Turek, Mark Wragg, Mathieu Buisson, Mrk, Nacimota, oɔɯəɹ, Poorkenny, Sam Martin, th1rdey3, TheIncorrigible1, Tim, tjrobinson, TravisEz13, vonPryz, Xalorous
2	¿Cómo descargar el último artefacto de Artifactory usando el script de Powershell (v2.0 o inferior)?	ANIL
3	Alias	jumbo
4	Análisis CSV	Andrei Epure, Frode F.
5	Anonimizar IP (v4 y v6) en un archivo de texto con Powershell	NooJ
6	Automatización de infraestructura	Giulio Caccin, Ranadip Dutta
7	Ayuda basada en comentarios	Christophe
8	Bucles	Blockhead, Christopher G. Lewis, Clijsters, CmdrTchort, DAXaholic, Eris, Frode F., Gomibushi, Gordon Bell, Jay Bazuzi, Jon, jumbo, mákos, Poorkenny, Ranadip Dutta, Richard, Roman , SeeuD1, Shawn Esterman, StephenP, TessellatingHeckler, TheIncorrigible1, VertigoRay
9	Cambiar la declaración	Anthony Neace, Frode F., jumbo, oɔɯəɹ, Ranadip Dutta, TravisEz13
10	Clases de PowerShell	boeprox, Brant Bobby, Frode F., Jaqueline Vanek, Mert Gülsoy, Ranadip Dutta, xvorsx
11	Cmdlet Naming	TravisEz13
12	Codificar /	VertigoRay

	Decodificar URL	
13	Comportamiento de retorno en PowerShell	Bert Levrau, camilohe, Eris, jumbo, Ranadip Dutta, Thomas Gerot
14	Comunicación TCP con PowerShell	autosvet, RamenChef, Richard
15	Comunicarse con APIs RESTful	autosvet, Clijsters, HAL9256, kdtong, RamenChef, Ranadip Dutta, Sam Martin, YChi Lu
16	Configuración del estado deseado	autosvet, CmdrTchort, Frode F., RamenChef
17	Conjuntos de parámetros	Bert Levrau, Poorkenny
18	consultas de powershell sql	Venkatakrishnan
19	Convenciones de nombres	niksofteng
20	Creación de recursos basados en clases DSC	Trevor Sullivan
21	Cumplimiento de requisitos previos de script	autosvet, Frode F., jumbo, RamenChef
22	Ejecutando ejecutables	RamenChef, W1M0R
23	Enviando email	Adam M., jimmyb, megamorf, NooJ, Ranadip Dutta, void, Yusuke Arakawa
24	Expresiones regulares	Frode F.
25	Firma de Scripts	AP., Frode F.
26	Flujos de trabajo de PowerShell	Trevor Sullivan
27	Funciones de PowerShell	Bert Levrau, Eris, James Ruskin, Luke Ryan, niksofteng, Ranadip Dutta, Richard, TessellatingHeckler, TravisEz13, Xalorous

28	Gestión de paquetes	TravisEz13
29	GUI en Powershell	Sam Martin
30	HashTables	Florian Meyer, Ranadip Dutta, TravisEz13
31	Incrustar código gestionado (C # | VB)	ajb101
32	Instrumentos de cuerda	Frode F., restless1987, void
33	Introducción a Pester	Frode F., Sam Martin
34	Introducción a Psake	Roman
35	Línea de comandos de PowerShell.exe	Frode F.
36	Lógica condicional	Liam, lloyd, miken32, TravisEz13
37	Los operadores	Anthony Neace, Bevo, Clijsters, Gordon Bell, JPBlanc, Mark Wragg, Ranadip Dutta
38	Manejo de errores	Prageeth Saravanan
39	Manejo de secretos y credenciales	4444, briantist, Ranadip Dutta, TravisEz13
40	Módulo ActiveDirectory	Lachie White
41	Módulo de archivo	James Ruskin, RapidCoder
42	Módulo de SharePoint	Raziel
43	Módulo de tareas programadas	Sam Martin
44	Módulo ISE	Florian Meyer
45	Módulos Powershell	autosvet, Mike Shepard, TravisEz13, Trevor Sullivan
46	Módulos, Scripts y Funciones.	Frode F., Ranadip Dutta, Xalorous
47	MongoDB	Thomas Gerot, Zteffer
48	Operadores	TravisEz13

	Especiales	
49	Parámetros comunes	autosvet, jumbo, RamenChef
50	Parámetros dinámicos de PowerShell	Poorkenny
51	Perfiles de Powershell	Frode F., Kolob Canyon
52	PowerShell "Streams"; Depuración, detallado, advertencia, error, salida e información	DarkLite1, Dave Anderson, megamorf
53	Powershell Remoting	Avshalom, megamorf, Moerwald, Sam Martin, ShaneC
54	Propiedades calculadas	Prageeth Saravanan
55	PSScriptAnalyzer - Analizador de scripts de PowerShell	Mark Wragg, mattnicola
56	Reconocimiento de Amazon Web Services (AWS)	Trevor Sullivan
57	Salpicaduras	autosvet, Frode F., Moerwald, Petru Zaharia, Poorkenny, RamenChef, Ranadip Dutta, TravisEz13, xXhRQ8sD2L7Z
58	Seguridad y criptografía	YChi Lu
59	Servicio de almacenamiento simple de Amazon Web Services (AWS) (S3)	Trevor Sullivan
60	Set básico de operaciones	Euro Micelli, Ranadip Dutta, TravisEz13
61	Trabajando con archivos XML	autosvet, Frode F., Giorgio Gambino, Lieven Keersmaekers, RamenChef, Richard, Rowshi

62	Trabajando con la tubería de PowerShell	Alban, Atsch, Clijsters, Deptor, James Ruskin, Keith, oɔɯəɹ, Sam Martin
63	Trabajando con objetos	Chris N, djwork, Mathieu Buisson, megamorf
64	Trabajos de fondo de PowerShell	Clijsters, mattnicola, Ranadip Dutta, Richard, TravisEz13
65	Usando clases estáticas existentes	Austin T French, briantist, motcke, Ranadip Dutta, Xenophane
66	Usando la barra de progreso	Clijsters, jumbo, Ranadip Dutta
67	Usando ShouldProcess	Brant Bobby, Charlie Joynt, Schwarzie2478
68	Uso del sistema de ayuda	Frode F., Madniz, mattnicola, RamenChef
69	Variables automáticas	Brant Bobby, jumbo, Mateusz Piotrowski, Moerwald, Ranadip Dutta, Roman
70	Variables automáticas - parte 2	Roman
71	Variables de entorno	autosvet
72	Variables en PowerShell	autosvet, Eris, Liam, Prageeth Saravanan, Ranadip Dutta, restless1987, Steve K
73	Variables incorporadas	Trevor Sullivan
74	WMI y CIM	Frode F.